Game Development with Godot 4: A First Course

From your first node to a finished point-and-click adventure with Popochiu

y2kmallrat · First edition, October 2026
Written for Godot 4.7 (stable) and Popochiu 2.2.1

An unofficial guide. This book is an independent educational work. It is not produced, endorsed, or sponsored by the Godot Foundation, by Carenalga, or by the Popochiu project or Godot Engine contributors. All text in it was written for this book.

Godot Engine is Copyright (c) 2014-present Godot Engine contributors and Copyright (c) 2007-2014 Juan Linietsky and Ariel Manzur, released under the MIT License. Popochiu is a free, open-source Godot plugin by Carenalga and contributors, released under the MIT License. Thank you to both communities.

"Godot" and the Godot logo are trademarks of the Godot Foundation. The book uses the names only to refer to the software.

Copyright (c) 2026 y2kmallrat. The text, diagrams, screenshots and cover art of this book are licensed under the Creative Commons Attribution-NonCommercial 4.0 International license (CC BY-NC 4.0, https://creativecommons.org/licenses/by-nc/4.0/). You may share and adapt them with credit to the author, but not for commercial use. The example code is released under the MIT License.

Contents

Part I: Foundations

Chapter 1: Game Engines and Godot

1.1 What a game engine is

1.2 Why Godot

1.3 Choosing your Godot: versions, builds, and languages

1.4 What your computer needs

1.5 Downloading and installing Godot on Windows

1.6 The Project Manager

1.7 Creating your first project

1.8 Anatomy of a project folder

1.9 A first look at what you made

Lab 1: Install Godot and show a window with a label

Chapter 2: The Editor, Nodes, and Scenes

2.1 A tour of the editor

2.2 The workspaces

2.3 Nodes

2.4 The node type hierarchy

2.5 Scenes: trees you can save and reuse

2.6 Composition, inheritance, and "everything is a node"

2.7 The Inspector and properties

2.8 Saving and running

2.9 Watching the running game: the Remote scene tree

2.10 A first, tiny script

Lab 2: Build the "Hello Room" scene

Chapter 3: GDScript I: Variables, Types, and the Life of a Script

3.1 Scripts, nodes, and the script editor

3.2 Talking to the Output panel

3.3 Variables, constants, and built-in types

3.4 Static typing and type inference

3.5 Operators

3.6 Strings, formatting, and enums

3.7 The life of a script: lifecycle functions

Lab 3: Hero Report Card

Chapter 4: GDScript II: Decisions, Loops, Functions, and Collections

4.1 Making decisions: if and match

4.2 Loops

4.3 Functions

4.4 Arrays

4.5 Dictionaries

4.6 Algorithms and randomness

4.7 Meet LineEdit and Button

Lab 4: Number Guessing Game

Chapter 5: GDScript III: Classes, Signals, and Autoloads

5.1 Every script is a class

5.2 @export and @onready

5.3 Finding nodes

5.4 Signals
5.5 Groups and get_tree()
5.6 await: waiting without freezing
5.7 Autoloads, call down and signal up, and composition
Lab 5: The Light Switch

Part II: Making 2D Games

Chapter 6: 2D Basics and Input
6.1 The 2D world: coordinates and Node2D
6.2 Local vs. global, and parent-child transforms
6.3 Sprite2D and textures
6.4 Placeholder art with built-in nodes
6.5 Time and movement: _process and delta
6.6 Camera2D
6.7 Input: the Input Map and reading actions
Fragment: add an action named "jump" mapped to the Space bar.
6.8 Events: _input and _unhandled_input
6.9 The mouse and Vector2 math
Lab 6: Move and Aim
Chapter 7: Physics and Collisions
7.1 Physics versus process
7.2 The four physics bodies
7.3 Collision layers and masks
7.4 CharacterBody2D and move_and_slide
7.5 Gravity, jumping, and platformer movement
7.6 RigidBody2D, briefly
7.7 Area2D: detecting overlaps
7.8 TileMapLayer, raycasts, and debugging
Lab 7: Coin Dash
Chapter 8: User Interface
8.1 Two families of nodes: Control and Node2D
8.2 Anchors, offsets and size flags
8.3 Containers
8.4 The common widgets
8.5 Themes and theme overrides
8.6 Focus, keyboard and gamepad navigation
8.7 CanvasLayer, pausing, scene switching and stretching
Lab 8: Main menu, options and HUD for a small game
Chapter 9: Animation and Audio
9.1 AnimationPlayer: animating anything
9.2 AnimatedSprite2D and SpriteFrames
9.3 Tween: animation by code
9.4 Timers, particles and screen shake
9.5 Audio
9.6 Putting it together: picking the right tool
Lab 9: An animated, sound-enabled door and button
Chapter 10: Resources and Saving
10.1 Nodes and Resources
10.2 Custom Resources

10.3 preload, load and PackedScene
10.4 Files: res://, user:// and FileAccess
10.5 JSON
10.6 ConfigFile for settings
10.7 Designing a save system
Lab 10: Save and load for Coin Dash
Chapter 11: Professional Habits
11.1 Debugging
11.2 Version control with git
11.3 Project structure and naming
11.4 Designing a small game
11.5 Exporting to Windows
11.6 Licenses and credits
11.7 Using an AI assistant as a tutor, responsibly
Lab 11: Break it, fix it, ship it
Part III: Point-and-Click Adventures with Popochiu
Chapter 12: Adventure Game Design
12.1 A short history of the genre
12.2 Verbs or one click?
12.3 The puzzle: a lock and a key
12.4 Dependency graphs
12.5 Fair play: how to design puzzles that are not insulting
12.6 Failure should cost little
12.7 Game flow: pacing a small adventure
12.8 Scope: the thing that kills most projects
12.9 Writing for adventure games
12.10 Paper design of "Weird Hotel"
Lab 12: Write the Weird Hotel design document
Chapter 13: Installing Popochiu and How It Is Built
13.1 What a plugin is
13.2 Choosing the right versions
13.3 Installing step by step
13.4 Tour of the dock
13.5 The generated folders
13.6 The ten autoloads
13.7 How Popochiu scripts relate to Godot scenes
13.8 Pitfalls
Lab 13: Create the weird-hotel project and install Popochiu
Chapter 14: Rooms, Characters, Props and Hotspots
14.1 What a room is
14.2 Backgrounds and placeholder pictures
14.3 Walkable areas and markers
14.4 The player character
14.5 Props, hotspots and regions
14.6 Clicks, right-clicks and speech
14.7 Room hooks and travel between rooms
14.8 Worked example: the Drawer, as first written
14.9 Where does the game remember things?

Lab 14: Bedroom, Hallway, Player, Drawer and Door
Chapter 15: Inventory, Puzzles, and State
15.1 Inventory items in Popochiu
15.2 The inventory GUI and using items
15.3 Lock-and-key logic
15.4 The state problem and its solution
15.5 Pickups in the Hallway: Bone and Twine
15.6 Combining items
15.7 Testing puzzles systematically
Lab 15: The key, door, and flute chain
Chapter 16: Dialogue Trees and Cutscenes
16.1 Dialog trees and their design
16.2 Writing good dialogue and voice
16.3 Dialogs and characters in Popochiu
16.4 Scripting the Frog conversation
16.5 Character actions, sequences, and the queue
16.6 Camera effects and the Hallway intro
16.7 Testing dialogue
Lab 16: The Frog and the first-visit intro
Chapter 17: Mini-games and the Boss Fight
17.1 Why a plain Godot scene inside an adventure?
17.2 The contract pattern
17.3 Layers, blocking, and input
17.4 The Jar: catch the bug
17.5 The Input Map and the boss fight
17.6 The Ceiling, death, and respawn
17.7 Difficulty tuning and game feel
17.8 Testing alone, and writing robust code
Lab 17: Both mini-games working in Weird Hotel
Chapter 18: Saving, Polish, Testing and Export
18.1 Saving and loading in Popochiu
18.2 Making your own data save-safe
18.3 Audio in Popochiu
18.4 Transitions, camera shake, and idle lines
18.5 Accessibility
18.6 Playtesting with real players
18.7 Exporting to Windows
18.8 Licenses, credits, and what comes next
Lab 18: Save, polish and ship Weird Hotel

Appendices

Appendix A: GDScript 4 Quick Reference
Appendix B: Godot Editor Shortcuts and Tips
Appendix C: Troubleshooting and Error-Message Dictionary
Appendix D: Answers to Review Questions
Appendix E: Glossary
Appendix F: Working with Claude Code as a Tutor
Appendix G: Weird Hotel Complete Script Listings

Preface

This book teaches you to make games with Godot, a free and open-source game engine. It assumes you have never opened Godot before. It does assume you have met the basic ideas of programming, such as variables, loops and functions, in a language like Java, C or Python. If you have not, Chapters 3 and 4 move slowly enough to follow, but expect to reread them.

The book has two aims. The first is to give you a real foundation in Godot: the editor, the scene system, the GDScript language, 2D graphics, physics, interfaces, animation, sound, saving data and shipping a finished program. These skills transfer to any kind of 2D game. The second aim is to apply that foundation to one genre, the point-and-click adventure, using a Godot plugin called Popochiu. By the last chapter you will have built *Weird Hotel*, a small, strange adventure game with two rooms, an inventory, item puzzles, a talking frog, a mini-game, a boss fight, death and respawn, save files and a Windows build.

Like a school textbook, this book covers more than the final project strictly needs. Part II teaches physics bodies, tilemaps and menus even though an adventure game uses little of them. You will need them for your second game, and Popochiu makes much more sense once you know what it is built on.

How this book is organized

Part	Chapters	What you learn
I. Foundations	1-5	What an engine is, the Godot editor, nodes and scenes, and the GDScript language from variables to signals
II. Making 2D games	6-11	Movement and input, physics and collisions, user interfaces, animation and audio, resources and saving, debugging, version control and export
III. Adventure games with Popochiu	12-18	Adventure game design, Popochiu's architecture, rooms and characters, inventory puzzles, dialogue, mini-games and a boss fight, saving, polish and release
Appendices	A-G	Reference material: GDScript summary, shortcuts, troubleshooting, answers, glossary, working with an AI tutor, complete Weird Hotel listings

How each chapter works

Every chapter follows the same pattern, so you always know where you are:

1. **Learning objectives**: what you will be able to do by the end.
2. **Key terms**: the new words, defined up front.
3. **Numbered sections**: the lesson itself, with worked examples and short **Try it** activities.
4. **Lab**: a longer hands-on project with numbered steps and an expected result.
5. **Chapter summary**: the main ideas in a few bullets.
6. **Review questions**: check your understanding. Answers are in Appendix D.
7. **Exercises**: graded **Practice** (easy), **Apply** (medium) and **Challenge** (hard).
8. **Common mistakes**: symptom, cause and fix for the errors beginners hit most.
9. **Further reading**: official documentation pages to go deeper.

Callout boxes appear throughout:

> **Note:** extra detail worth knowing.
>
> **Warning:** something that can lose work or cause confusing bugs.
>
> **Why it works:** the reasoning behind a technique.
>
> **Try it:** a small activity to do right now.

How to study with this book

- **Type the code; don't paste it.** Typing forces you to read every line, and the typos you make teach you to read error messages.
- **Run often.** Every lab step leaves the project in a working state. Run the game after each step, not only at the end.
- **Make it work with boxes first.** Use plain shapes and placeholder text. Gameplay is the point; art comes last.
- **Commit after every working step.** Chapter 11 teaches git. From Part III on, every lab ends with a commit.
- **Do the review questions before checking Appendix D.** Getting one wrong and then understanding why is worth more than reading the answer.
- **When stuck, ask for an explanation, not a fix.** Appendix F shows how to use an AI assistant as a tutor rather than a ghostwriter.

About versions

Godot and Popochiu both change over time. This book was written against **Godot 4.7** and **Popochiu 2.2.1**, the stable pairing as of October 2026. Popochiu only supports specific Godot versions, so Chapter 13 shows you how to check the version table before installing. If a menu name or button label on your screen differs slightly from the book, the idea is almost always the same. Where the book could not confirm a detail for your exact version, it says so and tells you where to check.

A suggested 15-week schedule

For a one-semester course, or for self-study at about 4-6 hours a week:

Week	Chapters	Milestone
1	1-2	Godot installed; first scene built and run
2	3	First scripts; types and the script lifecycle
3	4	Number-guessing game (Lab 4)
4	5	Light-switch system with signals and an autoload (Lab 5)
5	6	Move-and-aim prototype (Lab 6)
6	7	Coin Dash (Lab 7)
7	8-9	Menus, HUD, animation and sound
8	10-11	Save system, git, debugging, first Windows export. **Midterm project:** a small original 2D game
9	12	Weird Hotel design document

Week	Chapters	Milestone
10	13-14	Popochiu installed; two rooms with a walking player
11	15	Key, door and flute puzzles with persistent state
12	16	Frog dialogue and the hallway cutscene
13	17	Bug-catching mini-game and boss fight
14	18	Saving, polish, playtesting, export
15	Capstone	Your own short adventure game, graded with the rubric in Chapter 18

Part I: Foundations

Chapter 1: Game Engines and Godot

Learning objectives

By the end of this chapter you will be able to:

- Explain what a game engine is and describe the game loop in your own words.
- Compare Godot with other engines and justify why this book uses it.
- Choose the correct Godot download (version 4.x, Standard build, GDScript), install it on Windows, and open the Project Manager.
- Create a new project, choose a renderer, and explain what each renderer is for.
- Identify the files and folders Godot creates (`project.godot`, `res://`, `.godot/`, `.import` files, `.gitignore`) and say which ones belong in version control.
- Build and run a minimal project that opens a window containing a label.

Key terms

- **Game engine**: a program that supplies the shared machinery of a game (drawing, sound, input, physics, file loading) so you can focus on your own game.
- **Game loop**: the cycle of reading input, updating the game world, and drawing a frame, repeated many times per second.
- **Frame**: one picture drawn to the screen; a game at 60 frames per second draws 60 of them each second.
- **Open source**: software whose source code is public and may be used and changed under a license.
- **MIT license**: a short, permissive license that lets anyone use, change, and sell software made with the code, as long as the license notice is kept.
- **GDScript**: Godot's own scripting language, designed for the engine and similar in feel to Python.
- **Project Manager**: the first window Godot shows; it lists, creates, imports, and opens your projects.
- **Renderer**: the part of the engine that turns your scene into pixels using the graphics card.
- **Project**: a folder that contains a `project.godot` file plus everything your game needs.
- `res://`: the path prefix that means "the root of the current project folder".
- **Asset**: any non-code file your game uses, such as an image, sound, or font.
- **Import**: the step where Godot converts an asset into a form it can load quickly.

1.1 What a game engine is

Imagine you want to make a small game where a character walks across a screen. Without an engine you would have to write code to open a window, ask the operating system for keyboard presses, load a picture from disk, send that picture to the graphics card, keep time so the game runs at the same speed on every computer, play a sound, and close everything safely when the player quits. None of that is your game. It is the plumbing under your game, and every game needs the same plumbing.

A **game engine** is software that provides that plumbing in a reusable form. You describe your game (a character, a room, rules about what happens when they touch a door) and the engine handles the rest. Most engines supply at least these parts:

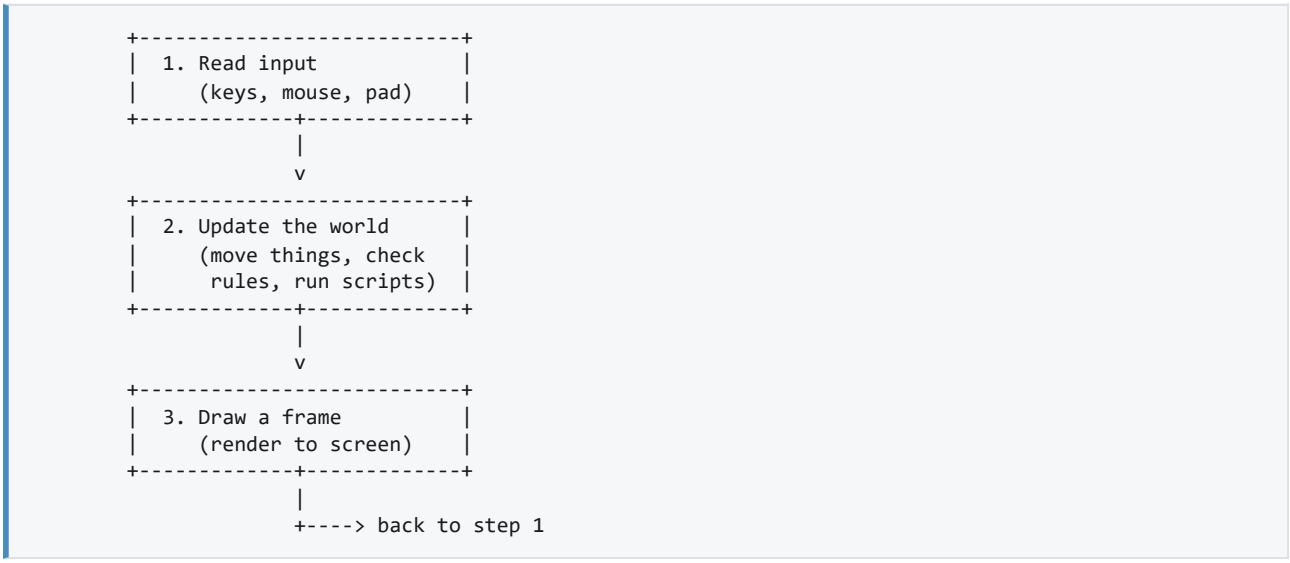
Part	What it does	Example in a game
Rendering	Draws images, shapes, text, and 3D models on screen	Showing a room and a character
Input	Reads keyboard, mouse, touch, and game controllers	Clicking on a door
Physics	Moves objects and detects when they touch	A ball bouncing off a wall
Audio	Plays and mixes sound and music	Footsteps, a background tune
Resource loading	Reads images, sounds, and data files from disk	Loading the level
Scene management	Organizes game objects and decides what exists right now	Switching from a menu to a room
Editor tools	Lets you build levels and set values without writing everything in code	Dragging a sprite into place
Export	Packages the game so other people can run it	Producing a Windows `.exe`

Engines range from small libraries that only draw and play sound, to huge tools with a full visual editor. Godot is the second kind: it includes an editor, a scripting language, a 2D and 3D renderer, physics, audio, and exporting to many platforms, all in one download.

> **Note:** An engine is not a game. It is closer to a kitchen than to a meal. A good kitchen makes cooking easier, but you still have to decide what to cook.

The game loop

Almost every game, in any engine, runs on the same idea: the **game loop**. A loop repeats these three jobs over and over until the player quits.



One trip around the loop produces one **frame**. A game that runs at 60 frames per second goes around this loop 60 times every second, so each trip has about 16 milliseconds to finish. If one trip takes longer, the game feels slow or jumpy.

You rarely write this loop yourself in Godot. The engine owns it. Instead, you write small pieces of code that the engine calls at the right moment: "when you are ready, run this", "every frame, run that", "when a key is pressed, run this other thing". This pattern, where the framework calls your code instead of your code calling the framework, is called *inversion of control*. If you have used event handlers in Java (for example a button click listener), you have seen the same idea.

Godot actually runs two loops side by side. The **process** loop runs once per drawn frame, and its speed depends on how fast the computer is. The **physics** loop runs at a fixed rate, 60 times per second by default, so that movement and collisions behave the same on fast and slow machines. You will use both from Chapter 6 onward. For now, remember only this: *the engine runs a loop, and your scripts are called from inside it.*

> **Try it:** Open any simple game you own, or even a web game. Think about the three loop jobs. What input does it read? What changes each frame? What would you notice if the "draw" step stopped but the "update" step kept running? (Answer: the game would still be running, but the screen would freeze.)

What an engine saves you

A real example helps. The following table estimates what you would need to build yourself to show a moving character without an engine, versus with Godot.

Task	Without an engine	With Godot
Open a window	Call operating-system code (different on every OS)	Automatic when the game starts
Draw a picture	Learn a graphics API and write shaders	Add a `Sprite2D` node and pick a picture
Move on key press	Poll the keyboard every frame yourself	Check an input action in a script
Make the game run on Mac and Linux	Rewrite or port large parts	Choose another export target
Detect two shapes touching	Write collision math	Add collision shapes and read a signal

This is why beginners and professionals both use engines: the saved time is enormous. The trade-off is that you must learn the engine's way of organizing things. That is what Chapters 1 to 5 are about.

1.2 Why Godot

There are several well-known engines. Unity, Unreal Engine, GameMaker, and Godot are common choices, and each suits different projects. This book uses Godot for reasons that matter to a learner.

Godot is free and open source

Godot is **open source**: its full source code is public, anyone can read it, and anyone can propose improvements. It is released under the **MIT license**, one of the most permissive licenses in common use. In plain words, the MIT license means:

- You may download and use Godot for any purpose, including commercial games, at no cost.
- You do not pay royalties or a share of your sales.

- You do not have to show a Godot logo or ask permission (though crediting the engine is polite and common).
- You may copy and modify the engine's source code, as long as you keep the license notice.

Compare this with engines that have subscription fees, revenue thresholds, or license terms that can change. With Godot, the terms attached to the version you downloaded cannot be changed later; a copy you have is yours to keep using. The engine is developed in the open by a community and funded through donations and sponsorships, with a non-profit foundation looking after the project.

> **Note:** MIT covers the engine. Your *game* is yours, and you choose its license. Art, music, and fonts you download from other places have their own licenses; always check them.

Godot is small and quick

The Godot editor is a single program, with no installer, that is small by modern standards. It starts in seconds, uses modest memory, and can run on older laptops. Compare this with engines that take many gigabytes and a long first launch. For a learner with an ordinary computer, that makes a real difference to how often you are willing to open the tool and try an idea.

Godot is built from nodes and scenes

Godot's central idea is that a game is a *tree of small building blocks* called **nodes**, saved in files called **scenes**. A node does one job: show a picture, play a sound, detect a click, hold text. You combine nodes into trees, and you combine trees into bigger trees. Chapter 2 teaches this properly. The key benefit is that the same simple idea (nodes in trees) applies to a menu, a character, a whole level, and a bullet.

Godot has a language designed for it

Godot comes with **GDScript**, a scripting language made for the engine. It is readable, it uses indentation instead of braces, and it understands Godot's types directly. Because the editor and the language are designed together, you get helpful features, such as auto-complete for node names and the built-in documentation, without any setup.

Godot is excellent for 2D

Many engines treat 2D as a flattened version of 3D. Godot has a dedicated 2D engine that works in real pixels. This makes it a very good choice for pixel art, platformers, top-down games, and the point-and-click adventure you will build in Part III of this book. Godot also does 3D, but this book stays in 2D.

Honest trade-offs

No engine is best at everything. Be aware of these limits:

- Godot's 3D tools are improving, but huge, high-end 3D games with advanced graphics are not its strongest area.
- The ecosystem of ready-made add-ons and tutorials is smaller than for the biggest engines, though it grows every year.
- Some console platforms need extra steps or third-party help for publishing, because console makers restrict their development kits.

- Because Godot changes fast, older tutorials can be out of date. This is why this book teaches the Godot 4 way of doing things, and why you should always check the version a tutorial was written for.

For a first course in game development, the benefits (free, light, simple model, great 2D) clearly outweigh these limits.

1.3 Choosing your Godot: versions, builds, and languages

When you visit the download page you meet a few choices. Make them once, correctly, and you will not think about them again.

Version 3 or version 4?

Godot has two major generations in common use.

	Godot 3.x	**Godot 4.x**
Status	Older, still receives limited fixes	Current, actively developed
Scripting	GDScript 1.0	GDScript 2.0 (faster, typed, lambdas, `await`)
2D and 3D rendering	OpenGL based	New renderers (Forward+, Mobile, Compatibility)
Names of many nodes	`Sprite`, `KinematicBody2D`	`Sprite2D`, `CharacterBody2D`
Tutorials online	Many, but outdated	Growing fast

This book uses Godot 4 only. Everything, from node names to script syntax, is Godot 4. Use **the latest stable Godot 4.x release** (for example, if the newest stable version when you read this is "4.7", use that). Do not worry about the exact minor number. The first number, 4, is what matters; the second number changes with each feature release, and the lessons in this book are written to stay valid across 4.x releases.

> **Warning:** If you search the web for help and see `onready var`, `yield`, `export var`, `KinematicBody2D`, or `connect("pressed", self, "_on_pressed")`, that is Godot 3 code. It will not work in Godot 4. Look for a Godot 4 answer instead.

Stable, beta, and dev builds

The download page offers a **stable** release and, under other tabs, pre-release builds named *dev*, *beta*, and *release candidate*. Pre-release builds are for testers and may break your project. As a learner, always choose **stable**.

Standard or .NET?

Each stable release comes in two flavors:

	Standard	**.NET**
Scripting languages	GDScript (and C++ via extensions)	GDScript **and** C#
Needs extra software	No	Yes, the .NET SDK must be installed separately
Download size	Smaller	Larger

Choose **Standard**. A project made in either flavor can usually be opened in the other, but only the .NET build can run C# scripts. You do not need C# for this book.

GDScript or C#?

If you know Java or C, you may ask whether you should use C#. Here is the comparison.

	GDScript	C#
Setup	None	Install the .NET SDK, an editor such as Visual Studio or Rider
Fit with the engine	Excellent: made for it	Good, but sits on top
Learning curve	Gentle	Familiar to Java users, more ceremony
Speed of your own code	Fine for most 2D games	Faster for heavy calculation
Documentation and tutorials	Most of the Godot docs use it	Docs show C# alongside
This book	Used everywhere	Not used

GDScript is the right choice for beginners and for nearly all 2D games. The Popochiu add-on you will use in Part III is also written in GDScript. Once you finish this book, learning C# in Godot is easy because the node and scene ideas are identical.

1.4 What your computer needs

Godot is forgiving. You do not need a gaming computer.

Item	What you need
Operating system	A current, 64-bit version of Windows 10 or 11 (Godot also runs on macOS and Linux, but this book shows Windows)
Memory (RAM)	4 GB is workable; 8 GB or more is comfortable
Graphics	Any graphics chip from the last several years; older chips can still use the Compatibility renderer
Disk space	Under 1 GB for the editor and your first projects
Screen	1366x768 works; a larger screen gives you more room for panels
Mouse	A three-button mouse with a scroll wheel is strongly recommended; the middle button pans the 2D view
Internet	Needed once, to download Godot, and later for documentation and add-ons

If your computer is older, or your graphics driver is old, choose the **Compatibility** renderer when you create a project (see Section 1.7). It runs on the widest range of hardware.

> **Try it:** Check your Windows version. Press the Windows key, type `About your PC`, and press Enter. Note the "System type" line. It should say "64-bit operating system".

1.5 Downloading and installing Godot on Windows

Godot does not have an installer. You download one zip file, extract it, and run the program. This is called a *portable* program, and it means installing and removing Godot cannot damage the rest of your computer.

Step by step

1. Open a browser and go to **godotengine.org**. Choose **Download**.
2. Choose the **Windows** download. The page offers the latest stable release at the top. Pick the **Standard** version, not the ".NET" one.
3. Save the zip file. Its name looks like `Godot_v4.x-stable_win64.exe.zip`, where `4.x` is the version number (for example `4.7`).
4. Create a folder for the engine. A short path without spaces works well, such as `C:\Godot`.
5. Right-click the zip file and choose **Extract All...**, and extract into `C:\Godot`.
6. Open the folder. You will see two programs. They are the same engine: - `Godot_v4.x-stable_win64.exe`: the normal editor. - `Godot_v4.x-stable_win64_console.exe`: the same editor plus a text window that prints extra messages, which helps when something goes wrong.
7. Double-click the first one to start Godot.

> **Warning:** Do not run Godot from inside the zip file without extracting it. Windows may let it open, but Godot cannot save its settings, and it can fail in confusing ways. Always extract first.

First launch and Windows security

On first launch Windows may show a blue "Windows protected your PC" or a firewall prompt. This appears for many new programs that Microsoft has not seen often. If you downloaded Godot from the official site, choose **More info** and then **Run anyway**. If a firewall asks about network access, you can allow private networks; Godot only uses the network for optional features like the in-editor asset library.

Making Godot easy to find

Because there is no installer, Windows does not create a Start Menu entry. Right-click the `.exe` and choose **Show more options > Send to > Desktop (create shortcut)**, or pin it to your taskbar. When a new Godot version appears, extract the new version into its own folder next to the old one (for example `C:\Godot\4.x`). Keeping versions side by side lets old projects keep opening in the version that made them.

> **Why it works:** Godot stores its own editor settings in your user profile, not next to the program. So you can delete or replace the program folder without losing your settings.

Other ways to install

You may also get Godot through Steam, through itch.io, or through a Windows package manager such as `winget`. All give the same engine. This book assumes the zip download because it is the simplest and the least likely to differ from what you see in the text.

1.6 The Project Manager

When Godot starts and you have not told it to open a project, it shows the **Project Manager**. It is the front door of the engine. It is not the editor; it is a launcher that lists your projects.

The first time, Godot may show a short message asking about a local project list or offering to look at the asset library. Read it, then continue.

Layout

The Project Manager window has these parts. Its exact layout changes a little between 4.x releases (in some versions a button such as **Scan** moves into a menu or the Settings), so if a button below is not where you expect, look in the right-click menu of a project row or in the Settings:

Area	What it does
Tabs along the top: **Local Projects** and **Asset Library Projects**	Local Projects shows projects on your disk. The asset library tab lets you download demo projects.
Project list (center)	One row per project with its name, folder, and the Godot version it was last saved with.
Buttons on the right: **Create**, **Import**, **Scan**, **Edit**, **Run**, **Rename**, **Manage Tags**, **Remove**	Act on the project you select.
Search box and filter	Find a project by name or tag.
Settings (gear icon, top right)	Language, theme, and other options.

What the main buttons do:

- **Create** opens the dialog for a new project (Section 1.7).
- **Import** lets you browse to an existing `project.godot` file and add it to the list. Use this for a project you downloaded or copied from another computer.
- **Scan** searches a folder and adds every Godot project it finds inside.
- **Edit** opens the selected project in the editor. Double-clicking the row does the same.
- **Run** runs the selected project's game directly, without opening the editor.
- **Remove** removes the project from the *list*. It does not delete the files. (The dialog may offer an option to also delete the project folder; read it carefully.)

The version warning

Each project remembers the Godot version that last saved it. If you open a project in a *newer* version, Godot offers to convert it. After conversion, older versions may no longer open it. Always make a copy or use version control before converting an important project. If you open a project with an *older* editor than the one that made it, Godot warns you, because it may not understand the newer files.

1.7 Creating your first project

Click **Create**. A dialog titled **Create New Project** appears. It has these fields:

Field	Meaning
Project Name	A name for the game. It becomes the window title and the name shown in the list. You can change it later in Project Settings.
Project Path	The folder where the project will live. Use **Browse** to choose a parent folder.
Create Folder	Makes a new folder named after the project inside the path you chose. Leave it on.
Renderer	The graphics engine (see below).
Version Control Metadata	Choose **Git** to have Godot create a `.gitignore` and `.gitattributes`, or **None**.

> **Note:** Menu and button names shift slightly between 4.x releases. The final button in the dialog is **Create** in newer versions and **Create & Edit** in some older 4.x versions. Both do the same thing: make the folder and open the editor.

Pick an **empty** folder, or let Godot create one. Never choose a folder that already holds many other files, because Godot will treat everything in it as part of the project.

Choosing a renderer

The **renderer** decides how Godot talks to your graphics card. Godot 4 has three. The choice is made per project, and you can change it later in Project Settings, though changing it mid-project can alter how things look.

Renderer	Based on	Best for	Notes
Forward+	Modern desktop graphics APIs (Vulkan, Direct3D 12, or Metal)	Desktop games that want the most advanced 3D effects	Highest quality and the most features. Needs a reasonably modern graphics card.
Mobile	The same modern APIs, with a lighter approach	Phones, tablets, and mid-range hardware with 3D	Fewer features than Forward+, faster on weaker devices.
Compatibility	OpenGL, an older and very widely supported API	Older computers, web export, simple 2D and 3D games	Works almost everywhere. Some advanced 3D effects are missing.

For this book, which makes 2D games, **Compatibility** is the safest choice and is what the text assumes whenever it asks you to pick. It runs on old and new hardware, and 2D games look the same. If your computer is modern, Forward+ also works fine for 2D; the difference you would see is mostly in 3D lighting and effects. The Godot documentation's page on renderers describes the exact current feature differences.

> **Why it works:** A 2D game draws flat pictures. Nearly every 2D feature exists in all three renderers, so the cheapest, most compatible one is enough.

After you click **Create**, the editor opens. Chapter 2 tours it. For now, the empty editor is your goal.

1.8 Anatomy of a project folder

Open the project folder in Windows File Explorer. A brand-new project contains very few things:

```
my_first_project/
|-- .godot/                 (hidden-ish cache folder, created by the editor)
|-- .gitattributes          (only if you chose Git)
|-- .gitignore              (only if you chose Git)
|-- icon.svg                (default project icon)
|-- icon.svg.import         (Godot's import settings for icon.svg)
`-- project.godot           (the project's main settings file)
```

Each of these has a purpose.

project.godot

This is the file that *defines* a project. Godot treats any folder containing `project.godot` as a project's root. It is a plain text file, readable in any text editor, in a simple settings format. A fresh one looks like this (your version numbers will differ):

```
; Engine configuration file.
; It's best edited using the editor UI and not directly,
; since the parameters that go here are not all obvious.

config_version=5

[application]

config/name="My First Project"
config/features=PackedStringArray("4.x", "GL Compatibility")
config/icon="res://icon.svg"

[rendering]

renderer/rendering_method="gl_compatibility"
renderer/rendering_method.mobile="gl_compatibility"
```

Notice the pieces:

- `config_version=5` is the format version of the file, used by Godot 4.
- `[application]` is a section. `config/name` is the project name.
- `config/features` records the Godot version and the renderer you chose.
- `config/icon` points to an image using a `res://` path.
- Settings you change in **Project > Project Settings** are saved here. Only settings that differ from the defaults are written, which keeps the file short.

> **Warning:** The file warns that it is best edited using the editor. Do edit it by hand only if you know exactly what a setting does. A typo can stop the project from opening.

res://, the project root

Inside Godot, you will not see paths like `C:\Users\you\Documents\my_first_project\icon.svg`. Instead you see `res://icon.svg`. The prefix `res://` means "the folder that contains `project.godot`". The word comes from *resources*.

This has two benefits:

1. A project works the same wherever you put it on disk. You can move or copy the whole folder and every path inside still works.

2. The same path works after you export the game, even though the files are then packed into a single archive.

There is also `user://`, a separate folder for data that the *running game* writes, such as save files and settings. It lives in a per-user location chosen by the operating system, and you will use it in Chapter 10. Remember the difference: `res://` is for things you ship with the game and is read-only after export; `user://` is where the player's game may write.

> **Try it:** In the Godot editor, find the **FileSystem** dock (usually the lower-left panel). You will see `res://` at the top with `icon.svg` inside it. Right-click `icon.svg` and choose **Open in File Manager**. Windows opens the real folder. Compare what you see with the tree above.

The .godot folder

The `.godot` folder is created and managed by the editor. It holds:

- **Imported assets**: the converted versions of your images and sounds.
- **Caches**: scene information, shader caches, and the editor's record of your open files and layout for this project.
- **Class lists and similar data** the editor needs to start quickly.

Everything in `.godot` can be regenerated. If you delete the folder while the editor is closed, Godot rebuilds it next time you open the project (it will take a moment while assets are re-imported). Because it can be rebuilt, and because it can be large, **it should not be saved in version control or shared**.

The name starts with a dot, which means some tools treat it as a hidden folder. In File Explorer it may not appear unless you turn on **View > Show > Hidden items**.

> **Warning:** Do not hand-edit anything in `.godot`. If the editor behaves strangely, close it, delete the `.godot` folder, and reopen the project. That is a safe and common fix.

icon.svg and the .import files

Every time you add an image, sound, or other asset to your project, Godot creates a small text file next to it with the extra extension `.import`. For `icon.svg` this is `icon.svg.import`. It records *how* to import the asset: compression, filtering, and similar settings. The converted result lives in `.godot/imported/`.

```
res://
|-- icon.svg           (your original: you keep this)
`-- icon.svg.import    (settings: you keep this too)

res://.godot/imported/
`-- icon.svg-<hash>.ctex   (the converted result: rebuilt automatically)
```

The rule to remember:

File or folder	Save in version control?	Why
Original assets (`icon.svg`, `.png`, `.wav`)	**Yes**	They are your work

File or folder	Save in version control?	Why
`.import` files	**Yes**	They hold your import settings, and keep other computers consistent
Scenes (`.tscn`), scripts (`.gd`), `project.godot`	**Yes**	They are your game
`.godot/` folder	**No**	It is rebuilt automatically

You can edit an asset's import settings in the **Import** dock (a tab next to the Scene dock). After changing settings there, press **Reimport** to apply them.

.gitignore and .gitattributes

If you selected **Git** under Version Control Metadata, Godot made two small files. They matter only if you use **Git**, a program that records the history of your project so you can undo mistakes and share work. Git itself is introduced in Chapter 11. Here is what Godot wrote.

`.gitignore` tells Git which things to leave out:

```
# Godot 4+ specific ignores
.godot/
/android/
```

That is why the `.godot` folder is not tracked. The second line ignores a folder used when exporting to Android.

`.gitattributes` tells Git to normalize line endings so that a project edited on Windows and on Linux does not show every line as changed:

```
# Normalize EOL for all files that Git considers text files.
* text=auto eol=lf
```

If you did not choose Git when creating the project, you can add these files by hand later, or use **Project > Version Control > Version Control Settings...** in the editor.

1.9 A first look at what you made

You have a project, but it has no scene yet, so there is nothing to run. The next chapter explains scenes in depth. In the lab below you make just enough of one to see the engine work end to end: create a scene, put text in it, mark it as the one to start with, and run.

Three small ideas are needed first.

- A **node** is one building block of your game. In the lab you use two kinds: a `Control` node (a general user-interface element) and a `Label` node (shows text).
- A **scene** is a tree of nodes saved in a file whose name ends in `.tscn`. Chapter 2 covers scenes properly.
- The **main scene** is the scene Godot starts when you press the play button for the whole project. If none is chosen, Godot asks you the first time.

Do not worry if these feel loose. Doing the lab once makes them concrete.

Lab 1: Install Godot and show a window with a label

Goal: Install Godot, create a project, and run it so that a window opens with the text "Hello, Godot!" inside.

Time: 30 to 45 minutes.

What you need: Windows, an internet connection, about 200 MB free disk space.

Part A: Install

1. In a browser, go to **godotengine.org/download** and choose Windows. Download the latest stable **Standard** build (not .NET).
2. Create the folder `C:\Godot`.
3. Extract the zip into `C:\Godot`.
4. Double-click `Godot_v4.x-stable_win64.exe`. If Windows warns you, choose **More info > Run anyway**.
5. Confirm you see the **Project Manager**.

Part B: Create the project

6. Click **Create**.
7. In **Project Name** type `hello_godot`.
8. For **Project Path**, click **Browse** and choose a folder such as `C:\GodotProjects`. Create it first in File Explorer if it does not exist. Leave **Create Folder** on.
9. Under **Renderer**, choose **Compatibility**.
10. Under **Version Control Metadata**, choose **Git**.
11. Click **Create** (or **Create & Edit**). The editor opens.

Part C: Make a scene with a label

12. In the **Scene** dock (top left), you see the text "Create Root Node:" and four buttons: **2D Scene**, **3D Scene**, **User Interface**, **Other Node**. Click **User Interface**. A node named `Control` appears as the root.
13. Right-click the root node and choose **Add Child Node**. (Or select the root and press **Ctrl+A**.)
14. In the search box type `Label`. Select **Label** and click **Create**. The tree now looks like this:

```
Control
`-- Label
```

15. Select the `Label` node. In the **Inspector** (right side) find the property **Text**. Click the box and type `Hello, Godot!`.
16. Look at the main view. The text is in the top left corner and is quite small. Make it bigger: in the Inspector open **Theme Overrides > Font Sizes** and set **Font Size** to `48`. (If the section is collapsed, click its title to expand it.)
17. Move the label into the middle of the window. You can drag it with the mouse, or set its position in the Inspector under **Layout > Transform > Position**. Try `x = 400`, `y = 260`.

Part D: Save and set the main scene

18. Press **Ctrl+S** (or choose **Scene > Save Scene**). A dialog asks where to save. Name the file `main.tscn` and save it in `res://`.
19. Press **F5** (**Run Project**), the play button at the top right. A dialog says no main scene is defined. Click **Select Current**. Godot remembers this choice.

Part E: Run and check

20. A new window opens with your text in it.
21. Close the window with the **X** button, or press **F8** in the editor to stop the game.
22. Optional: give the window a title. Open **Project > Project Settings**, and in the **Application > Config** page set **Name** to `Hello Godot`. Close the dialog and press F5 again. The title bar of the game window now reads "Hello Godot".
23. Open **Project > Project Settings** again, and under **Display > Window** note the default **Viewport Width** (1152) and **Viewport Height** (648). These control the size of the game window. Change them to `800` and `450`, run again, and move the label so it is still visible.

Part F: Look at the folder

24. In the **FileSystem** dock, right-click `main.tscn` and choose **Open in File Manager**.
25. Find the files from Section 1.8: `project.godot`, `icon.svg`, `icon.svg.import`, `.gitignore`, and your new `main.tscn`. Turn on **Show hidden items** to see `.godot`.
26. Open `project.godot` in Notepad (right-click, **Open with**, Notepad). Find `config/name` and the line that mentions your main scene (`run/main_scene`). Close it without changing anything.

Expected result: Pressing F5 opens a window whose title is the project name and whose content shows the text "Hello, Godot!" in large letters on the default dark grey background. The project folder contains `project.godot`, `main.tscn`, `icon.svg`, `icon.svg.import`, `.gitignore`, `.gitattributes`, and a `.godot` folder.

> **Note:** If the label looks cut off, the label is wider than the window. Reduce the font size or the position, or make the window bigger in Project Settings.

Chapter summary

- A game engine supplies the shared parts of a game (drawing, input, audio, physics, exporting) so you write only what is unique to your game.
- Games run a loop: read input, update the world, draw a frame. In Godot the engine owns the loop and calls your code from inside it, with a per-frame loop and a fixed-rate physics loop.
- Godot is free and open source under the MIT license, light enough for ordinary computers, built around nodes and scenes, and especially strong in 2D.
- This book uses the latest stable Godot 4.x, the Standard (not .NET) build, and GDScript. Godot 3 code and tutorials use different names and syntax.
- On Windows, Godot is a portable zip: extract it and run the `.exe`. No installer is needed.
- The Project Manager creates, imports, and opens projects. For a new 2D project, the Compatibility renderer is the safest choice.
- A project is a folder with a `project.godot` file. `res://` means that folder; `user://` is where a running game writes data.

- Keep your assets, `.import` files, scenes, scripts, and `project.godot` in version control; do not keep the `.godot` folder, which Godot rebuilds.

Review questions

1. **Multiple choice.** Which of these is *not* normally a job of a game engine? a) Drawing images on screen b) Reading keyboard and mouse input c) Deciding what your game's story should be d) Playing audio

2. **True or false.** In the game loop, the engine draws a frame before it reads input.

3. **Short answer.** In two or three sentences, describe the three main jobs repeated in the game loop.

4. **Multiple choice.** What does the MIT license allow you to do with Godot? a) Use it for commercial games without paying royalties b) Use it only for non-commercial projects c) Use it only if you display the Godot logo d) Use it only for a limited trial period

5. **True or false.** This book uses the .NET build of Godot because it is required for GDScript.

6. **Multiple choice.** Which line of code tells you immediately that a tutorial was written for Godot 3 and not Godot 4? a) `extends Node2D` b) `var speed: float = 200.0` c) `export var speed = 200` d) `func _ready() -> void:`

7. **Short answer.** What does the path prefix `res://` mean, and name one advantage of using it instead of a full Windows path.

8. **Multiple choice.** Which renderer is the most widely compatible with older hardware? a) Forward+ b) Mobile c) Compatibility d) They are all equally compatible

9. **True or false.** The `.godot` folder should be saved in version control because it holds your game's scenes.

10. **Short answer.** Why do `.import` files exist, and should they be saved in version control?

Exercises

1. Practice: Read a project file. Open the `project.godot` file of your `hello_godot` project in Notepad. Write down (a) the project name, (b) the renderer setting, and (c) the path of the main scene. Then change the project name inside Godot using **Project > Project Settings > Application > Config > Name**, save, and open the file again to see exactly which line changed. Do not edit the file by hand.

2. Practice: Rebuild the cache. Close the editor. In File Explorer, delete the `.godot` folder of `hello_godot`. Open the project again from the Project Manager. Describe what happened, how long it took, and whether the project still works. Then explain in two sentences why this is safe.

3. Apply: Two renderers. Create a second project called `renderer_test` using the **Forward+** renderer (if your computer supports it). Repeat the steps of the lab so it shows the same label. Compare the two projects: do they look different? Compare their `project.godot` files and list the lines that differ. Write a short paragraph on which you would pick for a 2D game on a very old laptop and why.

4. Apply: Make it yours. Change your `hello_godot` scene so that the window is 960 by 540 pixels, the text is in the exact center of the window, and the label shows your name. Change the text color: select the Label, then in the Inspector open **Theme Overrides > Colors** and set **Font Color**. (Hint: set **Horizontal Alignment** on the Label to center and give the Label a wider size; or work out the position from the label's size.)

5. Challenge: A tour of the docs. Open the official Godot documentation (docs.godotengine.org) and the page "Introduction to Godot". In your own words, write a half-page note answering: what does the documentation say Godot's scene system is for? Then find the "Overview of renderers" page and make a table with three rows (Forward+, Mobile, Compatibility) and a column "One thing it is best at" using only what the documentation says.

Common mistakes

- **Symptom:** Windows says the program cannot save settings, or Godot closes unexpectedly right after opening. **Cause:** Godot was run from inside the zip file without extracting it. **Fix:** Extract all files into a folder such as `C:\Godot` and run it from there.
- **Symptom:** You press F5 and nothing opens, or a dialog asks to "Select Main Scene". **Cause:** No main scene is set. **Fix:** Save your scene, press F5, and choose **Select Current**; or set **Project > Project Settings > Application > Run > Main Scene**.
- **Symptom:** You create a project and it opens with many unrelated files already in it, or Godot is slow to start the editor. **Cause:** You chose a non-empty or very large folder as the project path, and Godot scans everything below it. **Fix:** Create the project in a new, empty folder (leave **Create Folder** ticked).
- **Symptom:** A tutorial's code gives red errors such as "Unexpected identifier" at `onready` or `yield`. **Cause:** The tutorial is for Godot 3. **Fix:** Look for a Godot 4 version of the tutorial, or use the Godot 4 equivalents taught in this book (`@onready`, `await`).
- **Symptom:** You removed a project from the Project Manager and are afraid you lost it. **Cause:** **Remove** only takes the project off the list. **Fix:** Use **Import** and browse to its `project.godot` file; the files are still on disk.
- **Symptom:** You committed the `.godot` folder to Git and the repository is huge or shows many changes. **Cause:** No `.gitignore`. **Fix:** Add `.godot/` to `.gitignore` and remove the folder from Git's tracking (Chapter 11 shows how).

Further reading

- Godot documentation home: https://docs.godotengine.org/en/stable/
- Introduction to Godot: https://docs.godotengine.org/en/stable/getting_started/introduction/introduction_to_godot.html
- Overview of Godot's key concepts: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html
- The Project Manager: https://docs.godotengine.org/en/stable/tutorials/editor/project_manager.html
- Overview of renderers: https://docs.godotengine.org/en/stable/tutorials/rendering/renderers.html
- Project organization: https://docs.godotengine.org/en/stable/tutorials/best_practices/project_organization.html
- Download page: https://godotengine.org/download

Chapter 2: The Editor, Nodes, and Scenes

Learning objectives

By the end of this chapter you will be able to:

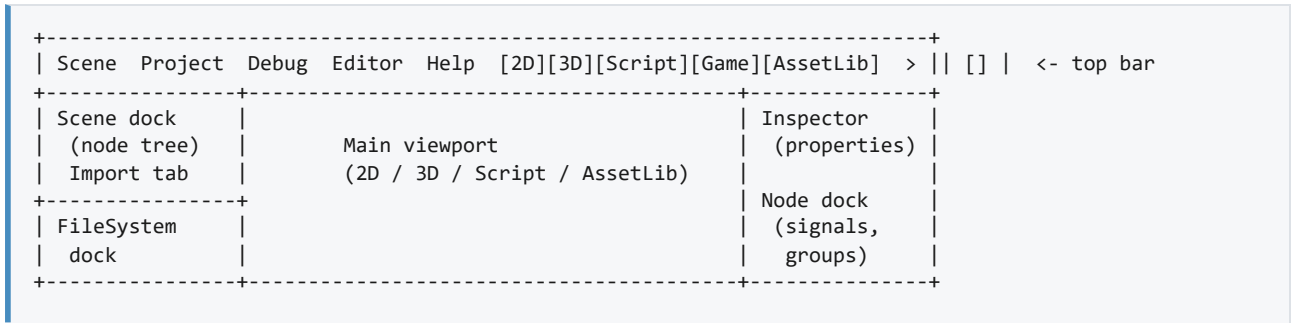
- Name the main areas of the Godot 4 editor, say what each is for, and switch between the workspaces (2D, 3D, Script, Game, AssetLib).
- Add, rename, move, and delete nodes in the scene tree, and change their properties in the Inspector.
- Explain how Node, CanvasItem, Node2D, and Control relate to each other.
- Save a scene, instance it inside another scene, and explain why this is better than copying.
- Compare composition with inheritance and apply the "everything is a node" idea.
- Run a project or a single scene with F5, F6, and F8, and inspect the running game in the Remote scene tree.
- Attach a very short script to a node.

Key terms

- **Node**: the basic building block of a Godot game; each node has a name, a type, properties, and can have children.
- **Scene**: a tree of nodes saved in a `.tscn` file; it can be used again and again.
- **Scene tree**: the live arrangement of all nodes in a running game, with parent-child relationships.
- **Root node**: the single top node of a scene.
- **Child / parent**: a node directly below (child) or above (parent) another in the tree.
- **Instance**: a copy of a saved scene placed into another scene (or the running game).
- **Property**: a named value on a node, such as `position` or `text`.
- **Inspector**: the dock that shows and edits the properties of the selected node.
- **Main scene**: the scene that starts when you run the whole project.
- **Composition**: building something complex by combining small parts.
- **Inheritance**: building something new by extending an existing type.
- **Workspace**: one of the editor's main screens (2D, 3D, Script, Game, AssetLib).

2.1 A tour of the editor

Open the `hello_godot` project from Chapter 1 (or any project). The editor window is divided into areas. Learn their names now, because the whole book refers to them.



```
| Bottom panel: Output | Debugger | Audio | Animation | Shader Editor ...    |
+-------------------------------------------------------------------------+
```

The layout is a default; you can drag the dividers to resize and drag docks to other positions. The names below are the standard ones.

The top bar

- **Menus** on the left: **Scene**, **Project**, **Debug**, **Editor**, **Help**. You will use these often:
- **Scene > New Scene**, **Open Scene...**, **Save Scene**, **Save Scene As...**, **Close Scene**.
- **Project > Project Settings...**, **Export...**, **Reload Current Project**, **Quit to Project List**.
- **Debug**: options to show collision shapes and other helpers while the game runs.
- **Editor > Editor Settings...**: settings for the editor itself (font size, theme, script editor behavior), shared by all projects.
- **Help**: the built-in class reference (**Search Help**), the online docs, and "About Godot".
- **Workspace buttons** in the middle: **2D**, **3D**, **Script**, **Game** (added in Godot 4.4; it shows the running game inside the editor), and **AssetLib**. Details in Section 2.2.
- **Run buttons** on the right: **Run Project** (play icon), **Pause**, **Stop**, **Run Current Scene**, and **Run Specific Scene**. Section 2.8 covers them.
- A **renderer drop-down** at the far right shows the renderer in use (for example, Compatibility).

The Scene dock (top left)

The **Scene dock** shows the nodes of the scene you have open, as an indented tree. This is where you add, delete, rename, and re-arrange nodes. A tab beside it, the **Import** dock, edits import settings for assets. When your project is running, the Scene dock gains **Local** and **Remote** tabs (Section 2.9).

The FileSystem dock (bottom left)

The **FileSystem dock** shows the files in your project, starting at `res://`. You can create folders, drag files into scenes, rename and move files, and double-click a `.tscn` file to open it. Moving files here (not in Windows File Explorer) is important, because Godot updates references to them for you.

The viewport (middle)

The large central area changes with the selected workspace. In the 2D workspace it is a canvas where you see and move 2D things. A toolbar sits along its top with tools: **Select**, **Move**, **Rotate**, **Scale**, plus snapping and grouping options. Navigating the 2D view:

Action	Control
Pan	Hold the middle mouse button and drag; or hold Space and drag with the left button
Zoom	Mouse wheel; or Ctrl + wheel
Select a node	Left-click it in the viewport or in the Scene dock
Focus on the selection	Press **F**

A thin rectangle outline in the 2D view shows the game window size (by default 1152 by 648 pixels). Things outside it will not be seen when the game runs.

The Inspector (top right)

The **Inspector** shows the properties of the selected node. It is where much of your work happens. It is described in Section 2.7.

The Node dock (right, beside the Inspector)

The **Node dock** has two tabs: **Signals**, listing the events the selected node can send, and **Groups**, which lists labels you can attach to nodes. You will use signals from Chapter 5 and groups in Chapter 5 as well. For now, know that it exists.

The bottom panels

The strip at the bottom has tabs that open panels:

Panel	Use
Output	Text printed by `print()` and engine messages while the game runs
Debugger	Errors and warnings (with a clickable stack trace), plus tools such as the profiler and monitors
Audio	The audio bus mixer (Chapter 9)
Animation	Appears when you select an animation node (Chapter 9)
Shader Editor	For shader code; not used in this book

Click a tab to open its panel and click it again to close it. If you ever lose a panel, check these tabs first.

> **Try it:** Click each bottom tab in turn. Which tab shows errors? (**Debugger**.) Which shows what `print()` writes? (**Output**.) Now drag the divider between the viewport and the bottom panel to make it taller.

Resetting the layout

If you drag things around and get lost, use **Editor > Editor Layout** (or the **Editor > Editor Layouts** submenu in some versions) and choose **Default** or save your own layout.

2.2 The workspaces

The buttons at the top switch the *main screen*. They change only the central area; the docks stay.

Workspace	What it shows	When you use it
2D	A flat canvas with a grid and ruler	Placing sprites, UI, tiles, markers. Most of this book.
3D	A three-dimensional view with a camera	3D games. Mentioned only for completeness.
Script	A code editor with a list of open scripts on the left and the class reference	Writing GDScript. From Chapter 3.
Game	The running game, shown inside the editor window (Godot 4.4 and later)	Watching and adjusting a playtest without a separate window. Optional in this book.
AssetLib	The online library of add-ons and demo projects	Downloading plugins and examples (needs internet).

On Windows the shortcuts to switch are **Ctrl+F1**, **Ctrl+F2**, **Ctrl+F3**, and so on, in the order of the buttons (2D first, then 3D, Script, and so on). Check **Editor > Editor Settings > Shortcuts** if yours differ. When you select a node, the editor often picks a fitting workspace: choosing a `Node2D` shows it in the 2D view.

> **Note:** The **Game** workspace needs Godot 4.4 or newer. It shows the running game inside the editor window and lets you tweak it live. It is a handy extra, but this book does not depend on it; the Remote scene tree in Section 2.9 works the same way with or without it.

The **Script** workspace has a small built-in help system. Press **F1** to open the class reference, or hold **Ctrl** and click any class name in code to jump to its documentation. Developers consult it constantly; you should too.

2.3 Nodes

Now to the main idea. A **node** is the smallest building block in Godot. Each node:

- has a **name**, which must be unique among its siblings (the other children of the same parent);
- has a **type** (also called a *class*), such as `Sprite2D` or `Label`, which decides what it can do;
- has **properties**, the values you can edit, such as position, color, or text;
- can receive a call to run every frame, so it can **update** itself;
- can have a **script** attached to give it custom behavior;
- can have **child nodes**, and it has exactly one parent (except the root).

A node does one thing. That is deliberate. Compare these:

Node type	What it does
`Node`	Nothing visible; the plain base for everything. Useful as an organizer.
`Sprite2D`	Draws a texture (a picture) on screen.
`Label`	Draws text.
`Button`	A clickable button.
`Timer`	Counts time, then sends a `timeout` signal.
`AudioStreamPlayer`	Plays a sound.
`CollisionShape2D`	Defines an area for physics checks.
`Camera2D`	Decides which part of the world the player sees.

Godot has hundreds of node types. You do not memorize them; you learn the common twenty and look up the rest.

Adding a node

To add a node:

1. Select the node that will be the *parent* in the Scene dock.
2. Click the + button at the top of the Scene dock ("Add Child Node"), or right-click and choose **Add Child Node...**, or press **Ctrl+A**.
3. In the dialog, type part of the type name in the search box (for example `sprite`). The list narrows. Select a type and press **Create**.

New nodes get the type name by default, such as `Sprite2D`. Rename a node by selecting it and pressing **F2**, or double-clicking its name.

Common actions on nodes

Right-click a node in the Scene dock to see the menu. The useful items:

Menu item	What it does
Add Child Node...	Adds a new node below the selected one
Instantiate Child Scene...	Adds a saved scene as a child (Section 2.5)
Rename (F2)	Changes the node's name
Duplicate (Ctrl+D)	Copies the node and its children
Move Child Up / Down	Changes order among siblings (which matters for drawing order)
Reparent	Moves the node under another parent
Change Type...	Swaps the node's type, keeping what it can
Make Scene Root	Makes this node the top of the scene
Save Branch as Scene	Saves this node and its children as a new scene file
Delete Node(s) (Delete key)	Removes the node and its children

You can also drag nodes in the Scene dock to re-parent or reorder them. **Ctrl+Z** undoes mistakes.

The scene tree and names

Nodes are arranged in a **tree**: a hierarchy with one root and branches. A tree is just a way to say "who contains whom":

```
Room            <- root node (parent of Floor, Lamp, Title)
|-- Floor       <- child of Room
|-- Lamp        <- child of Room
|   `-- Shade   <- child of Lamp, grandchild of Room
`-- Title
```

The tree gives every node a **path**, built from names separated by slashes, like a folder path. `Room/Lamp/Shade` is the full address of `Shade`, counting from the root. Scripts use these names (`$Lamp/Shade`) to find nodes (Chapter 3 onward), which is why names matter.

What a parent gives its children

The tree is not only a filing system. A child inherits some behavior from its parent in the tree:

- **Position, rotation, and scale** (for 2D nodes): move the parent and all children move with it, because a child's position is *relative to its parent*.
- **Visibility**: hide the parent and its children disappear too.
- **Lifetime**: delete the parent and the children are deleted. A scene starts "ready" bottom-up: children are ready before parents.
- **Drawing order**: in 2D, nodes lower in the tree draw on top of earlier ones (unless changed with Z Index).

This is why a character scene works as one unit: a root, with a sprite, a collision shape, and a camera as children. Move the root and the whole character moves.

> **Why it works:** Because child positions are relative to the parent, you can build a lamp with its pole at (0, 0) and later put the whole lamp anywhere. Every part follows. You only describe the lamp once.

2.4 The node type hierarchy

Node types are organized into a **class hierarchy**: a family tree of types, in which each type *extends* (inherits from) another and gains its abilities. You may know this from Java: `Dog extends Animal`. Godot works the same way.

The top part of the hierarchy that matters here:

```
Object
`-- Node                       (anything in the scene tree: name, parent, children, _process)
    |-- CanvasItem             (anything drawn in 2D: visible, modulate, z_index)
    |   |-- Node2D             (a 2D thing with position, rotation, scale)
    |   |   |-- Sprite2D
    |   |   |-- Camera2D
    |   |   |-- CollisionShape2D
    |   |   `-- ...
    |   `-- Control            (a user-interface thing: anchors, size, themes)
    |       |-- Label
    |       |-- Button
    |       |-- ColorRect
    |       |-- Container (VBoxContainer, HBoxContainer, ...)
    |       `-- ...
    |-- Node3D                 (a 3D thing: not used in this book)
    |-- Timer
    |-- AudioStreamPlayer
    `-- ...
```

Read it as "is a". A `Sprite2D` *is a* `Node2D`, which *is a* `CanvasItem`, which *is a* `Node`, which *is an* `Object`. Everything a `Node2D` can do, a `Sprite2D` can do too. In the Inspector the properties are grouped by class, with the most basic class at the bottom, so you can see this layering.

The four classes to remember:

Class	What it adds	Examples
Node	A name, a place in the tree, children, a script, update calls. Has no position and draws nothing.	`Timer`, `AudioStreamPlayer`, plain `Node`
CanvasItem	The ability to be drawn in 2D: `visible`, `modulate` (tint), `z_index`. You rarely add this directly; it is a base class.	(base class)
Node2D	A **position**, **rotation**, and **scale** in 2D space, relative to its parent.	`Sprite2D`, `Camera2D`, `Area2D`
Control	Everything needed for user interfaces: a rectangle with a **size**, **anchors**, **margins**, **themes**, and mouse focus.	`Label`, `Button`, `ColorRect`

Both `Node2D` and `Control` extend `CanvasItem`, so both can be drawn. The difference: `Node2D` thinks in *world coordinates* (a point on a map), while `Control` thinks in *rectangles that fit into other rectangles* (like windows and buttons). Use `Node2D`-family nodes for game-world objects, and `Control`-family nodes for menus, text, and heads-up displays.

> **Warning:** When a `Control` is a child of a `Node2D` (or the reverse), the Control's anchors and layout features do not work, because the parent is not a Control. The Control still shows, and you place it by its position and size. You will do exactly this in the lab. Chapter 8 teaches proper UI layout.
>
> **Try it:** In a new scene, add a `Node2D`, then add a `Sprite2D` child. Select the `Sprite2D` and look at the Inspector from top to bottom. You will find sections with these headings, in order: `Sprite2D`, `Node2D` (listing Transform properties), `CanvasItem` (Visibility, Ordering, Texture), and `Node` (Process and Editor Description). This is the inheritance chain, visible.

Choosing a type

When you are unsure which node to use, ask what job it must do:

1. Needs only to hold other nodes, with no visuals? Use `Node` (or `Node2D` if the children need a shared position).
2. Is it part of the game world with a position? Use something in the `Node2D` family.
3. Is it text, a button, a panel, a menu? Use something in the `Control` family.
4. Is it a helper with no look (timer, sound)? Use the specific node, which extends `Node`.

2.5 Scenes: trees you can save and reuse

A **scene** is a tree of nodes saved to disk. The file ends in `.tscn` ("text scene"), and is plain text, so version control tools handle it well. A scene can be tiny (a single button) or huge (a whole level).

A scene has these properties:

- It always has exactly **one root node**. Everything else is below it.
- It can be **saved**, **opened**, and **edited** on its own, in its own editor tab.
- It can be **run** by itself (F6) to test it.
- It can be **instanced** inside other scenes.

Scenes as the unit of reuse

In many tools, "scene" means "level". In Godot it means "any saved tree". A level is a scene. A character is a scene. A bullet is a scene. A button with a special look is a scene. The idea is the same at every scale.

Think of a scene as a *blueprint* (in programming terms, a class) and an instance as the *thing built from it* (an object). The blueprint is one file on disk. You can place many instances of it.

Instancing a scene

To **instance** a scene is to place a copy of it into another scene as a single node. In the editor you can do it by:

- **Right-click a node > Instantiate Child Scene...**, then pick a `.tscn` file; or
- clicking the chain-link button at the top of the Scene dock (**Instantiate a scene file as a Node**) or pressing **Ctrl+Shift+A**; or
- dragging a `.tscn` file from the FileSystem dock into the Scene dock or the viewport.

An instanced scene appears in the tree as one node with a small **clapboard icon** (a movie slate) on the right of the row. You do not see its internal nodes unless you expand it, and its inner nodes are locked from accidental changes by default.

```
HelloRoom                  <- root of the room scene
|-- Floor
|-- Lamp   [clapboard]     <- instance of lamp.tscn: looks like one node
`-- Lamp2  [clapboard]     <- a second instance of the same lamp.tscn
```

What you can do with an instance:

Action	How
Move or rotate it	Change its own **Transform** (position, rotation, scale)
Rename it	F2 as usual; the name is local to this room
Override a root property	Select it; changes in the Inspector apply to this instance only
See inside	Click the small arrow to expand it
Edit inside without leaving	Right-click > **Editable Children** (use sparingly)
Open the original scene	Click the clapboard icon, or right-click > **Open in Editor**

Why instancing beats copying

Suppose you want two lamps in a room. You could copy and paste the lamp's nodes twice. But if you later change the lamp's color, you must change both copies. With three hundred torches in a dungeon you would be lost.

With instancing, both lamps are *references* to one blueprint. Change `lamp.tscn` once, save, and every instance in every room updates. That is the same reason you write a function instead of pasting the same code in ten places.

> **Why it works:** The room scene file does not contain the lamp's nodes. It stores "an instance of `res://lamp.tscn` named `Lamp`, at position (250, 260)". When the room loads, Godot reads the lamp file and builds the nodes.

Scene trees at runtime

When the game runs, Godot builds one big **scene tree** out of all the active scenes. The very top of that tree is a hidden `Window` node called `root`, and under it sits your main scene. Instanced scenes are expanded into their nodes. The result is one tree of nodes, whatever files they came from. This is the live tree you can view as the Remote tree (Section 2.9).

2.6 Composition, inheritance, and "everything is a node"

How should you organize a complex game object? Two techniques exist.

Inheritance

Inheritance means making a new type by extending an existing one. In classic object-oriented design you might write:

```
Enemy
|-- FlyingEnemy
|   `-- FlyingShootingEnemy
`-- WalkingEnemy
```

Each level adds or changes behavior. It works for simple cases, but trees of types grow hard to change. What if you want an enemy that flies, shoots, *and* walks on the ground? You have to either rearrange the tree or copy code.

Composition

Composition means building an object by assembling small parts, each with one job. A Godot enemy might be:

```
Enemy (CharacterBody2D)
|-- Sprite2D        (looks)
|-- CollisionShape2D (body)
|-- HealthPart      (a small scene that tracks health)
|-- ShooterPart     (a small scene that fires bullets)
`-- Timer           (attack rate)
```

Need a flying enemy that shoots? Take the same parts and leave out walking. Need one without shooting? Leave out `ShooterPart`. You change the object by changing its parts, not by re-doing a family tree.

What Godot recommends

Godot uses **both**, but leans on composition:

- **Node types** use inheritance (`Sprite2D` extends `Node2D`) so that common abilities are shared.
- **Your own objects** are best built by composition: small scenes combined into bigger scenes.
- **Scripts** can also inherit (`extends Node2D`), and Chapter 5 shows how. Use it where it helps, but do not build deep chains.

A good rule: *prefer a node with a small job, combined with others, over one huge node doing everything*. If a scene grows hard to understand, split a branch into its own scene with **Save Branch as Scene**.

Everything is a node

Godot's design says: a menu is nodes, a level is nodes, a character is nodes, a timer is a node, a sound player is a node. There is no special "menu system" and "level system" with different rules. This uniformity has practical effects:

- You learn one set of actions (add, move, rename, instance, delete) and they work everywhere.
- A scene can contain any mix: a level scene can hold UI nodes, sound nodes, and character scenes.
- Scenes can be nested to any depth: a game scene holds a level scene, which holds a room scene, which holds a lamp scene.
- Even your whole game is one big scene tree.

> **Note:** Not everything in Godot is a node. Data such as pictures, fonts, and sounds are **Resources**, which are not placed in the tree but are used *by* nodes. For example, a `Sprite2D` node uses a `Texture2D` resource. You will meet resources again in Chapter 10.

Choosing the root and a naming scheme

Choose the root node type by what the scene *is*:

Scene is…	Good root
A world object with a position (lamp, door, bullet)	`Node2D` (or a physics body)
A room or level	`Node2D`
A menu or HUD	`Control`
A pure organizer	`Node`

Naming conventions. Godot's style guide, and this book, use two simple rules:

Thing	Style	Examples
Files and folders (scenes, scripts, images, sounds)	`snake_case` (lowercase words with underscores)	`hello_room.tscn`, `lamp.tscn`, `player_walk.png`
Node names	`PascalCase` (each word capitalized, no spaces)	`HelloRoom`, `Lamp`, `FloorTile`

The reason for lowercase file names: Windows ignores case in file names but Linux and macOS do not. A project that works on Windows with `Lamp.tscn` and `lamp.tscn` mixed up can break after export to another system. Using one style avoids the whole problem. Also avoid spaces and special characters in names.

The root node of a scene is usually named in PascalCase after the file: `lamp.tscn` has a root called `Lamp`. When you save a new scene, Godot proposes this pairing for you.

2.7 The Inspector and properties

Select any node and look at the **Inspector**. It lists the node's **properties**, grouped by the class that defines them. Each is a named value stored with the node. The Inspector lets you view and edit them without writing code.

Reading the Inspector

Part	Meaning
Header (top)	The node's name and a button to open its documentation
Search properties box	Filters the list; type `pos` to find position
Section titles	Click to collapse or expand (for example Transform, Visibility, Theme Overrides)
Property rows	Name on the left, editor on the right
Revert arrow	A circular arrow appears when a value differs from the default; click it to restore the default

Property editors depend on the type of value:

Type	Editor
Number	A field; drag the label left or right to change it quickly; type a value
Text	A text box (multi-line for some)
Vector2 (x and y together, such as position or size)	Two number fields
Color	A color swatch; click to open a color picker
Boolean	A checkbox
Enum (a list of choices)	A drop-down
Resource (a texture, font, sound)	A slot; drag a file from the FileSystem, or click `<empty>` to choose

Important properties to know now

- **Node2D > Transform**: `Position` (x, y) in pixels relative to the parent; `Rotation` (degrees in the editor); `Scale`.
- **CanvasItem > Visibility**: `Visible` and `Modulate` (a color that tints the node and its children).
- **CanvasItem > Ordering**: `Z Index`, which pushes a node in front of or behind others.
- **Control > Layout**: `Position` and `Size`, plus anchors.
- **Label > Text**; **ColorRect > Color**; **Sprite2D > Texture**.

Coordinates in 2D

In Godot 2D, the origin (0, 0) is the **top-left** corner. **x increases to the right** and **y increases downward**. This surprises people who expect y to go up, as in school graphs. Chapter 6 treats this fully; remember it now so positions in this chapter make sense.

```
(0,0) ------------> x
  |
  |
  v
  y
```

The unsaved dot and editing safely

When you change anything, a small `(*)` or dot appears on the scene's tab to show unsaved changes. Press **Ctrl+S** often. If the editor crashes, unsaved changes are lost. The editor also warns you when you close with unsaved work.

2.8 Saving and running

Saving scenes

Command	Menu path	Shortcut
Save the current scene	**Scene > Save Scene**	Ctrl+S
Save under a different name	**Scene > Save Scene As...**	Ctrl+Shift+S
Save every open scene	**Scene > Save All Scenes**	Ctrl+Shift+Alt+S
Open a scene file	**Scene > Open Scene...**	Ctrl+O

Command	Menu path	Shortcut
Close the current scene	**Scene > Close Scene**	Ctrl+W
Create a new scene	**Scene > New Scene**	Ctrl+N

When you save a scene the first time, Godot asks for the file name. Use `snake_case.tscn` and keep scenes in sensible folders, such as `res://rooms/` and `res://objects/`.

Scenes you have open appear as **tabs above the viewport**. You can have several open and switch between them, which is how you edit a lamp while looking at the room that uses it.

Running the game

You can run the game in three ways. All are in the top-right corner and each has a key:

Button	Key	What it runs
Run Project (play icon)	**F5**	The *main scene* of the project
Run Current Scene (clapboard-play icon)	**F6**	Only the scene you have open
Stop (square icon)	**F8**	Closes the running game
Pause	**F7**	Pauses the running game
Run Specific Scene	(button; shortcut in Editor Settings)	Lets you choose any scene to run

The buttons and keys are the stable way to run; menu entries for them differ a little between 4.x releases.

F6 is your testing friend. If you are building a lamp, you can run just the lamp scene without launching the whole game. Make a habit of testing each scene on its own with F6.

The main scene

The first time you press F5, Godot asks which scene to start. That choice is the **main scene**. You can change it any time at **Project > Project Settings > Application > Run > Main Scene**, or by right-clicking a scene in the FileSystem dock and choosing **Set As Main Scene**. The main scene is where the player's game begins. In later chapters it may be a menu; in the adventure it is the first room or the start screen.

What happens when you run

When you press F5 or F6:

1. Godot saves your open scenes (if you have the default setting that asks you to).
2. It starts a *second program*, the game, in its own window. The editor keeps running.
3. Messages and errors appear in the **Output** and **Debugger** panels.
4. When you press F8 (or close the game window), the game stops. Changes you make to a running game do **not** change your saved scenes.

> **Warning:** Edits made to the Inspector while the game runs, through the Remote tree below, are temporary. Make permanent changes only when the game is stopped.

2.9 Watching the running game: the Remote scene tree

While the game runs, look at the Scene dock. Two tabs appear at its top: **Local** and **Remote**.

- **Local** is the scene tree you are editing in the editor.
- **Remote** is the *live* tree of the running game, which you can inspect.

In the Remote tab you see the full runtime tree: the hidden `root` window, then your main scene with all instanced scenes expanded into their nodes. Click any node and the Inspector shows its **current** property values, which can differ from those saved on disk. If a script moved a node, you see the new position. Change a value in the Inspector and the running game changes right away.

Uses of the Remote tree:

- Find out *why something is invisible* (is `visible` false? is the position off screen?).
- See which nodes exist at the moment, including ones created by scripts.
- Try values quickly (a faster speed, a different color) before editing the scene for real.
- Confirm that an instanced scene's inner nodes were built.

Changes in the Remote tree vanish when the game stops. If you like a value, write it down and apply it in the Local tab.

> **Try it:** Run your Chapter 1 project. In the Scene dock, click the **Remote** tab, select the Label, and in the Inspector change its Text. Watch the running window change. Press F8 and check the Local tree: the text is unchanged.

2.10 A first, tiny script

Full programming starts in Chapter 3, but you should see one script now to know where code goes. A **script** is a text file, with the extension `.gd`, attached to a node. It adds behavior.

To attach a script:

1. Select a node in the Scene dock.
2. Click the **Attach Script** button (a scroll with a plus icon) at the top of the Scene dock, or right-click the node and choose **Attach Script...**.
3. A dialog opens. Leave the language as **GDScript** and the template as the default. The **Path** is filled in with a snake_case file name matching the scene or node. Click **Create**.

The Script workspace opens with a template. For the lab, replace everything with the short script below.

File `res://objects/lamp.gd` (in the lab), attached to the root `Lamp` (a `Node2D`) of the lamp scene:

```
extends Node2D

func _ready() -> void:
    print(name, " is ready at ", position)
```

What each part does, briefly:

- `extends Node2D` says this script belongs to a `Node2D` node (the type of the node you attached it to).

- `func _ready() -> void:` defines a function that Godot calls once, when the node and all its children have entered the scene tree.
- `print(...)` writes text to the **Output** panel. `name` and `position` are properties of the node itself, the same ones you see in the Inspector.
- The indentation with a tab is how GDScript knows a line belongs to the function. Chapter 3 explains.

When the script is attached, a small scroll icon appears beside the node in the Scene dock. Click it to jump to the script. To remove or change the script, use the same icon area or right-click > **Clear Script** / **Detach Script**.

Lab 2: Build the "Hello Room" scene

Goal: Build a lamp as its own scene, place two instances of it in a room scene, give the lamp a small script, and run the result. You will also prove that editing the lamp once changes both copies.

Time: 45 to 75 minutes.

What you need: Chapter 1 finished (or any Godot 4 project), set to the default window size of 1152 by 648.

Part A: Prepare the project

1. Open **Project Manager** and open `hello_godot`. (Or create a new project called `hello_room` with the Compatibility renderer.)
2. In the **FileSystem** dock, right-click `res://` and choose **New Folder...**. Name it `objects`. Create another folder named `rooms`.

Part B: Build the lamp scene

3. Choose **Scene > New Scene**. Click **2D Scene**. A `Node2D` appears. Rename it to `Lamp` (select it, press F2, type `Lamp`, press Enter).
4. Add a child: select `Lamp`, press **Ctrl+A**, search `ColorRect`, click **Create**. Rename it `Pole`.
5. With `Pole` selected, in the Inspector open **Layout > Transform**. Set **Position** to `(-5, 0)` and **Size** to `(10, 160)`. Then set **Color** (under the ColorRect section) to a mid grey.
6. Select `Lamp` again, add another `ColorRect` child and rename it `Shade`. Set its **Position** to `(-40, -40)`, **Size** to `(80, 50)`, and **Color** to a warm yellow.
7. You now have this tree:

```
Lamp            (Node2D)
|-- Pole        (ColorRect)  position (-5, 0),   size (10, 160), grey
`-- Shade       (ColorRect)  position (-40, -40), size (80, 50),  yellow
```

The lamp root sits at the **top** of the pole; the pole stands down 160 pixels from the root. Zoom in to see it. 8. Save the scene: **Scene > Save Scene** (Ctrl+S). Name it `lamp.tscn` in `res://objects/`. 9. Press **F6** to run only the lamp. A window shows a yellow shade on a grey pole in the top-left area (partly cut off, since the origin is the corner). That is fine; stop with **F8**.

Part C: Add the short script

10. Select the `Lamp` root node. Click **Attach Script**. Leave GDScript and the default template. Set the path to `res://objects/lamp.gd` and click **Create**.

11. In the Script workspace, delete the template text and type exactly:

```
extends Node2D

func _ready() -> void:
    print(name, " is ready at ", position)
```

12. Press **Ctrl+S** to save the script. Switch back to the **2D** workspace (top of the window).

Part D: Build the room scene

13. Choose **Scene > New Scene**, then **2D Scene**. Rename the root `HelloRoom`.
14. Add a `ColorRect` child, rename it `Wall`. Under Layout set **Position** `(0, 0)`, **Size** `(1152, 420)`. Choose a soft blue-grey color.
15. Add another `ColorRect` child to `HelloRoom`, rename it `Floor`. Set **Position** `(0, 420)`, **Size** `(1152, 228)`, and a brown color.
16. Add a `Label` child to `HelloRoom`, rename it `Title`. Set **Text** to `Hello Room`. Open **Theme Overrides > Font Sizes** and set **Font Size** to `40`. Set its **Position** to `(440, 40)`.
17. Save the room as `res://rooms/hello_room.tscn`.

The tree is:

```
HelloRoom       (Node2D)
|-- Wall        (ColorRect)
|-- Floor       (ColorRect)
`-- Title       (Label)
```

> **Note:** The order matters. `Wall` is above `Floor` and `Title` in the tree, so they draw on top of it. If the Title is invisible, move it to the bottom of the list by dragging, or right-click > **Move Child Down**.

Part E: Instance the lamp twice

18. Select `HelloRoom`. Click the chain-link button at the top of the Scene dock (**Instantiate a scene file as a Node**). Choose `res://objects/lamp.tscn` and click **Open**.
19. A node named `Lamp` with a clapboard icon appears. Select it and set its **Position** (Node2D > Transform) to `(250, 260)`. Its pole now ends at y = 420, exactly where the floor starts.
20. Instance the lamp a second time the same way. Godot names it `Lamp2` automatically. Set its position to `(850, 260)`.
21. Rename `Lamp2` to `LampRight` and `Lamp` to `LampLeft` (F2). Names are only labels; each is still an instance of the same file.

The final tree:

```
HelloRoom       (Node2D)
|-- Wall        (ColorRect)
|-- Floor       (ColorRect)
|-- Title       (Label)
|-- LampLeft    (instance of lamp.tscn)
`-- LampRight   (instance of lamp.tscn)
```

22. Save with **Ctrl+S**.

Part F: Run it

23. Press **F6** to run the current scene. A 1152 by 648 window shows a wall, a floor, the text "Hello Room", and two lamps standing on the floor.
24. Look at the **Output** panel in the editor. You should see two lines like these:

```
LampLeft is ready at (250, 260)
LampRight is ready at (850, 260)
```

Both lines come from the *same script* in the *same scene*, but each instance reports its own name and position. Stop the game with **F8**.

25. To make this the project's main scene, right-click `hello_room.tscn` in the FileSystem dock and choose **Set As Main Scene**. Press **F5**. The same window appears.

Part G: Watch it live

26. With the game running, click the **Remote** tab at the top of the Scene dock. Expand `HelloRoom`. Click `LampLeft`, expand it, and select `Shade`. In the Inspector change **Color** to red. Only the left lamp's shade changes, in the running game.
27. Select `LampRight`'s `Shade` and set its **Color** to green. The right one changes. Each instance has its own nodes at runtime.
28. Press **F8**. Check the **Local** tab: nothing was saved.

Part H: One change, two lamps

29. Double-click `lamp.tscn` in the FileSystem dock (or click the clapboard icon beside `LampLeft`) to open the lamp scene.
30. Select `Shade` and change its **Color** to orange and its **Size** to `(100, 50)`, with **Position** `(-50, -40)` to keep it centered. Save with **Ctrl+S**.
31. Switch back to the `hello_room.tscn` tab and press **F6**. Both lamps now have the wider orange shade. You edited one file; both instances changed.
32. Optional: Select only `LampRight` in the room, set its **Rotation** (in the Transform group) to `15` degrees, and run again. Only that instance is tilted, while the blueprint is untouched.

Expected result: Running `hello_room.tscn` (F5 or F6) opens a window that shows a wall, a floor, a title, and two instances of the lamp scene. The Output panel prints one "is ready" line per lamp with the right name and position. Changing `lamp.tscn` and saving updates both lamps. The Remote tab lets you change a lamp live without saving anything. Your files are `res://objects/lamp.tscn`, `res://objects/lamp.gd`, and `res://rooms/hello_room.tscn`.

Chapter summary

- The editor has a menu and run bar on top; the Scene, Import, and FileSystem docks on the left; the viewport in the middle; the Inspector and Node dock on the right; and Output, Debugger, and other panels at the bottom.
- Workspaces switch the main view: 2D, 3D, Script, Game (Godot 4.4 and later), and AssetLib.
- A node is a small building block with a name, a type, properties, and children. Nodes form a tree, and a child's position, visibility, and lifetime follow its parent.

- Node types form a family tree: Node, then CanvasItem, then either Node2D (world things with position) or Control (interface things with size and anchors).
- A scene is a saved tree (`.tscn`) with one root. Instancing places a scene inside another; changing the original updates every instance.
- Prefer composition (small parts combined) over deep inheritance. In Godot nearly everything, including menus and levels, is built from nodes.
- Use `snake_case` for files and folders and `PascalCase` for node names.
- F5 runs the project's main scene, F6 runs the open scene, and F8 stops. The Remote scene tree lets you inspect and tweak the running game without saving the changes.

Review questions

1. **Multiple choice.** Which dock shows the files of your project? a) Scene b) Inspector c) FileSystem d) Node

2. **True or false.** A node can have more than one parent.

3. **Short answer.** What is the difference between a node and a scene?

4. **Multiple choice.** Which class is the base for things that have a 2D position, rotation, and scale? a) Control b) Node2D c) Timer d) Resource

5. **True or false.** If you move a parent `Node2D` to the right, its child `Node2D` nodes also move to the right.

6. **Multiple choice.** Which key runs only the scene you currently have open? a) F5 b) F6 c) F7 d) F8

7. **Short answer.** You have placed the same lamp scene five times in a room and now want to change its color. Explain why instancing makes this easy.

8. **Multiple choice.** Following Godot naming conventions, which pair is correct for a scene file and its root node? a) `Hero.tscn` and `hero` b) `hero_walk.tscn` and `HeroWalk` c) `Hero Walk.tscn` and `hero walk` d) `HERO_WALK.TSCN` and `HERO_WALK`

9. **True or false.** Changes you make to a node in the Remote scene tree while the game runs are saved into your scene file.

10. **Short answer.** Give one reason to prefer composition over inheritance when building a game character, and describe a small example.

Exercises

1. Practice: Name the parts. Take a screenshot of your Godot editor and label by hand (or in a drawing tool) these eight areas: the menu bar, workspace buttons, run buttons, Scene dock, FileSystem dock, viewport, Inspector, and bottom panel. Next to each, write one sentence on what you used it for in Lab 2.

2. Practice: Add a third lamp. In `hello_room.tscn`, add a third instance of the lamp scene in the middle of the room, at position `(550, 260)`, and name it `LampCenter`. Run the scene and read the Output. Then use the Inspector to make only this lamp twice as big using **Scale** `(2, 2)`. Explain where the lamp's bottom ends up and why (hint: scale is applied around the root's position).

3. Apply: A lamp that remembers a name. In `lamp.gd`, add these two lines inside `_ready()`, below the existing `print`: `for child in get_children():` and, indented one more tab under it, `print(child.name)`. Run and check the Output. (This is a preview of loops. You will learn `for` loops properly in Chapter 4; for now just copy the shape of the code.) Describe what happens to the output if you add a third child node to the lamp scene.

4. Apply: Build a second reusable object. Create `res://objects/picture.tscn` with a `Node2D` root named `Picture`, a `ColorRect` named `Frame` (dark brown, `160 x 100`), and a smaller `ColorRect` named `Canvas` (a light color, `140 x 80`) placed so it sits inside the frame with a 10-pixel border. Instance it twice on the wall of `HelloRoom` at different positions. Change the frame color in the `picture.tscn` file and confirm both change.

5. Challenge: A room made of rooms. Make a new scene `res://rooms/house.tscn` with a `Node2D` root named `House`. Instance `hello_room.tscn` inside it, and then instance it a second time and shift the second copy to x = 1152 so you have two rooms side by side. Run it with F6. Explain what you see in the **Remote** tree (how many `LampLeft` nodes exist, and under which parents?), and why scenes inside scenes inside scenes is the same idea at three levels. Finally, find out what the Output panel prints when two `HelloRoom` instances both contain lamps with the same node name.

Common mistakes

- **Symptom:** You press F5 and the wrong scene opens, or Godot asks again which scene to run. **Cause:** The main scene is not set, or is set to an old scene. **Fix:** Right-click the right scene in the FileSystem dock and choose **Set As Main Scene**, or use **Project > Project Settings > Application > Run > Main Scene**.
- **Symptom:** You added a node but cannot see it in the game. **Cause:** Often it is drawn behind another node (check the order in the Scene dock), is outside the visible window area, has `Visible` unchecked, or is a `Control` with a size of zero. **Fix:** Check the Remote tree while running; look at `position`, `size`, `visible`, and its place in the tree order.
- **Symptom:** A red or yellow icon appears beside a node, or a warning says a node is missing a shape or texture. **Cause:** The node needs a required child or resource (for example, a `CollisionShape2D` needs a shape). **Fix:** Hover the icon to read the message and supply what it asks for.
- **Symptom:** After renaming or moving a file in Windows Explorer, scenes show "Missing dependency" errors. **Cause:** Godot did not know about the move, so references broke. **Fix:** Move and rename files inside Godot's **FileSystem** dock, which updates references. Undo the Explorer change if needed.
- **Symptom:** Your edits to an instanced lamp are lost, or only appear on one copy. **Cause:** You changed an instance's properties in the room (which apply to that instance only) instead of editing the original scene. **Fix:** Open the original scene (click the clapboard icon) and change it there for changes that should apply to all copies.
- **Symptom:** Godot will not attach a script, or reports an error such as "Script inherits from native type 'Sprite2D', so it can't be assigned to an object of type: 'Node2D'". **Cause:** The script's first line (`extends ...`) names a type that does not fit the node it is attached to, for example `extends Sprite2D` on a `Node2D`. **Fix:** Make `extends` match the node's type (or a type it inherits from).

Further reading

- Nodes and scenes: https://docs.godotengine.org/en/stable/getting_started/step_by_step/nodes_and_scenes.html
- Instancing: https://docs.godotengine.org/en/stable/getting_started/step_by_step/instancing.html

- Scene organization: https://docs.godotengine.org/en/stable/tutorials/best_practices/scene_organization.html
- First look at the editor: https://docs.godotengine.org/en/stable/getting_started/introduction/first_look_at_the_editor.html
- Godot's design philosophy: https://docs.godotengine.org/en/stable/getting_started/introduction/godot_design_philosophy.html
- GDScript style guide (naming): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_styleguide.html
- The Inspector dock: https://docs.godotengine.org/en/stable/tutorials/editor/inspector_dock.html

Chapter 3: GDScript I: Variables, Types, and the Life of a Script

Learning objectives

By the end of this chapter you will be able to:

- Create a script file, attach it to a node, and explain what `extends` does.
- Write comments, indent code with tabs, and send messages to the Output panel with `print`, `print_rich`, and `push_warning`.
- Declare variables and constants with `var` and `const`, and choose the right built-in type for a value.
- Use static typing and type inference (`:=`) to catch mistakes before the game runs, and predict the result of arithmetic, comparison, and logical expressions, including the integer division trap.
- Format text with `%`, `format()`, and `str()`, and group related names with `enum`.
- Explain when Godot calls `_ready`, `_process`, `_physics_process`, and `_input`, and write a script that uses all four.

Key terms

- **Script**: a text file of GDScript code that adds behavior to a node.
- **GDScript**: Godot's own programming language, designed to look like Python and to work closely with nodes.
- **Variable**: a named box that holds a value; declared with `var`.
- **Constant**: a named value that can never change; declared with `const`.
- **Type**: the kind of data a value is, such as whole number, decimal number, or text.
- **Static typing**: writing the type of a variable or function in the code so Godot can check it before the game runs.
- **Type inference**: letting Godot work out the type from the value, written `:=`.
- **Operator**: a symbol such as `+`, `==`, or `and` that combines values.
- **Enum**: a named list of constants, usually used for states or choices.
- **Lifecycle function**: a function with a special name, such as `_ready`, that Godot calls for you at a fixed moment.
- **Delta**: the number of seconds that passed since the previous frame.
- **Output panel**: the editor panel at the bottom where `print` messages appear.

3.1 Scripts, nodes, and the script editor

In Chapter 2 you built scenes out of nodes. Each node type already knows how to do something: a `Label` shows text, a `Sprite2D` shows a picture. A **script** lets you teach a node something new. The script is a plain text file, saved inside your project folder, written in **GDScript**.

> **Note:** GDScript is not Python, but if you have seen Python it will feel familiar: no semicolons, no curly braces, and blocks of code are shown by indentation. If you know Java or C, the biggest changes are that blocks are indentation instead of `{ }` and that line endings end a statement.

The one-script-one-node idea

A script is attached to exactly one node at a time. When the scene starts, the script becomes part of that node. Inside the script you can read and change the node's own properties (for example `text` on a Label) and call its functions.

Setting up the playground scene

For the examples in this chapter you need a small test scene. Build it once and reuse it for every example.

Scene tree:

```
Playground   (Control)
└── Output   (Label)
```

Steps:

1. Create a new project (or use your Chapter 2 project). Choose **Scene > New Scene**, then pick **User Interface**. This creates a `Control` root node. Rename it `Playground`.
2. Right-click `Playground`, choose **Add Child Node**, and add a `Label`. Rename it `Output`.
3. Select `Playground`. In the Inspector, open **Layout > Anchors Preset** and choose **Full Rect**. Do the same for `Output`. This makes both fill the window.
4. Save the scene with Ctrl+S as `res://playground.tscn`.
5. Select `Playground`, then click the **Attach Script** button (the scroll icon above the Scene dock). In the dialog leave the language as GDScript, make sure the template is **Node: Default**, set the path to `res://playground.gd`, and click **Create**.
6. The Script workspace opens. Replace everything with the code below.

File: `res://playground.gd`, attached to `Playground` (Control).

```gdscript
extends Control

@onready var out: Label = $Output

# Shows text in the Label and also in the Output panel.
func say(text: Variant) -> void:
    out.text += str(text) + "\n"
    print(text)

func _ready() -> void:
    say("Hello, Godot!")
```

Press **F6** (Run Current Scene). A window opens with the words `Hello, Godot!`, and the same text appears in the editor's **Output** panel.

What the output looks like:

```
Hello, Godot!
```

> **Note:** The line `@onready var out: Label = $Output` finds the Label node and stores it in a variable. You will learn exactly how it works in Chapter 5. For now, treat it as a recipe: it gives the script a handle called `out` on the `Output` label. The `say` function is our own small helper that adds a line to the label and to the Output panel.

For most examples in this chapter, the text will say "fragment: put this inside `_ready()`". That means: delete the line `say("Hello, Godot!")` and paste the fragment in its place, indented with one tab.

What `extends` means

The first line of every script says which kind of node it builds on:

```
extends Control
```

This means "my script is a Control, plus the extra things I write below." Because of `extends Control`, the script automatically has every property and function a Control has (`size`, `visible`, `modulate`, and many more) without importing anything. The `extends` word must match the node you attach the script to, or a more general parent type. A script that says `extends Node` can be attached to a Label (because every Label is also a Node), but it then cannot use Label features such as `text` directly.

Indentation: use tabs

GDScript uses indentation to show which lines belong together. After a line ending in a colon `:`, the next lines are indented one level deeper.

```
extends Node

func _ready() -> void:
    print("This line is inside the function.")
    print("So is this one.")

func _exit_tree() -> void:
    print("This line belongs to a different function.")
```

The two `print` lines under `_ready` are indented one tab, so they belong to that function. The next `func` starts again at the left edge, so it is a new function. Godot's editor inserts a **tab** when you press the Tab key, and every code block in this book uses tabs. Do not mix tabs and spaces in one file. If you do, Godot reports an error such as "Used space character for indentation instead of tab as used before in the file." To fix a whole file, use **Edit > Indentation > Convert Indent to Tabs** in the script editor.

Comments

A comment is a note for humans; Godot ignores it. A comment starts with `#` and runs to the end of the line.

```
# This whole line is a comment.
var lives: int = 3  # A comment can follow code on the same line.
```

Comments that start with two hashes, `##`, are **documentation comments**. The editor shows them as help text when someone hovers over the thing they describe.

```gdscript
## How many lives the player starts with.
var starting_lives: int = 3
```

GDScript has no block comment. To comment out several lines, select them and press **Ctrl+K** (the same shortcut turns the comment off again). Use comments for the "why", not for the obvious "what".

> **Try it:** In the playground script, put a `#` in front of the line `say("Hello, Godot!")` and run the scene. The label is empty, because the line is now a comment. Press Ctrl+K on the line to remove the `#` and run again.

Naming rules

GDScript style follows common conventions. Godot does not force them, but all the official examples (and this book) use them.

Thing	Style	Example
Variables and functions	snake_case	`player_speed`, `take_damage()`
Constants and enum members	CONSTANT_CASE	`MAX_HEALTH`, `Rank.VETERAN`
Class names and node names	PascalCase	`Playground`, `HealthBar`
Private-by-convention names	leading underscore	`_cooldown`
Names of built-in lifecycle functions	leading underscore	`_ready`

Names cannot start with a digit, cannot contain spaces, and cannot be a reserved word such as `var`, `if`, `func`, or `class`.

3.2 Talking to the Output panel

When a program does not behave, the first question is "what is the value of this variable right now?" The cheapest answer is to print it.

`print`

```gdscript
func _ready() -> void:
    print("Score: ", 120)
    print("Hello", " ", "world")
    prints("Score:", 120)
    printt("a", "b", "c")
```

- `print` joins all its arguments with nothing in between.
- `prints` puts a **space** between arguments.
- `printt` puts a **tab** between arguments.

What the output looks like (the last line has two tab characters):

```
Score: 120
Hello world
Score: 120
a   b   c
```

Because `print` adds no spaces on its own, `print("Score:", 120)` would show `Score:120`. If you want spaces, add them yourself or use `prints`.

`print` accepts any type of value. It turns each into text automatically, so you can print numbers, vectors, arrays, and more.

`print_rich`

The Output panel can show a little formatting if you use **BBCode** tags in square brackets.

```
func _ready() -> void:
    print_rich("[b]Bold[/b], [i]italic[/i], and [color=red]red[/color] text.")
```

In the Output panel you will see the words styled. Use it to make important messages stand out while debugging. The tags only work in the editor's Output panel, not in the Label of your game window. (A `RichTextLabel` node can show BBCode in a game; you will meet it later.)

`push_warning` and `push_error`

Sometimes you want to report a problem without crashing.

```
func _ready() -> void:
    var lives: int = 0
    if lives == 0:
        push_warning("The player has no lives left.")
    push_error("This is how an error message looks.")
```

`push_warning` prints a yellow message and `push_error` a red one. Both also appear in the **Debugger** panel under the **Errors** tab, with the file and line number. Neither one stops the game. They are the right tool for "this should not happen, but I can keep going".

> **Warning:** A message printed with `print` is easy to lose in a long log. When a message means "something is wrong", use `push_warning` or `push_error` so it shows in color and in the Errors tab.

There is also `printerr`, which prints text to the error stream. For now, `print`, `push_warning`, and `push_error` are enough.

Two quick `if` previews

Several examples need a decision. Chapter 4 explains `if` fully; here is all you need now:

```
if lives == 0:
    print("Game over")
```

The line ends with a colon, and the indented line below runs only when the condition is true.

> **Try it:** Change `lives` to `2` in the warning example and run again. The warning disappears, but the error line still prints. Why? Because the `push_error` line is not inside the `if` block.

3.3 Variables, constants, and built-in types

Declaring a variable

A **variable** stores a value under a name. Declare it with `var`.

```
var player_name: String = "Mira"
var lives: int = 3
```

The part after the colon is the **type** (more on that in section 3.4). After declaring, you can read the variable by writing its name and change it with `=`.

```
func _ready() -> void:
    var lives: int = 3
    say("Lives: " + str(lives))
    lives = lives - 1
    say("Lives: " + str(lives))
```

What the output looks like:

```
Lives: 3
Lives: 2
```

The call `str(lives)` turns the number into text so it can be joined to a string with `+`. Without it, GDScript reports an error because you cannot add a number to a string.

Where a variable lives

- A variable declared **inside a function** is local: only that function can see it, and it disappears when the function ends.
- A variable declared **at the top of the file**, outside any function, is a **member variable**. Every function in the script can use it, and it lasts as long as the node exists.

```
extends Control

@onready var out: Label = $Output

var score: int = 0  # member variable: shared by the whole script

func add_points(amount: int) -> void:
    var bonus: int = 5  # local variable: exists only during this call
    score += amount + bonus

func _ready() -> void:
    add_points(10)
    out.text = "Score: " + str(score)
```

Running this shows `Score: 15`.

Constants

A **constant** is a value that must never change while the game runs. Declare it with `const`, and write its name in CONSTANT_CASE.

```
const MAX_LIVES: int = 5
const GAME_TITLE: String = "Weird Hotel"
```

Trying to assign to a constant (`MAX_LIVES = 6`) is an error: "Cannot assign a new value to a constant." Constants must be known before the game starts, so the value has to be a literal or a simple expression of other constants, not the result of calling an ordinary function.

Use a constant whenever a number has a meaning. `if lives > MAX_LIVES` is clearer than `if lives > 5`, and when you change your mind you edit one line.

The built-in types

Godot gives you a family of basic types. You will use all of these constantly.

Type	Holds	Example literal
`int`	whole numbers, positive or negative	`42`, `-7`, `1_000_000`
`float`	decimal numbers	`3.14`, `0.5`, `1e3`
`bool`	`true` or `false`	`true`
`String`	text	`"Hello"`
`Vector2`	two floats `x` and `y` (a 2D point or direction)	`Vector2(100, 50)`
`Vector3`	three floats `x`, `y`, `z`	`Vector3(1.5, 2.5, 3.5)`
`Color`	red, green, blue, alpha, each from 0 to 1	`Color(1, 0.5, 0)`, `Color.RED`
`NodePath`	a path to a node in the scene tree	`^"Output"`
`null`	"no value"	`null`

Underscores inside number literals are ignored by Godot, so `1_000_000` is one million and easy to read.

Let us meet each type in turn. Put this in `_ready()`.

Fragment: put this inside `_ready()`.

```
    var count: int = 42
    var ratio: float = 0.75
    var alive: bool = true
    var title: String = "Weird Hotel"
    var pos: Vector2 = Vector2(100.5, 50.5)
    var depth: Vector3 = Vector3(1.5, 2.5, 3.5)
    var tint: Color = Color(1, 0, 0)
    var path: NodePath = ^"Output"
    var nothing = null

    say(count)
    say(ratio)
    say(alive)
    say(title)
    say(pos)
    say(depth)
    say(tint.to_html(false))
    say(path)
    say(nothing)
```

What the output looks like:

```
42
0.75
true
Weird Hotel
(100.5, 50.5)
(1.5, 2.5, 3.5)
ff0000
Output
<null>
```

> **Note:** In Godot 4 a whole-number float such as `5.0` prints as `5.0`, but a `Vector2` prints its parts without the `.0` (for example `(4, 4)`). Do not write code that depends on how a float looks when printed. When appearance matters, format the number yourself (section 3.6).

Numbers: `int` and `float`

An `int` is exact. A `float` is stored in binary, so some decimal fractions cannot be stored exactly. The classic example:

```
func _ready() -> void:
    say(0.1 + 0.2 == 0.3)
    say(is_equal_approx(0.1 + 0.2, 0.3))
```

What the output looks like:

```
false
true
```

Never compare two floats with `==` when they were calculated. Use `is_equal_approx(a, b)`, which allows for tiny rounding differences.

To convert between the two:

```
func _ready() -> void:
    say(int(3.9))        # drops the decimals: 3
    say(int(-3.9))       # -3
    say(float(7))        # becomes a float
    say(roundi(3.5))     # rounds to nearest int: 4
    say(floori(-3.5))    # rounds down: -4
    say(ceili(3.1))      # rounds up: 4
```

What the output looks like:

```
3
-3
7.0
4
-4
4
```

Vectors

A `Vector2` bundles two numbers and comes with useful math.

```
func _ready() -> void:
    var a := Vector2(3.5, 1.5)
    var b := Vector2(0.5, 2.5)
    say(a + b)
    say(a * 2)
    say(a.x)
    say(a.length())
    say(Vector2.ZERO)
    a.x = 10.5
    say(a)
```

What the output looks like (`a.length()` is shown rounded to about six digits):

```
(4, 4)
(7, 3)
3.5
3.80789
(0, 0)
(10.5, 1.5)
```

You read the parts with `.x` and `.y` and you can assign to them. A `Vector3` works the same way with a third part `.z`. Vectors are the heart of positions and movement in Chapter 6.

Colors

A `Color` has four parts: `r`, `g`, `b`, and `a` (alpha, the opacity), each between 0 and 1.

```
func _ready() -> void:
    var from_numbers := Color(1, 0.5, 0)
    var from_name := Color.SKY_BLUE
    var from_hex := Color("#ff8800")
    out.add_theme_color_override("font_color", Color.YELLOW)
    say("Colors are floats from 0 to 1")
```

Run it: the text turns yellow. `add_theme_color_override("font_color", ...)` is a Control method that changes one color of the label's look; Chapter 8 covers themes.

NodePath and `null`

A `NodePath` is not the node itself. It is a written address, such as `^"Output"` or `^"Menu/Buttons/Start"`, that tells Godot how to find the node from where the script is. The shortcut `$Output` that we used earlier means "get the node at path `Output`." Write a NodePath literal with a caret and quotes: `^"Output"`.

`null` means "nothing here". A variable that should hold a node but has none holds `null`. Using `null` as if it were a real object is the most common runtime error: "Invalid access to property or key 'x' on a base object of type 'Nil'" (or "on a null instance"). The fix is to find out why the variable is empty.

Truthiness

In an `if` test GDScript treats several empty values as false: `false`, `0`, `0.0`, `""` (empty string), `null`, and empty collections. Everything else is true. This is handy, but when the meaning is not obvious, write the comparison out: `if lives > 0`.

> **Try it:** Declare a variable for your favorite number, your name, whether you like pizza, and a `Vector2` for the location of your house on an imaginary map. Print each with `say()`. Change one value and run again.

3.4 Static typing and type inference

GDScript is happy to run without any types:

```
var speed = 200
speed = "fast"   # legal in dynamic code: the variable can hold anything
```

That freedom is a trap. A typo or a wrong kind of value is found only when the line runs, which might be an hour into a play session. **Static typing** means writing the type you expect so Godot can check the code as you type.

Three ways to declare

```
var a = 5          # untyped: the variable can hold any value (a Variant)
var b: int = 5     # typed: only ints (and values that convert to int)
var c := 5         # inferred: Godot sees 5 is an int and makes c an int
```

`:=` is the **inference** operator: it means "figure out the type from the value on the right, then lock it in." After this line, `c` is an `int` forever. It is the shortest way to get type safety and it is what many Godot developers use inside functions.

```
func _ready() -> void:
    var speed := 200.0
    var title := "Weird Hotel"
    var origin := Vector2(0, 0)
    say(speed)
    say(title)
    say(origin)
```

When to write the type out

Inference only works when the right side has a clear type. When it does not, write the type yourself.

```
var lives: int = 3                  # fine, but "var lives := 3" is shorter
var target: Node2D = null           # needed: null alone has no type
var items: Array[String] = []       # needed to get a typed array
```

If you try `var x := null`, Godot reports that it cannot infer the type from `null`.

What the checker catches

Declare this in `_ready()` and look at the editor before running:

```
    var lives: int = "three"
```

The line gets a red underline and an error such as "Cannot assign a value of type String to variable \"lives\" with specified type int." The game will not start until you fix it. That is the entire point: the mistake costs you five seconds now instead of a bug report later.

One thing is allowed with a warning: giving a decimal to an int variable.

```
    var lives: int = 3.7   # allowed, but the editor warns about narrowing conversion
```

The value is cut to `3`. The yellow warning is Godot telling you precision is being lost; change the code so the loss is deliberate, for example `int(3.7)`.

Typing functions

Functions can declare the type of each parameter and of what they return. `-> void` means "returns nothing".

```
func add(a: int, b: int) -> int:
    return a + b

func announce(message: String) -> void:
    say(message)
```

A function with a typed return must return a value of that type on every path. Mixing the types is flagged by the editor. Use typed parameters and return types in your own functions; this book does so almost everywhere.

Casting with `as` and checking with `is`

```
func _ready() -> void:
    var value: Variant = 42
    if value is int:
        say("value is an int")
    var number := value as int
    say(number + 1)
```

What the output looks like:

```
value is an int
43
```

`is` asks "is this value of that type?" and gives `true` or `false`. `as` converts (or, for nodes, gives `null` when the object is not of that type). `Variant` is the name of the "anything" type that untyped variables use; you can write it explicitly when you really mean "any value".

> **Why it works:** Typed code is also faster, because Godot can skip checking types at run time. And the editor can offer better auto-complete: after `var out: Label`, typing `out.` shows exactly the Label's properties.
>
> **Note:** You can ask Godot to warn about every untyped variable. In **Project > Project Settings** turn on **Advanced Settings**, open **Debug > GDScript**, and set **Untyped Declaration** to Warn. This is optional, but a strong habit builder.

3.5 Operators

An **operator** is a symbol that combines or compares values.

Arithmetic

Operator	Meaning	Example	Result
`+`	add	`5 + 3`	`8`
`-`	subtract	`5 - 3`	`2`
`*`	multiply	`5 * 3`	`15`
`/`	divide	`7 / 2.0`	`3.5`
`%`	remainder (modulo), integers only	`7 % 3`	`1`
`**`	power	`2 ** 10`	`1024`

The shortcuts `+=`, `-=`, `*=`, `/=` update a variable in place: `score += 10` means `score = score + 10`.

The integer division gotcha

When **both** sides of `/` are integers, GDScript performs integer division and throws the remainder away.

```
func _ready() -> void:
    say(7 / 2)
    say(7 / 2.0)
    say(float(7) / 2)
    say(-7 / 2)
    say(7 % 3)
    say(-7 % 3)
```

What the output looks like:

```
3
3.5
3.5
-3
1
-1
```

Read this carefully:

- `7 / 2` is `3`, not `3.5`, because both numbers are ints.
- Making one side a float (`2.0`, or `float(7)`) switches to normal division.
- Division of negative integers cuts toward zero, so `-7 / 2` is `-3`.
- `%` keeps the sign of the left number, so `-7 % 3` is `-1`.

The editor warns about this with an "Integer division" warning on the line. A classic bug: `var percent = health / max_health * 100` where both are ints gives `0` for any health below max. Fix it by making one side a float:

```
func _ready() -> void:
    var health: int = 40
    var max_health: int = 100
    var wrong := health / max_health * 100
    var right := float(health) / max_health * 100.0
    say(wrong)
    say(right)
```

What the output looks like:

```
0
40.0
```

> **Warning:** Whenever you divide and the answer is suddenly 0 or too small, look for two ints. This is one of the top beginner bugs in every language that has integer division.

The `%` operator is useful for wrapping and for "every n-th time": `frame % 60 == 0` is true once every 60 frames. It does not work on floats; use `fmod(a, b)` for those.

Comparison

These give `true` or `false`.

Operator	Meaning
`==`	equal
`!=`	not equal
`<`, `>`	less than, greater than
`<=`, `>=`	less than or equal, greater than or equal

A single `=` assigns; a double `==` compares. Writing `if lives = 3:` is a classic mistake that Godot catches as an error.

Logical operators

Operator	Also written	Meaning
`and`	`&&`	true when both are true
`or`	`||`	true when at least one is true
`not`	`!`	flips true and false

Use the word forms; they read like English and are the style used in Godot's own docs.

```
func _ready() -> void:
    var has_key := true
    var door_locked := true
    say(has_key and door_locked)
    say(has_key or false)
    say(not door_locked)
```

What the output looks like:

```
true
true
false
```

`and` and `or` are *short-circuit*: in `a and b`, if `a` is false, `b` is never evaluated. This matters when `b` would crash if `a` is false, as in `if target != null and target.visible:`.

Other operators worth knowing now

```
func _ready() -> void:
    var hp := 30
    say("low" if hp < 50 else "fine")   # conditional (ternary) expression
    say(3 in [1, 2, 3])                   # membership test
    say("ell" in "hello")                 # works on text too
    say(clampi(120, 0, 100))              # keep a value in a range
```

What the output looks like:

```
low
true
true
100
```

`a if condition else b` picks one of two values. `in` tests membership. Helper functions `clamp`, `clampi` (ints), `clampf` (floats), `min`, `max`, `abs`, `mini`, `maxi`, `minf`, `maxf` do common number jobs.

Operator order follows ordinary math: `**` first, then `*`, `/`, `%`, then `+`, `-`, then comparisons, then `not`, `and`, `or`. When unsure, add parentheses.

> **Try it:** Predict, then check, the output of these three lines: `say(10 / 4)`, `say(10 % 4)`, `say(10 / 4.0)`. You should get `2`, `2`, and `2.5`.

3.6 Strings, formatting, and enums

Strings

A `String` is text in double quotes (single quotes also work). Special characters use a backslash: `\n` is a new line, `\t` a tab, `\"` a quote mark, `\\` a backslash.

```
func _ready() -> void:
    var s := "Weird Hotel"
    say(s.length())
    say(s.to_upper())
    say(s.to_lower())
    say(s[0])
    say(s.begins_with("Weird"))
    say(s.contains("Hot"))
    say(s.replace("Weird", "Normal"))
    say("  padded  ".strip_edges() + "|")
    say("-".repeat(10))
```

What the output looks like:

```
11
WEIRD HOTEL
weird hotel
W
true
true
Normal Hotel
padded|
----------
```

You join strings with `+`. Text and numbers cannot be joined directly, so convert with `str()`:

```
func _ready() -> void:
    var coins := 7
    say("Coins: " + str(coins))
    say(str("A", 1, true))
```

What the output looks like:

```
Coins: 7
A1true
```

`str()` accepts any number of values of any type and joins their text forms. To go the other way, use `"42".to_int()` and `"3.5".to_float()`. Check first with `"42".is_valid_int()` when the text comes from a player.

A multi-line string uses triple quotes:

```
func _ready() -> void:
    say("""Line one
Line two""")
```

(Inside a triple-quoted string, indentation is part of the text, so the second line starts at the left edge.)

The `%` format operator

Joining many pieces with `+` gets ugly. The `%` operator on a string fills in placeholders instead.

```
func _ready() -> void:
    var player := "Mira"
    var level := 7
    var hp := 82.5
    say("Name: %s" % player)
    say("%s is level %d" % [player, level])
    say("HP: %.1f" % hp)
    say("Level: %03d" % level)
    say("Done: %d%%" % 100)
```

What the output looks like:

```
Name: Mira
Mira is level 7
HP: 82.5
Level: 007
Done: 100%
```

Placeholders:

Placeholder	Meaning
`%s`	any value as text
`%d`	integer
`%f`	float (default six decimals)
`%.2f`	float with 2 decimals

Placeholder	Meaning
`%5d`	integer padded to width 5 with spaces
`%05d`	integer padded to width 5 with zeros
`%x`	integer in hexadecimal
`%%`	a literal percent sign

With one placeholder you give one value after `%`. With several, put the values in an array `[a, b]`, in the same order as the placeholders. A mismatch between the number of placeholders and values is an error at run time.

`String.format()`

When there are many values, named placeholders in curly braces are clearer.

```
func _ready() -> void:
    var template := "{name} found {n} coins in the {place}."
    say(template.format({"name": "Mira", "n": 3, "place": "hallway"}))
    say("{0} + {1}".format(["apples", "pears"]))
```

What the output looks like:

```
Mira found 3 coins in the hallway.
apples + pears
```

`format()` takes a dictionary (you will meet dictionaries in Chapter 4) for named keys, or an array for numbered `{0}`, `{1}`. The advantage: a template can be stored, translated, or changed without touching the code that fills it.

> **Try it:** Write a line that prints your name, your age, and your favorite food using `%` with an array, then rewrite it with `format()`. Which one do you find easier to read?

Enums

Suppose a door can be locked, closed, or open. You could use the numbers `0`, `1`, and `2`, but then `if state == 1` tells the reader nothing. An **enum** gives names to the numbers.

```
extends Control

enum DoorState { LOCKED, CLOSED, OPEN }

@onready var out: Label = $Output

var door: DoorState = DoorState.LOCKED

func _ready() -> void:
    out.text = "Door state number: " + str(door) + "\n"
    out.text += "Door state name: " + DoorState.keys()[door] + "\n"
    door = DoorState.OPEN
    out.text += "Now: " + DoorState.keys()[door]
```

What the output looks like:

```
Door state number: 0
Door state name: LOCKED
Now: OPEN
```

How enums work:

- By default the first name is `0`, the next `1`, and so on.
- You can choose your own numbers: `enum Size { SMALL = 1, BIG = 10 }`.
- `DoorState.keys()` returns the list of names, so `DoorState.keys()[door]` turns a number back into its name. This is handy for debugging.
- Using the enum name as a type (`var door: DoorState`) lets the editor offer the choices in auto-complete.
- An enum written without a name, `enum { LOCKED, CLOSED, OPEN }`, creates plain constants you use directly as `LOCKED`.

Enum members use CONSTANT_CASE and the enum itself uses PascalCase.

3.7 The life of a script: lifecycle functions

So far all our code ran in `_ready()`. Where does that function come from? You never call it. **Godot** calls it, at the right moment. Functions like this are called **lifecycle functions** (or virtual functions or callbacks), and they all start with an underscore. You choose which to write; if you do not write one, Godot skips it.

Function	When Godot calls it	Typical use
`_init()`	the moment the object is created	very early setup (rarely needed in nodes)
`_ready()`	once, after the node and all its children are in the scene	set up, find nodes, show first text
`_process(delta)`	every drawn frame	things that change smoothly: timers, animation, UI
`_physics_process(delta)`	at a fixed rate (60 times per second by default)	movement and physics
`_input(event)`	whenever the player gives input	key presses, mouse clicks
`_exit_tree()`	when the node leaves the scene	cleanup

`_ready`

`_ready()` runs once. Godot guarantees that when it runs, all the node's children are ready, which is why we can safely look up `$Output` there.

`_process` and delta

`_process(delta)` runs once per frame. A fast computer draws many frames per second and a slow one fewer, so the time between frames changes. The parameter `delta` is the number of seconds since the previous frame (about 0.016 on a 60 FPS screen). Multiply movement and timers by `delta` to make them take the same *real time* on every computer.

```
extends Control
```

```gdscript
@onready var out: Label = $Output

var elapsed: float = 0.0

func _process(delta: float) -> void:
    elapsed += delta
    out.text = "Seconds since start: %.1f" % elapsed
```

Run the scene (F6). The label counts up in tenths of a second, in real time.

What the output looks like (at about three seconds):

```
Seconds since start: 3.0
```

A common mistake is to add a fixed amount every frame (`x += 5`). That makes the game run at different speeds on different machines. Use `x += 5 * delta`, which means "5 units per second". Chapter 6 builds on this idea.

`_physics_process`

`_physics_process(delta)` is called on a fixed schedule, 60 times a second unless you change **Project Settings > Physics > Common > Physics Ticks per Second**. Because it ticks regularly, it is the right place for anything involving collisions and moving bodies (Chapter 7). Its `delta` is almost constant. Here we count ticks and print once per second:

```gdscript
extends Node

var ticks: int = 0

func _physics_process(_delta: float) -> void:
    ticks += 1
    if ticks % 60 == 0:
        print("One second passed, ticks: ", ticks)
```

Attach this to a plain `Node` in a new scene (or temporarily to the Playground root, replacing its script) and run it. A line appears in the Output panel every second.

What the output looks like:

```
One second passed, ticks: 60
One second passed, ticks: 120
One second passed, ticks: 180
```

The underscore in `_delta` tells Godot (and readers) that we do not use that parameter, which avoids a warning.

> **Note:** `_process` runs once per frame and the frame rate varies; `_physics_process` runs at the fixed physics rate. If you only remember one rule: *physics and movement of bodies in* `_physics_process`*; visuals and UI in* `_process`.

`_input`

`_input(event)` receives every input event: key presses and releases, mouse movement, button clicks. The `event` parameter is an object whose type tells you what happened.

```
extends Control

@onready var out: Label = $Output

func _input(event: InputEvent) -> void:
    var key := event as InputEventKey
    if key == null:
        return
    if key.pressed and not key.echo:
        out.text = "You pressed: " + OS.get_keycode_string(key.keycode)
```

Run the scene and press keys. The label shows the key's name.

What the output looks like after pressing the Space bar and then A:

```
You pressed: Space
You pressed: A
```

Here is how it works:

1. `event as InputEventKey` gives the event as a keyboard event, or `null` if it was something else (mouse, controller). We return early when it is not a key.
2. `key.pressed` is true on press and false on release. `key.echo` is true for the repeated events produced while a key is held down. We want only the first press, so we test `key.pressed and not key.echo`.
3. `key.keycode` holds which key; `OS.get_keycode_string()` turns it into readable text.

For game controls you will usually not check specific keys. Instead you define named actions such as `"jump"` in the **Input Map** and ask whether the action happened (`event.is_action_pressed("jump")`). Chapter 6 covers the Input Map.

> **Try it:** Change the `_input` script so that pressing the Escape key (`KEY_ESCAPE`) prints "Goodbye" and quits the game with `get_tree().quit()`. Hint: compare `key.keycode == KEY_ESCAPE`.

Order of events

When you run a scene, Godot calls `_ready()` first (once), then repeats frame after frame: input events, `_physics_process` ticks, and `_process` calls. You will see that order in the lab.

Lab 3: Hero Report Card

Goal: Build a small program that stores a hero's stats in typed variables, formats a report in a Label, reacts to the keyboard, and uses all four lifecycle functions.

Time: 30-45 minutes.

Scene tree:

```
HeroReport   (Control)   <- script hero_report.gd
└── Output   (Label)
```

Steps

1. Create a new scene with a `Control` root named `HeroReport`. Add a `Label` child named `Output`.
2. Set the Anchors Preset of both nodes to **Full Rect**.
3. Select `Output`. In the Inspector set **Autowrap Mode** to **Word (Smart)** so long lines wrap.
4. Save the scene as `res://hero_report.tscn`.
5. Attach a script to `HeroReport`, saved as `res://hero_report.gd`, and type the code below.
6. Press **F6** to run.

> **Note:** This lab uses a few things Chapter 4 teaches properly: `elif` (short for "else if"), `return` to leave a function early, and writing your own functions with parameters. Read them as you would read a recipe; the comments and the Expected result tell you what they do.

File: `res://hero_report.gd`, attached to `HeroReport` (Control).

```gdscript
extends Control

enum Rank { ROOKIE, VETERAN, LEGEND }

const TITLE: String = "HERO REPORT"
const MAX_LEVEL: int = 50

var hero_name: String = "Mira"
var level: int = 7
var hp: float = 82.5
var max_hp: float = 100.0
var position_xy: Vector2 = Vector2(3.5, 4.5)
var banner_color: Color = Color.GOLD
var is_alive: bool = true
var rank: Rank = Rank.ROOKIE

var elapsed: float = 0.0
var last_second: int = -1
var physics_ticks: int = 0
var key_presses: int = 0

@onready var out: Label = $Output

func _ready() -> void:
    out.add_theme_color_override("font_color", banner_color)
    out.add_theme_font_size_override("font_size", 24)
    update_rank()
    refresh()
    print("Report ready. Keys: L = level up, H = take 10 damage, R = reset.")

func _process(delta: float) -> void:
    elapsed += delta
    var whole_seconds: int = int(elapsed)
    if whole_seconds != last_second:
        last_second = whole_seconds
        refresh()

func _physics_process(_delta: float) -> void:
    physics_ticks += 1
    if physics_ticks % 60 == 0:
        print("Physics ticks so far: ", physics_ticks)

func _input(event: InputEvent) -> void:
```

```
    var key := event as InputEventKey
    if key == null or not key.pressed or key.echo:
        return
    key_presses += 1
    if key.keycode == KEY_L:
        level = mini(level + 1, MAX_LEVEL)
        update_rank()
    elif key.keycode == KEY_H:
        take_damage(10.0)
    elif key.keycode == KEY_R:
        reset_hero()
    refresh()

func take_damage(amount: float) -> void:
    hp = maxf(hp - amount, 0.0)
    if hp == 0.0 and is_alive:
        is_alive = false
        push_warning("%s has fallen!" % hero_name)

func reset_hero() -> void:
    level = 7
    hp = max_hp
    is_alive = true
    update_rank()

func update_rank() -> void:
    if level < 10:
        rank = Rank.ROOKIE
    elif level < 40:
        rank = Rank.VETERAN
    else:
        rank = Rank.LEGEND

func refresh() -> void:
    var hp_percent: float = hp / max_hp * 100.0
    var text: String = "=== %s ===\n" % TITLE
    text += "Name:  %s\n" % hero_name
    text += "Level: %d / %d\n" % [level, MAX_LEVEL]
    text += "HP:    %.1f / %.1f (%.1f%%)\n" % [hp, max_hp, hp_percent]
    text += "Rank:  %s\n" % Rank.keys()[rank]
    text += "Position: %s\n" % str(position_xy)
    text += "Distance from origin: %.1f\n" % position_xy.length()
    text += "Alive: %s\n" % str(is_alive)
    text += "Key presses: %d\n" % key_presses
    text += "Time: %d s" % int(elapsed)
    out.text = text
```

Expected result

On start the window shows gold text (this is the first second; the time value counts up):

```
=== HERO REPORT ===
Name:  Mira
Level: 7 / 50
HP:    82.5 / 100.0 (82.5%)
Rank:  ROOKIE
Position: (3.5, 4.5)
Distance from origin: 5.7
Alive: true
Key presses: 0
Time: 0 s
```

Test it:

- Press **L** three times: Level becomes 10 and Rank changes to VETERAN.

- Press **H** nine times: HP goes to 0.0 and `Alive: false`, and the Debugger Errors tab shows the warning "Mira has fallen!". Pressing H again changes nothing (the HP stops at 0).
- Press **R**: the hero returns to level 7 and full HP.
- Watch the Output panel: a "Physics ticks so far" line appears every second.
- The `Time` line increases every second without a key press.

Stretch goals

- Add a second `Vector2` and print the distance between the two positions with `position_xy.distance_to(other)`.
- Add a `const` for the damage amount and use it instead of the bare `10.0`.
- Show the rank in a different color for each `Rank`.

Chapter summary

- A script is a text file attached to a node; `extends` names the node type the script builds on, and the script inherits everything that type can do.
- GDScript uses tabs for indentation, `#` for comments, and `print`, `print_rich`, `push_warning`, and `push_error` for messages.
- Declare variables with `var` and constants with `const`; member variables live as long as the node, local variables only during a function call.
- The built-in types you will use most are `int`, `float`, `bool`, `String`, `Vector2`, `Vector3`, `Color`, `NodePath`, and `null`.
- Static typing (`var x: int`, `var y := 5`, `-> void`) lets Godot catch mistakes before the game runs; use `:=` when the type is obvious.
- Dividing two ints gives an int; make one operand a float for ordinary division. Compare floats with `is_equal_approx`.
- Use `%` formatting, `format()`, and `str()` to build text; use `enum` to give names to groups of related numbers.
- Godot calls `_ready` once, `_process` every frame, `_physics_process` at a fixed rate, and `_input` for each input event; multiply by `delta` to make time-based changes independent of frame rate.

Review questions

1. What does `extends Control` at the top of a script mean?
2. True or false: you may freely mix tab and space indentation in one GDScript file.
3. What is the difference between `print("A", 1)` and `prints("A", 1)`?
4. Name one situation where `push_warning` is a better choice than `print`.
5. What does `var c := 5` do that `var c = 5` does not?
6. What is the value of `9 / 2`, and what is the value of `9 / 2.0`?
7. Why should you not write `if a + b == 0.3:` for float variables, and what should you use instead?
8. What does `"%s has %d coins" % ["Mira", 12]` produce?
9. Given `enum Mood { SAD, OKAY, HAPPY }`, what is the number stored in `Mood.HAPPY`, and how do you get the name `"HAPPY"` back from the number?
10. Multiple choice: Which function is the best place to move a character that uses physics? (a) `_ready` (b) `_process` (c) `_physics_process` (d) `_input`.

Exercises

1. **Practice.** In the playground scene, declare variables for a shop item: its name (String), price (float), quantity (int), and whether it is in stock (bool). Print one summary line with `%` formatting, for example `Lamp x3 at 9.50 each (in stock)`.
2. **Practice.** Write a function `average(a: int, b: int) -> float` that returns the exact average of two ints (so `average(3, 4)` is `3.5`). Print the result for `3, 4` and for `10, 20`. Explain in a comment why a naive `(a + b) / 2` would be wrong.
3. **Apply.** Create a `Stopwatch` script: it shows the time as `MM:SS` in a Label using `_process`. Pressing the Space bar (use `_input`) pauses and resumes it. Use `%02d` in your format string.
4. **Apply.** Define `enum Weather { SUNNY, CLOUDY, RAINY }`. Press the W key to cycle to the next weather using `%` (modulo), and show the weather name in the Label. Hint: `(current + 1) % Weather.size()`.
5. **Challenge.** Extend the Hero Report lab: add an `xp` int, give 25 XP per press of the X key, and level up automatically (and subtract 100 XP) whenever xp reaches 100. Display an XP bar made of text, like `[#####-----]`, using `"#".repeat(n)` and `"-".repeat(10 - n)`. Be careful with integer division when you compute `n`.

Common mistakes

- **Symptom:** Error "Unexpected indentation" or "Used space character for indentation instead of tab". **Cause:** spaces and tabs are mixed, or a line is indented when it should not be. **Fix:** use **Edit > Indentation > Convert Indent to Tabs**, and let the editor's Tab key do the indenting.
- **Symptom:** A percentage or ratio is always 0. **Cause:** both numbers are ints, so `/` drops the decimals. **Fix:** convert one operand with `float()` or write a decimal literal like `100.0`.
- **Symptom:** Error when adding text and a number, such as `"Score: " + score`. **Cause:** GDScript will not join a String and an int. **Fix:** write `"Score: " + str(score)` or `"Score: %d" % score`.
- **Symptom:** "Invalid access to property or key 'text' on a base object of type 'Nil'" (or null instance). **Cause:** the variable holding a node is `null`, often because the node path in `$Name` does not match the scene tree. **Fix:** check the exact node name and capitalization in the Scene dock.
- **Symptom:** A float comparison such as `0.1 + 0.2 == 0.3` is false. **Cause:** binary floats cannot store most decimals exactly. **Fix:** use `is_equal_approx()`.
- **Symptom:** Movement or timers run faster on a faster computer. **Cause:** a fixed amount is added each frame in `_process`. **Fix:** multiply by `delta`.

Further reading

- GDScript reference: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
- GDScript style guide: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_styleguide.html
- Static typing in GDScript: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/static_typing.html
- GDScript format strings: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_format_string.html
- Idle and physics processing: https://docs.godotengine.org/en/stable/tutorials/scripting/idle_and_physics_processing.html

Chapter 4: GDScript II: Decisions, Loops, Functions, and Collections

Learning objectives

By the end of this chapter you will be able to:

- Choose between `if` / `elif` / `else` and `match` to make a script take different paths.
- Repeat work with `for` and `while` loops, and control them with `break` and `continue`.
- Write functions with parameters, default values, and typed return values, and explain variable scope.
- Use lambdas and `Callable` values with array methods such as `map`, `filter`, and `sort_custom`.
- Store and process data in typed arrays and dictionaries, using their common methods and the `in` operator.
- Implement simple algorithms (sum, maximum, counting, searching), generate random numbers correctly in Godot 4, and build a complete text-based guessing game with `Label`, `LineEdit`, and `Button` nodes.

Key terms

- **Condition**: an expression that is `true` or `false`, used to decide what to do.
- **Branch**: one of the possible paths a decision can take.
- **Loop**: code that repeats, either a fixed number of times or until a condition changes.
- **Iteration**: one pass through the body of a loop.
- **Function**: a named, reusable block of code that can take inputs and return an output.
- **Parameter / argument**: a parameter is a name in a function's definition; an argument is the value you pass in a call.
- **Return value**: the value a function hands back to the code that called it.
- **Scope**: the part of the code where a variable name can be used.
- **Array**: an ordered list of values, read by position (index).
- **Dictionary**: a collection of key/value pairs, read by key.
- **Lambda**: a small function written inline without a name.
- **Callable**: a value that stands for a function and can be passed around and called later.

4.1 Making decisions: `if` and `match`

A program that always does the same thing is not a game. Games react: if the player has the key, open the door. Decisions are built from **conditions**, which are expressions that are `true` or `false` (Chapter 3).

For the examples in this chapter, reuse the **Playground** scene from Chapter 3: a `Control` root named `Playground` with a `Label` child named `Output`, both set to Full Rect. Its script, `res://playground.gd`, looks like this:

File: `res://playground.gd`, attached to `Playground` (Control).

```
extends Control

@onready var out: Label = $Output
```

```
func say(text: Variant) -> void:
    out.text += str(text) + "\n"
    print(text)

func _ready() -> void:
    say("Ready")
```

When an example says "fragment: put this inside `_ready()`", replace the `say("Ready")` line with the fragment, indented by one tab. When an example shows a complete function, put it anywhere at the left edge of the file, then call it from `_ready()`.

`if`, `elif`, `else`

```
func _ready() -> void:
    var temperature: int = 22
    if temperature < 0:
        say("Freezing")
    elif temperature < 15:
        say("Cold")
    elif temperature < 25:
        say("Pleasant")
    else:
        say("Hot")
```

What the output looks like:

```
Pleasant
```

Godot tests the conditions from top to bottom and runs only the **first** branch whose condition is true; the rest are skipped. That is why the order matters: `temperature < 25` is also true for 10, but `temperature < 15` came first and caught it. The final `else` is optional and catches everything left over. Each of `if`, `elif`, `else` ends with a colon, and its body is indented one level.

You can combine conditions with `and`, `or`, and `not`, and nest `if` blocks:

```
func can_open_door(has_key: bool, door_locked: bool, is_dark: bool) -> bool:
    if door_locked and not has_key:
        return false
    if is_dark:
        return has_key or not door_locked
    return true
```

> **Why it works:** The first `if` returns early for the case that can never succeed. Handling the "no" cases first and leaving the main path at the end is called using *guard clauses*, and it keeps code flat and easy to read.

`pass`

A block cannot be empty. When you want a placeholder for code you have not written yet, write `pass`. It does nothing.

```
func open_inventory() -> void:
    pass  # TODO: build the inventory screen
```

`match`: choosing between many exact values

When one value is compared against many possibilities, `match` is cleaner than a long `elif` chain. It looks at a value and runs the first branch whose **pattern** fits.

```
func describe(value: Variant) -> String:
    match value:
        0:
            return "zero"
        1, 2, 3:
            return "a small number"
        "hello":
            return "a greeting"
        []:
            return "an empty array"
        [1, 2]:
            return "the array one-two"
        [var first, var second]:
            return "a pair starting with %s and ending with %s" % [first, second]
        {"name": "Mira"}:
            return "a dictionary about Mira"
        var other:
            return "something else: %s" % str(other)
```

Call it with several different values:

```
func _ready() -> void:
    say(describe(0))
    say(describe(2))
    say(describe("hello"))
    say(describe([]))
    say(describe([5, 6]))
    say(describe({"name": "Mira", "level": 7}))
    say(describe(7.5))
```

What the output looks like:

```
zero
a small number
a greeting
an empty array
a pair starting with 5 and ending with 6
a dictionary about Mira
something else: 7.5
```

The kinds of pattern you can use:

Pattern	Matches
`5` or `"hello"`	a literal value
`1, 2, 3`	any one of several values (comma = "or")
`SomeEnum.VALUE` or a constant	that constant's value
`var x`	anything, and puts it in a new variable `x` (a catch-all with a name)
`_`	anything, without naming it (the wildcard)
`[a, b]`, `[]`	an array of exactly that shape
`[var a, var b]`	an array of two items, binding each to a variable

Pattern	Matches
`{"key": value}`	a dictionary that contains that key with that value (extra keys are allowed)

Two rules to remember:

1. **No fall-through.** Unlike C or Java, only one branch runs, and you never write `break`. If you really want to skip to the next matching pattern, write `continue` inside the branch.
2. **Ranges are not patterns.** `match score: 90..100:` does not work. For ranges use `if`/`elif`.

`match` is perfect with enums, because every named state gets its own branch:

```
enum DoorState { LOCKED, CLOSED, OPEN }

func door_text(state: DoorState) -> String:
    match state:
        DoorState.LOCKED:
            return "The door is locked."
        DoorState.CLOSED:
            return "The door is closed but not locked."
        DoorState.OPEN:
            return "The door is wide open."
    return "Unknown door state"
```

The last `return` is a safety net so the function returns a `String` on every path. (Godot's checker would complain otherwise.)

> **Try it:** Write a function `grade(score: int) -> String` that returns "A" for 90 or more, "B" for 80 to 89, "C" for 70 to 79, and "F" otherwise. Use `if`/`elif`, because ranges do not work in `match`. Test it with 95, 80, 79, and 12.

4.2 Loops

A **loop** repeats a block of code. GDScript has two: `for` (repeat for each item) and `while` (repeat as long as a condition holds).

`for` with `range`

`range()` creates a sequence of integers.

Call	Sequence
`range(5)`	0, 1, 2, 3, 4
`range(1, 6)`	1, 2, 3, 4, 5
`range(0, 10, 2)`	0, 2, 4, 6, 8
`range(5, 0, -1)`	5, 4, 3, 2, 1

The end value is never included. The loop below builds one line of text:

```
func _ready() -> void:
    var line: String = ""
    for i in range(5):
        line += str(i) + " "
```

```
    say(line)
    say("Countdown:")
    for i in range(3, 0, -1):
        say(i)
    say("Liftoff!")
```

What the output looks like:

```
0 1 2 3 4
Countdown:
3
2
1
Liftoff!
```

In GDScript you may write `for i in 5:` as a short form of `for i in range(5):`. Both are common.

`for` over arrays, dictionaries, and strings

A `for` loop can walk through any collection. The loop variable takes each item in turn.

```
func _ready() -> void:
    var fruits := ["apple", "pear", "plum"]
    for fruit in fruits:
        say(fruit)

    for i in fruits.size():
        say("%d: %s" % [i, fruits[i]])

    var stock := {"apples": 3, "pears": 0}
    for item in stock:
        say("%s: %d" % [item, stock[item]])

    for letter in "abc":
        say(letter)
```

What the output looks like:

```
apple
pear
plum
0: apple
1: pear
2: plum
apples: 3
pears: 0
a
b
c
```

Notes:

- Looping over an array gives you **values**. If you also need the position, loop over `range(array.size())` or `array.size()` and index into the array.
- Looping over a dictionary gives you the **keys**. Use `stock[item]` to get the value.
- Looping over a string gives you one character at a time.
- Do not add or remove items from the array you are looping over inside the loop; it causes skipped items. Loop over a copy (`for x in list.duplicate():`) if you must.

`while`

Use `while` when you do not know in advance how many repeats are needed.

```
func _ready() -> void:
    var n: int = 1
    var steps: int = 0
    while n < 100:
        n *= 2
        steps += 1
    say("Reached %d after %d doublings" % [n, steps])
```

What the output looks like:

```
Reached 128 after 7 doublings
```

> **Warning:** If the condition of a `while` loop never becomes false, the game freezes forever, because the loop never lets Godot draw the next frame. Make sure something inside the loop changes the condition. If it happens, press the Stop button in the editor (or F8).

A `while` loop in a game is rare; waiting for something over time is done with `_process` or `await` (Chapter 5), never by looping until it happens.

`break` and `continue`

- `break` leaves the loop immediately.
- `continue` skips the rest of this iteration and goes to the next one.

```
func _ready() -> void:
    # Find the first multiple of 7 above 50.
    for i in range(51, 100):
        if i % 7 == 0:
            say("First multiple of 7 above 50: %d" % i)
            break

    # Print only the odd numbers from 1 to 9.
    var odds: String = ""
    for i in range(1, 10):
        if i % 2 == 0:
            continue
        odds += str(i) + " "
    say(odds)
```

What the output looks like:

```
First multiple of 7 above 50: 56
1 3 5 7 9
```

Nested loops

A loop inside a loop repeats the inner one for each pass of the outer. This prints a small multiplication table:

```
func _ready() -> void:
    for row in range(1, 4):
        var line: String = ""
```

```
        for col in range(1, 5):
            line += "%3d" % (row * col)
        say(line)
```

What the output looks like (the Label may use a non-monospace font, so the columns may not line up exactly):

```
  1  2  3  4
  2  4  6  8
  3  6  9 12
```

> **Try it:** Change the nested loop to print a 5 by 5 grid of `#` and `.` where a `#` is shown when `row == col`. You should see a diagonal.

4.3 Functions

You have already written functions. A **function** is a named block of code you can run, whenever you like, by writing its name followed by parentheses. Functions are how you avoid copying the same lines in ten places.

Anatomy of a function

```
func add(a: int, b: int) -> int:
    return a + b
```

- `func` starts the definition.
- `add` is the function's name (snake_case).
- `a: int, b: int` are the **parameters**, each with a type.
- `-> int` says the function returns an int. Use `-> void` when it returns nothing.
- `return` sends the result back and ends the function at once.

Calling it:

```
func _ready() -> void:
    var total: int = add(2, 3)
    say(total)
    say(add(10, add(1, 1)))
```

What the output looks like:

```
5
12
```

The values `2` and `3` in the call are the **arguments**. A function call is itself an expression, so it can be placed inside another call.

Functions can be defined anywhere in the file; the order does not matter. You may call `add` above the place where it is written.

Default parameter values

A parameter can have a default, used when the caller leaves it out. Parameters with defaults must come after the ones without.

```
func greet(who: String, greeting: String = "Hello") -> String:
    return "%s, %s!" % [greeting, who]

func _ready() -> void:
    say(greet("Mira"))
    say(greet("Mira", "Welcome"))
```

What the output looks like:

```
Hello, Mira!
Welcome, Mira!
```

GDScript has no named arguments; you pass values in the order of the parameters.

Returning more than one value

A function returns one value, but that value can be a collection.

```
func min_and_max(values: Array[int]) -> Dictionary:
    return {"min": values.min(), "max": values.max()}
```

Early return and guard clauses

A `return` can appear anywhere. It is common to exit early when input is invalid:

```
func safe_divide(a: float, b: float) -> float:
    if b == 0.0:
        push_warning("safe_divide: division by zero")
        return 0.0
    return a / b
```

A typed function must return the right type on every path; a path that ends without `return` is an error for functions that declare a return type.

Recursion

A function may call itself. Each call needs a stopping condition or it never ends.

```
func factorial(n: int) -> int:
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

`factorial(5)` returns `120`. Use recursion sparingly; many problems read more clearly as loops.

Scope: where a name can be used

A variable can be used only inside the block where it was declared, from the line it was declared to the end of that block.

```
var score: int = 100  # member variable: visible in the whole file

func demo() -> void:
    var bonus: int = 5       # local to demo()
    if bonus > 0:
        var message: String = "Bonus!"   # local to this if block
        say(message)
    # say(message)  <- would be an error here: message does not exist anymore
    say(score + bonus)
```

Rules to remember:

- Member variables (top of the file) are visible everywhere in the script.
- Local variables disappear when their block ends, and the next call starts fresh.
- If a local variable has the same name as a member variable, the local one *shadows* it and the editor warns you. Pick different names.
- A function's parameters are local variables of that function.

Arguments of simple types (`int`, `float`, `bool`, `String`, vectors) are **copied** into the function: changing the parameter does not change the caller's variable. Arrays and dictionaries are different; see section 4.4.

Lambdas and `Callable`

Sometimes you need a tiny function for one use. A **lambda** is a function with no name, written with `func` right where you need it. It is stored in a variable of type `Callable`.

```
func _ready() -> void:
    var doubler := func(x: int) -> int: return x * 2
    say(doubler.call(21))

    var greet := func(who: String) -> String:
        return "Hi, " + who
    say(greet.call("Mira"))
```

What the output looks like:

```
42
Hi, Mira
```

To run a `Callable` you use `.call(arguments)`. A named function can also be used as a `Callable` by writing its name without parentheses:

```
func add(a: int, b: int) -> int:
    return a + b

func _ready() -> void:
    var operation: Callable = add
    say(operation.call(2, 3))
```

This prints `5`. Callables matter later because signals (Chapter 5) connect to them: "when this happens, call that."

Arrays have methods that take a Callable and apply it to every item. This style is very handy and short:

```
func _ready() -> void:
    var nums: Array[int] = [5, 2, 8, 1, 4]
    say(nums.map(func(n): return n * n))
    say(nums.filter(func(n): return n % 2 == 0))
    say(nums.reduce(func(total, n): return total + n, 0))
    nums.sort_custom(func(a, b): return a > b)
    say(nums)
    say(nums.any(func(n): return n > 7))
    say(nums.all(func(n): return n > 0))
```

What the output looks like:

```
[25, 4, 64, 1, 16]
[2, 8, 4]
20
[8, 5, 4, 2, 1]
true
true
```

- `map` makes a new array by applying the function to each item.
- `filter` keeps the items for which the function returns `true`.
- `reduce` folds all the items into one value; the second argument (`0`) is the starting value.
- `sort_custom` sorts in place; the function returns `true` when `a` should come before `b`.
- `any` and `all` ask whether at least one item, or every item, passes the test.

> **Note:** Lambdas can read variables from the function around them. Do not worry about the details yet. Use them for short one-line jobs like these and use ordinary named functions for anything longer.
>
> **Try it:** Using `filter`, make a new array from `[3, 14, 15, 9, 26]` that keeps only numbers greater than 10. Then use `map` to turn each number into text such as `"n=14"`, using `"n=%d" % n`.

4.4 Arrays

An **array** is an ordered list. Items are numbered from **0**, and the number is called the **index**.

```
func _ready() -> void:
    var items: Array[String] = ["key", "bone", "string"]
    say(items[0])
    say(items[-1])
    say(items.size())
    items[1] = "flute"
    say(items)
```

What the output looks like:

```
key
string
3
["key", "flute", "string"]
```

A negative index counts from the end: `items[-1]` is the last item. An index that is out of range (for example `items[10]` here) stops the script with an error such as "Out of bounds get index '10'". Check `size()` first when the index comes from somewhere you do not control.

Typed arrays

`Array[String]` means "an array that holds only Strings." The type goes in angle brackets. Typed arrays catch mistakes (adding a number to a list of names) and let the editor suggest the right methods on each item.

```
var names: Array[String] = []
var scores: Array[int] = [10, 20, 30]
var positions: Array[Vector2] = [Vector2(0, 0), Vector2(5, 5)]
```

If you write `var list := [1, 2, 3]`, Godot makes an ordinary untyped `Array`. To get a typed one you must write the type on the left. Adding a wrong type to a typed array fails at run time with an error message.

Common methods

```
func _ready() -> void:
    var inv: Array[String] = ["key", "bone"]
    inv.append("string")        # add at the end
    inv.push_front("map")       # add at the front
    inv.insert(1, "coin")       # insert at index 1
    say(inv)
    say(inv.has("key"))
    say(inv.find("bone"))
    say(inv.find("lamp"))
    inv.erase("coin")           # remove the first item equal to "coin"
    inv.remove_at(0)            # remove the item at index 0 ("map")
    say(inv)
    var last: String = inv.pop_back()
    say(last)
    say(inv.front())
    say(inv.back())
    say(inv.is_empty())
```

What the output looks like:

```
["map", "coin", "key", "bone", "string"]
true
3
-1
["key", "bone", "string"]
string
key
bone
false
```

Method	What it does
`append(v)` / `push_back(v)`	add to the end
`push_front(v)`	add to the front
`insert(i, v)`	insert at index `i`
`pop_back()` / `pop_front()`	remove and return the last / first item
`remove_at(i)`	remove the item at index `i`

Method	What it does
`erase(v)`	remove the first item equal to `v`
`size()`	number of items
`is_empty()`	true when there are no items
`has(v)`	true if `v` is in the array
`find(v)`	index of the first match, or `-1`
`count(v)`	how many items equal `v`
`clear()`	remove everything
`sort()` / `reverse()` / `shuffle()`	reorder in place
`min()` / `max()`	smallest / largest item (numbers)
`slice(from, to)`	a new array of the items from `from` up to (not including) `to`
`pick_random()`	one random item
`append_array(other)`	add all items of another array
`duplicate()`	make an independent copy

```
func _ready() -> void:
    var nums: Array[int] = [5, 3, 9, 1]
    nums.sort()
    say(nums)
    nums.reverse()
    say(nums)
    say(nums.max())
    say(nums.min())
    say(nums.slice(1, 3))
```

What the output looks like:

```
[1, 3, 5, 9]
[9, 5, 3, 1]
9
1
[5, 3]
```

The `in` operator

`in` is a short way to ask "is this in the collection?":

```
func _ready() -> void:
    var inv: Array[String] = ["key", "bone"]
    if "key" in inv:
        say("You have the key")
    say("lamp" in inv)
```

What the output looks like:

```
You have the key
false
```

It is equivalent to `inv.has("key")`. Use whichever reads better; `in` is popular in conditions.

Arrays are shared, not copied

When you assign an array to another variable, or pass it to a function, both names point at the **same** list. Changing it through one name changes it for the other.

```
func add_one(list: Array[int]) -> void:
    list.append(1)

func _ready() -> void:
    var a: Array[int] = [5]
    var b := a
    b.append(7)
    add_one(a)
    say(a)
    var c = a.duplicate()
    c.append(99)
    say(a)
    say(c)
```

What the output looks like:

```
[5, 7, 1]
[5, 7, 1]
[5, 7, 1, 99]
```

`duplicate()` is the way to get a separate copy. Dictionaries behave the same way.

Arrays of arrays and packed arrays

Items can be arrays themselves, which gives you grids:

```
func _ready() -> void:
    var grid := [[1, 2, 3], [4, 5, 6]]
    say(grid[1][2])
```

This prints `6` (row 1, then column 2).

Some Godot functions return special compact arrays, such as `PackedStringArray`. For example `"a,b,c".split(",")` returns one. You can loop over it and use `size()` and indexing just like an ordinary array:

```
func _ready() -> void:
    var parts := "red,green,blue".split(",")
    say(parts.size())
    for color_name in parts:
        say(color_name)
```

What the output looks like:

```
3
red
green
blue
```

4.5 Dictionaries

An array finds things by position. A **dictionary** finds things by a name called a **key**. Each key is paired with a **value**. It is the natural container for "properties of one thing": a character's name, level, and items.

```
func _ready() -> void:
    var hero := {"name": "Mira", "level": 7, "items": ["key"]}
    say(hero["name"])
    hero["level"] += 1
    hero["gold"] = 50
    say(hero["level"])
    say(hero.has("gold"))
    say(hero.get("mana", 0))
    say(hero.size())
    hero.erase("gold")
    say(hero.keys())
    say(hero.values())
```

What the output looks like:

```
Mira
8
true
0
4
["name", "level", "items"]
["Mira", 8, ["key"]]
```

Key points:

- Write a dictionary with curly braces and `key: value` pairs.
- Read with `d[key]`; assign with `d[key] = value`. Assigning to a key that does not exist **creates** it.
- Reading a key that does not exist with `d["missing"]` is an error. Use `d.get("missing", default)` to supply a fallback instead.
- `has(key)` and `key in d` test whether a key exists. Note that `in` checks keys, not values.
- `keys()` and `values()` return arrays. Keys stay in the order you added them.
- Keys are usually Strings, but they can be numbers, vectors, and more.
- A Lua-style form `{name = "Mira"}` with `hero.name` exists, but it is easy to confuse with properties. Use quoted string keys and `[ ]` in this book.

Looping over a dictionary

```
func _ready() -> void:
    var prices := {"apple": 3, "pear": 5, "plum": 4}
    for fruit in prices:
        say("%s costs %d" % [fruit, prices[fruit]])
    var total: int = 0
    for price in prices.values():
        total += price
    say("Total: %d" % total)
```

What the output looks like:

```
apple costs 3
pear costs 5
```

```
plum costs 4
Total: 12
```

Typed dictionaries

From Godot 4.4 on you can declare the key and value types:

```
var prices: Dictionary[String, int] = {"apple": 3, "pear": 5}
```

If your editor is an older 4.x version, this line is an error; use a plain `Dictionary` instead. Check your version in the editor's title bar or in **Help > About**.

A counting pattern

Dictionaries are the standard tool for "how many of each?":

```
func _ready() -> void:
    var words := "the cat and the hat and the bat".split(" ")
    var counts := {}
    for w in words:
        counts[w] = counts.get(w, 0) + 1
    for w in counts:
        say("%s: %d" % [w, counts[w]])
```

What the output looks like:

```
the: 3
cat: 1
and: 2
hat: 1
bat: 1
```

The key line is `counts[w] = counts.get(w, 0) + 1`: "take the current count for `w` (or 0 if none yet), add one, and store it."

> **Try it:** Use the counting pattern on the string `"mississippi"` (loop over its letters directly). The result should be m: 1, i: 4, s: 4, p: 2.

4.6 Algorithms and randomness

An **algorithm** is just a clear list of steps to solve a problem. Most game logic is a handful of small algorithms built from loops and conditions.

Sum, average, and maximum

```
func sum_of(values: Array[int]) -> int:
    var total: int = 0
    for v in values:
        total += v
    return total

func average_of(values: Array[int]) -> float:
    if values.is_empty():
```

```
        return 0.0
    return float(sum_of(values)) / values.size()

func max_of(values: Array[int]) -> int:
    if values.is_empty():
        return 0
    var best: int = values[0]
    for v in values:
        if v > best:
            best = v
    return best

func _ready() -> void:
    var data: Array[int] = [3, 8, 1, 9, 4]
    say("Sum: %d" % sum_of(data))
    say("Average: %.1f" % average_of(data))
    say("Max: %d" % max_of(data))
```

What the output looks like:

```
Sum: 25
Average: 5.0
Max: 9
```

Three habits to notice:

1. Start the running total at a sensible value (`0` for a sum). Start the "best so far" at the first item, not at 0 (a list of negative numbers would break a start of 0).
2. Guard against an empty array before dividing or reading `values[0]`.
3. `float(...)` appears before the division so we do not do integer division.

(Arrays have built-in `min()` and `max()`, but writing the loop once teaches you how it works, and the same pattern covers cases the built-ins do not, such as "the enemy with the lowest health".)

Counting and searching

```
func count_evens(values: Array[int]) -> int:
    var n: int = 0
    for v in values:
        if v % 2 == 0:
            n += 1
    return n

func index_of_first_over(values: Array[int], limit: int) -> int:
    for i in values.size():
        if values[i] > limit:
            return i
    return -1
```

With `data = [3, 8, 1, 9, 4]`, `count_evens(data)` is `2` and `index_of_first_over(data, 5)` is `1`. Returning `-1` for "not found" is a common convention because -1 is never a valid index.

Random numbers

Games depend on randomness: dice, loot, enemy choices. Godot gives you global functions:

Function	Result
`randi_range(a, b)`	a random **int** from `a` to `b`, both included

Function	Result
`randf_range(a, b)`	a random **float** from `a` up to `b`
`randi()`	a random 32-bit non-negative int
`randf()`	a random float from 0.0 to 1.0
`array.pick_random()`	one random item from an array
`array.shuffle()`	shuffles an array in place

```
func _ready() -> void:
    say("Dice: %d" % randi_range(1, 6))
    say("Chance: %.2f" % randf_range(0.0, 1.0))
    var greetings: Array[String] = ["Hi", "Hello", "Howdy"]
    say(greetings.pick_random())
```

What the output looks like (different each run):

```
Dice: 4
Chance: 0.37
Howdy
```

> **Note on `randomize()`:** In Godot 3 you had to call `randomize()` once at startup or you got the same "random" numbers every run. In Godot 4 the random generator is seeded automatically when the game starts, so you do **not** need to call it. Calling it again is harmless. To get the *same* sequence every time (useful for testing), call `seed(12345)` once.

If you need several independent random streams, create your own generator:

```
func _ready() -> void:
    var rng := RandomNumberGenerator.new()
    rng.seed = 12345
    say(rng.randi_range(1, 6))
    say(rng.randi_range(1, 6))
```

Because the seed is fixed, these two numbers are the same every time you run the project on the same Godot version.

To check that `randi_range` is fair, roll a die 600 times and count:

```
func _ready() -> void:
    var counts: Array[int] = [0, 0, 0, 0, 0, 0]
    for _i in 600:
        var roll: int = randi_range(1, 6)
        counts[roll - 1] += 1
    say(counts)
```

What the output looks like (each number is near 100, but never exactly the same twice):

```
[97, 104, 99, 108, 93, 99]
```

> **Try it:** Write `func roll_dice(count: int, sides: int) -> int` that rolls `count` dice with `sides` sides and returns the total. Print `roll_dice(2, 6)` five times.

4.7 Meet `LineEdit` and `Button`

The lab at the end of this chapter is a guessing game: the player types a number and presses a button. That needs two user-interface nodes. Both belong to the **Control** family, the nodes made for interfaces. Chapter 8 covers Controls in depth (anchors, containers, themes). For now you only need a few facts.

Node	What it is	Properties we use	Signals we use
`Label`	shows text	`text`	none
`LineEdit`	a one-line box where the player types	`text`, `placeholder_text`, `editable`; methods `clear()`, `grab_focus()`	`text_submitted(new_text)` when Enter is pressed
`Button`	a clickable button	`text`, `disabled`	`pressed` when clicked
`VBoxContainer`	stacks its children in a vertical column automatically	none	none

A **signal** is a message a node sends when something happens. `pressed` is sent by a Button when the player clicks it. To react, you **connect** the signal to one of your functions:

```
button.pressed.connect(_on_button_pressed)
```

It reads: "when `button` emits `pressed`, call `_on_button_pressed`." Chapter 5 explains signals fully; here, copy the pattern. The function must exist in the script:

```
func _on_button_pressed() -> void:
    print("Button clicked")
```

Try it in a scratch scene to see the pieces working together.

Scene tree:

```
Scratch   (Control)
└── Box   (VBoxContainer)
    ├── Echo      (Label)
    ├── Entry     (LineEdit)
    └── SayButton (Button)
```

Set `Scratch` and `Box` to Full Rect. Select `SayButton` and set its **Text** property to `Say it`. Select `Entry` and set **Placeholder Text** to `Type here`.

File: `res://scratch.gd`, attached to `Scratch` (Control).

```
extends Control

@onready var echo: Label = $Box/Echo
@onready var entry: LineEdit = $Box/Entry
```

```
@onready var say_button: Button = $Box/SayButton

func _ready() -> void:
    say_button.pressed.connect(_on_say_pressed)
    entry.text_submitted.connect(_on_text_submitted)

func _on_say_pressed() -> void:
    echo.text = "You typed: " + entry.text

func _on_text_submitted(new_text: String) -> void:
    echo.text = "You pressed Enter with: " + new_text
```

Run with F6, type `hello`, and click the button, then press Enter in the box.

What the output looks like:

```
You typed: hello
You pressed Enter with: hello
```

The `VBoxContainer` placed the three children in a column for us; we never set their positions. Notice that the paths in `$Box/Echo` follow the scene tree: `Box` is the child of the script's node, and `Echo` is the child of `Box`.

Lab 4: Number Guessing Game

Goal: Build a complete text game. The computer picks a secret number from 1 to 100. The player has 7 tries, and after each wrong guess the game says "higher" or "lower". The game must handle bad input, show the guess history, and allow a new game.

Time: 45-75 minutes.

Concepts used: `randi_range`, `if/elif/else`, `match`, `for`, functions with parameters and returns, typed arrays, `const`, and the Control nodes from 4.7.

Scene tree:

```
GuessGame   (Control)         <- script guess_game.gd
└── Box     (VBoxContainer)
    ├── Title          (Label)
    ├── Prompt         (Label)
    ├── GuessInput     (LineEdit)
    ├── GuessButton    (Button)
    ├── History        (Label)
    └── NewGameButton  (Button)
```

Steps

1. Create a new scene with a `Control` root named `GuessGame`. Set its Anchors Preset to **Full Rect**.
2. Add a `VBoxContainer` child named `Box`; set it to **Full Rect** as well. In the Inspector, under **Theme Overrides > Constants**, you may set **Separation** to `12` for breathing room.
3. Add the six children shown in the tree above as children of `Box`. Use the right node types: `Label` for `Title`, `Prompt`, and `History`; `LineEdit` for `GuessInput`; `Button` for `GuessButton` and `NewGameButton`.
4. Set properties in the Inspector: `Title` Text = `Guess the Number`; `GuessInput` Placeholder Text = `Type a whole number`; `GuessButton` Text = `Guess`; `NewGameButton` Text = `New game`.

5. For `Prompt` and `History`, set **Autowrap Mode** to **Word (Smart)** so long lines wrap inside the window.
6. Save the scene as `res://guess_game.tscn`.
7. Attach a script to `GuessGame`, saved as `res://guess_game.gd`, with the code below.
8. Run with F6 and play several rounds.

File: `res://guess_game.gd`, attached to `GuessGame` (Control).

```
extends Control

const MIN_NUMBER: int = 1
const MAX_NUMBER: int = 100
const MAX_TRIES: int = 7
const CLOSE_RANGE: int = 5

var secret: int = 0
var tries_left: int = 0
var game_over: bool = false
var guesses: Array[int] = []

@onready var prompt: Label = $Box/Prompt
@onready var guess_input: LineEdit = $Box/GuessInput
@onready var guess_button: Button = $Box/GuessButton
@onready var history: Label = $Box/History
@onready var new_game_button: Button = $Box/NewGameButton

func _ready() -> void:
    guess_button.pressed.connect(_on_guess_button_pressed)
    guess_input.text_submitted.connect(_on_guess_submitted)
    new_game_button.pressed.connect(start_game)
    start_game()

func start_game() -> void:
    secret = randi_range(MIN_NUMBER, MAX_NUMBER)
    tries_left = MAX_TRIES
    game_over = false
    guesses.clear()
    guess_input.editable = true
    guess_button.disabled = false
    guess_input.clear()
    guess_input.grab_focus()
    prompt.text = "I am thinking of a number from %d to %d. You have %d tries." % [MIN_NUMBER,
MAX_NUMBER, MAX_TRIES]
    update_history()

func _on_guess_button_pressed() -> void:
    submit_guess(guess_input.text)

func _on_guess_submitted(new_text: String) -> void:
    submit_guess(new_text)

func submit_guess(raw_text: String) -> void:
    if game_over:
        return
    var cleaned: String = raw_text.strip_edges()
    guess_input.clear()
    guess_input.grab_focus()

    if not cleaned.is_valid_int():
        prompt.text = "'%s' is not a whole number. Try again (no tries lost)." % cleaned
        return

    var guess: int = cleaned.to_int()
    if guess < MIN_NUMBER or guess > MAX_NUMBER:
        prompt.text = "Pick a number from %d to %d (no tries lost)." % [MIN_NUMBER, MAX_NUMBER]
        return
    if guess in guesses:
```

```
        prompt.text = "You already tried %d (no tries lost)." % guess
        return

    guesses.append(guess)
    tries_left -= 1

    if guess == secret:
        end_game(true)
    elif tries_left == 0:
        end_game(false)
    else:
        var hint: String = "Too low, go higher." if guess < secret else "Too high, go lower."
        if absi(guess - secret) <= CLOSE_RANGE:
            hint += " You are very close!"
        prompt.text = "%d: %s Tries left: %d" % [guess, hint, tries_left]
    update_history()

func end_game(won: bool) -> void:
    game_over = true
    guess_input.editable = false
    guess_button.disabled = true
    if won:
        prompt.text = "You got it: %d! %s" % [secret, rating_for(guesses.size())]
    else:
        prompt.text = "Out of tries. The number was %d. Press New game." % secret
    update_history()

func rating_for(tries_used: int) -> String:
    match tries_used:
        1:
            return "First try. Unbelievable!"
        2, 3:
            return "Excellent!"
        4, 5:
            return "Well done."
        _:
            return "Phew, just in time."

func update_history() -> void:
    if guesses.is_empty():
        history.text = "No guesses yet."
        return
    var text: String = "Guesses: "
    for g in guesses:
        text += str(g) + "  "
    history.text = text
```

Expected result

- On start the window shows the title, the prompt "I am thinking of a number from 1 to 100. You have 7 tries.", an empty text box (with focus, so you can type immediately), a Guess button, "No guesses yet.", and a New game button.
- Typing `50` and pressing Enter (or clicking Guess) shows something like `50: Too high, go lower. Tries left: 6` and the history shows `Guesses: 50`.
- Typing `abc`, an empty box, `0`, or `101` shows an explanation and does **not** use up a try.
- Typing the same number twice shows "You already tried ..." and does not use up a try.
- A guess within 5 of the secret adds "You are very close!".
- Finding the number ends the game with a rating from `rating_for`, and the input and Guess button are disabled.
- Running out of tries reveals the number.
- **New game** picks a fresh secret and re-enables everything.

A smart player finishes in 7 tries or fewer with a halving strategy: guess 50, then 25 or 75, and so on. (Halving a range of 100 needs at most 7 steps because 2 to the power of 7 is 128.)

Stretch goals

- Add a score kept across games: a `wins` and `losses` count in two more variables, shown in `Title`.
- Add a difficulty setting using three buttons (easy 1-50 with 8 tries, normal, hard 1-200 with 8 tries). Put the settings in a dictionary such as `{"easy": {"max": 50, "tries": 8}}` and read them in `start_game`.
- Use `seed(42)` temporarily to get the same secret each run while you test.

Chapter summary

- `if/elif/else` runs the first branch whose condition is true; guard clauses that return early keep code flat. `pass` marks an intentionally empty block.
- `match` compares one value against patterns (literals, several values, variables, arrays, dictionaries, `_`), has no fall-through, and cannot match ranges.
- `for` walks a range, array, dictionary (keys), or string; `while` repeats while a condition is true; `break` leaves a loop and `continue` skips to the next iteration.
- Functions take typed parameters (with optional defaults), return typed values with `return`, and see their own local variables; member variables are visible to the whole script.
- Lambdas are nameless functions stored in `Callable` values and called with `.call()`; arrays accept them in `map`, `filter`, `reduce`, `any`, `all`, and `sort_custom`.
- Arrays are ordered and indexed from 0; write `Array[int]` to make them typed; assignment and function arguments share the same array unless you call `duplicate()`.
- Dictionaries store key/value pairs; use `get(key, default)` for safe reads, and `in` to test whether a key (or an array item) exists.
- `randi_range(a, b)` includes both ends; Godot 4 seeds the generator for you, so `randomize()` is not needed; `seed()` makes results repeatable. `Label`, `LineEdit`, and `Button` are Control nodes; connect a Button's `pressed` signal to a function to react to clicks.

Review questions

1. In an `if/elif/else` chain, if two conditions are both true, how many branches run, and which one?
2. True or false: in a `match` statement you must write `break` at the end of each branch to stop it falling into the next one.
3. Write the numbers produced by `range(2, 10, 3)`.
4. What is the difference between `break` and `continue`?
5. What happens when a `for` loop iterates over a dictionary: does it give keys, values, or pairs?
6. What is the difference between a *parameter* and an *argument*?
7. What does the following print, and why? `var a: Array[int] = [1]; var b := a; b.append(2); print(a)`
8. Which is safer for reading a key that might not exist: `d["mana"]` or `d.get("mana", 0)`? Explain.
9. In Godot 4, do you need to call `randomize()` before using `randi_range()`? What do you call if you want the same sequence every run?
10. Multiple choice: Which expression gives a random whole number from 1 to 20, including both ends? (a) `randi_range(1, 20)` (b) `randi_range(0, 20)` (c) `randf_range(1, 20)` (d) `randi() % 20`.

Exercises

1. **Practice.** Write a function `is_even(n: int) -> bool` and a loop that prints every even number from 1 to 20 using it. Then change the loop to use `range` with a step instead, and compare the two.
2. **Practice.** Use a dictionary to store a shopping list with item names as keys and quantities as values. Add three items, change one quantity, remove one item, and print the final list with a `for` loop.
3. **Apply.** Write `func count_vowels(text: String) -> int`. Use `for` over the letters of the string and the `in` operator against the string `"aeiou"`. Convert the text to lowercase first. Test it with `"Weird Hotel"` (the answer is 4).
4. **Apply.** Write `func shuffle_and_deal(deck: Array[int], hand_size: int) -> Array[int]` that copies the deck, shuffles the copy, and returns the first `hand_size` cards. Verify that the original deck is not changed.
5. **Challenge.** Extend the guessing game so the computer plays against itself with a binary-search strategy: add a button "Auto-solve" that repeatedly guesses the middle of the remaining range (using variables `low` and `high`) and shows every guess in the history. Prove to yourself it never needs more than 7 guesses for any secret from 1 to 100.

Common mistakes

- **Symptom:** A `match` branch never runs although the value looks right. **Cause:** the types differ, for example matching the text `"5"` against the number `5`. **Fix:** convert first (`text.to_int()`) or match the correct type.
- **Symptom:** The game freezes and the window stops responding. **Cause:** a `while` loop whose condition never becomes false. **Fix:** stop the game with the Stop button (F8), and make sure the loop body changes the variable in the condition.
- **Symptom:** Error "Out of bounds get index" (arrays) or "Invalid access to property or key" (dictionaries). **Cause:** reading beyond the last array item, or a dictionary key that does not exist. **Fix:** check `size()` or `is_empty()` first, or use `get(key, default)` for dictionaries.
- **Symptom:** Changing a copy of an array also changes the original. **Cause:** assignment shares the same array. **Fix:** use `duplicate()` when you need an independent copy.
- **Symptom:** Items are skipped when removing during a `for` loop. **Cause:** the array shrinks while it is being walked. **Fix:** loop over `list.duplicate()`, or build a new array with `filter`.
- **Symptom:** A random number is off by one, or a range is never reached. **Cause:** `randi_range` includes both ends but `range()` excludes the end; mixing the two conventions causes confusion. **Fix:** write the intended bounds as constants and test the first and last value.

Further reading

- GDScript reference (control flow, functions, match): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
- Array class reference: https://docs.godotengine.org/en/stable/classes/class_array.html
- Dictionary class reference: https://docs.godotengine.org/en/stable/classes/class_dictionary.html
- Callable class reference: https://docs.godotengine.org/en/stable/classes/class_callable.html
- RandomNumberGenerator class reference: https://docs.godotengine.org/en/stable/classes/class_randomnumbergenerator.html
- LineEdit class reference: https://docs.godotengine.org/en/stable/classes/class_lineedit.html

Chapter 5: GDScript III: Classes, Signals, and Autoloads

Learning objectives

By the end of this chapter you will be able to:

- Explain why every script is a class, and use `class_name`, inner classes, inheritance, and `super`.
- Expose variables to the Inspector with `@export` (including ranges, enums, and groups) and initialize node references with `@onready`.
- Find other nodes with `$`, `%`, `get_node`, and `get_parent`, and name nodes so the paths do not break.
- Declare, emit, connect, and disconnect signals, both the built-in ones and your own.
- Use groups and `get_tree()` to talk to many nodes at once, and pause a function with `await` until a timer or a signal fires.
- Create an autoload (singleton) and apply the rule "call down, signal up" to keep scenes independent.

Key terms

- **Class**: a blueprint that describes the data (variables) and behavior (functions) of a kind of object.
- **Instance**: one object created from a class.
- **Inheritance**: building a class on top of another so it gets all of the parent's features.
- **Override**: replacing an inherited function with your own version.
- **Signal**: a message an object sends when something happens, which other objects can listen to.
- **Callable**: a value that stands for a function (and optionally the object it belongs to).
- **Emit**: to send a signal.
- **Group**: a text label you attach to nodes so you can find or command them all at once.
- **Coroutine**: a function that contains `await` and can pause and resume.
- **Autoload**: a script or scene Godot loads automatically at startup and keeps alive for the whole game (a singleton).
- **Singleton**: an object of which only one exists, reachable from anywhere by its name.
- **Composition**: building a thing from small independent parts instead of one big inheritance chain.

5.1 Every script is a class

In Chapter 3 you attached a script to a node and wrote `extends Control`. Behind that line is a bigger idea: a GDScript file is a **class**. A class is a blueprint. When Godot loads your scene, it creates one **instance** of the class for the node, and that instance has all the variables and functions you wrote, plus everything it inherited from `Control`.

> **Why it works:** `extends Control` says "my class inherits from `Control`." `Control` itself inherits from `CanvasItem`, which inherits from `Node`, which inherits from `Object`. Your script sits at the end of that chain, so a `Label` script can use `visible` (from `CanvasItem`), `name` (from `Node`), and `text` (from `Label`) all at once.

Naming a class: `class_name`

A script file has no name that other scripts can use, except its path. Add `class_name` at the top and the class gets a global name that you can use as a type anywhere in the project.

File: `res://animal.gd` (not attached to any node; it is a plain object).

```
class_name Animal
extends RefCounted

var animal_name: String
var sound: String = "..."

func _init(p_name: String = "Animal") -> void:
    animal_name = p_name

func speak() -> String:
    return "%s says %s" % [animal_name, sound]
```

Things to know:

- `RefCounted` is the simplest object type in Godot. Objects that extend it are freed automatically when nothing refers to them. Nodes are different: a node you create with `.new()` must be added to the scene with `add_child()` or freed with `queue_free()`.
- `_init` is the **constructor**. Godot runs it when you write `Animal.new(...)`. Its parameters become the arguments of `new`. The prefix `p_` is a common convention for parameters that would otherwise clash with a member's name.
- A class name must not clash with a Godot class (`Node`, `Timer`, `Tween`, `Input`) or with an autoload name. Pick specific names like `Animal`, `Lamp`, or `HealthComponent`.

Inheritance and `super`

Now make a more specific kind of animal.

File: `res://dog.gd` (plain object).

```
class_name Dog
extends Animal

func _init(p_name: String = "Dog") -> void:
    super(p_name)
    sound = "Woof"

func speak() -> String:
    return super() + "!"
```

`Dog extends Animal`, so a Dog has `animal_name`, `sound`, and `speak()` for free. It **overrides** `speak()` by defining a function with the same name. Inside, `super()` calls the *parent's* version of the current function, so we reuse its work and add to it. In `_init`, `super(p_name)` passes the name up to the parent constructor.

Try it in the Chapter 3 Playground scene:

```
func _ready() -> void:
    var pets: Array[Animal] = [Animal.new("Thing"), Dog.new("Rex")]
    for pet in pets:
        say(pet.speak())
    say(pets[1] is Dog)
```

```
    say(pets[0] is Dog)
    var rex := pets[1] as Dog
    say(rex.animal_name)
```

What the output looks like:

```
Thing says ...
Rex says Woof!
true
false
Rex
```

Notice that the loop treats both objects as `Animal`, and each one runs its own `speak`. That is the point of inheritance: code that works with the general type does not need to know the specific one. `is` tests the real type of an object; `as` converts it so the editor knows it can use `Dog`-specific members.

Inheritance with nodes and lifecycle functions

The same works for nodes. When you override a lifecycle function such as `_ready`, call `super()` first if the parent also uses it, or the parent's setup is skipped.

File: `res://enemy_base.gd`, attached to a `Node2D` named `EnemyBase` in your own scenes.

```
class_name EnemyBase
extends Node2D

@export var max_health: int = 10
var health: int = 0

func _ready() -> void:
    health = max_health

func take_damage(amount: int) -> void:
    health = maxi(health - amount, 0)
    if health == 0:
        die()

func die() -> void:
    queue_free()
```

File: `res://slime.gd`, attached to a `Node2D` named `Slime`.

```
class_name Slime
extends EnemyBase

func _ready() -> void:
    super()
    print("Slime ready with %d health" % health)

func die() -> void:
    print("Splat!")
    super()
```

To test, give the Playground scene a `Node2D` child named `Slime` with `slime.gd` attached, and use this in `playground.gd`:

```
func _ready() -> void:
    var slime := $Slime as Slime
    slime.take_damage(4)
```

```
    say(slime.health)
    slime.take_damage(20)
```

What the Output panel looks like:

```
Slime ready with 10 health
6
Splat!
```

(The Slime's `_ready` runs first because children are ready before their parent.) After `Splat!` the Slime frees itself.

> **Warning:** Deep inheritance trees become brittle. Inheritance says "is a kind of"; if a relationship is really "has a", use composition (section 5.7).

Inner classes

A small helper class that only one script uses can live inside that script with the `class` keyword. An inner class without `extends` inherits from `RefCounted`.

```
class Item:
    var item_name: String
    var weight: float

    func _init(p_name: String, p_weight: float) -> void:
        item_name = p_name
        weight = p_weight

    func describe() -> String:
        return "%s (%.1f kg)" % [item_name, weight]

func _ready() -> void:
    var bag: Array[Item] = []
    bag.append(Item.new("Key", 0.1))
    bag.append(Item.new("Bone", 0.5))
    for item in bag:
        say(item.describe())
```

Put the inner class at the left edge of `playground.gd`, outside any function. What the output looks like:

```
Key (0.1 kg)
Bone (0.5 kg)
```

Use an inner class for quick, private structures. For anything shared between scripts, give it its own file and a `class_name`.

> **Try it:** Create `Cat` (extends `Animal`) with the sound "Meow" and no exclamation mark. Add a `Cat.new("Tom")` to the pets array and check the output.

5.2 `@export` and `@onready`

An **annotation** is a word starting with `@` placed before a declaration to give it extra behavior.

`@export`: variables in the Inspector

`@export` makes a variable editable in the Inspector panel. Designers (and you) can tune the game without touching code, and each node can have a different value.

```
extends Control

enum Difficulty { EASY, NORMAL, HARD }

@export var player_name: String = "Mira"
@export_range(1, 99, 1) var level: int = 5
@export_range(0.0, 1.0, 0.05) var volume: float = 0.8
@export var tint: Color = Color.WHITE
@export var difficulty: Difficulty = Difficulty.NORMAL
@export_enum("Warrior", "Mage", "Rogue") var job: String = "Mage"
@export_multiline var bio: String = ""
@export var scene_to_spawn: PackedScene
@export var icon: Texture2D
@export var tags: Array[String] = []

@export_group("Movement")
@export var speed: float = 200.0
@export var jump_height: float = 64.0

func _ready() -> void:
    print("%s the %s, level %d, speed %.1f" % [player_name, job, level, speed])
```

Save the script, select the node in the Scene dock, and look at the Inspector: all those variables appear, with the right kind of editor for each type (a number box, a slider, a color picker, a drop-down, a file slot for the scene and texture).

What the Output panel shows when you run it with the defaults:

```
Mira the Mage, level 5, speed 200.0
```

Annotation	What it gives you
`@export var x: int`	a number field (works for most types: bool, float, String, Color, Vector2, arrays)
`@export_range(min, max, step)`	a slider/box limited to a range
`@export_enum("A", "B")`	a drop-down of choices (stored as an int, or a String when the variable is a String)
`@export var mode: MyEnum`	a drop-down built from an `enum` in the script
`@export_multiline`	a larger text box for a String
`@export_file("*.png")`	a String path chosen with a file picker
`@export var scene: PackedScene`	a slot to drop a scene file into
`@export var target: Node2D`	a slot to pick another node in the scene
`@export_group("Name")`	a collapsible heading in the Inspector for the exports that follow it
`@export_subgroup("Name")`	a smaller heading inside a group

Rules worth remembering:

- Always give an exported variable a **type** or a default value so Godot knows which editor to show.

- The value you set in the Inspector is saved in the scene file and **wins over** the default in the script. If you change a default in code later, nodes where you already edited the value keep their old value. Use the small revert arrow next to the property to return to the default.
- Exported values are applied before `_ready`, so you can use them there, but not inside `_init`.
- Scripts must be saved before new exports appear in the Inspector.

A setter can keep an exported value legal:

```
@export var max_health: int = 10
@export var health: int = 10:
    set(value):
        health = clampi(value, 0, max_health)
```

When code does `health = 50`, the setter runs and stores `10` instead. Inside the setter, writing `health = ...` assigns the stored variable directly; it does not call the setter again.

> **Note:** A setter on an exported variable also runs while the scene is loading, possibly before other variables exist. Keep setters short and safe.

`@onready`: wait until the children exist

Member variables are set up when the object is created, which is *before* it is in the scene tree and before its children exist. That is too early to look up a child.

```
# WRONG: runs too early, $Output does not exist yet.
var out: Label = $Output
```

`@onready` delays the assignment until just before `_ready` runs, when the whole subtree is available:

```
@onready var out: Label = $Output
```

You have been using this since Chapter 3; now you know why it is needed. Use `@onready` for any variable initialized from a node path.

> **Try it:** In your Playground script, add `@export var greeting: String = "Hello from the Inspector"`. In `_ready()`, call `say(greeting)`. Run once, then select `Playground`, change the Greeting in the Inspector, and run again. The line changes without editing the script.

5.3 Finding nodes

Scripts often need to reach other nodes. Godot gives you several ways, all based on the **scene tree** you built in Chapter 2.

Use this scene for the examples in this section.

```
Playground     (Control)  <- playground.gd
├── Output     (Label)
├── Menu       (VBoxContainer)
│   ├── StartButton  (Button)   Text = "Start"
```

```
│   └── QuitButton   (Button)   Text = "Quit"
└── Footer     (Label)    Text = "v1.0"
```

Right-click `Footer` in the Scene dock and choose **Access as Unique Name**. A green `%` appears beside the node. (Also set the two Button texts in the Inspector.)

```
extends Control

@onready var out: Label = $Output
@onready var start_button: Button = $Menu/StartButton
@onready var footer: Label = %Footer

func say(text: Variant) -> void:
    out.text += str(text) + "\n"

func _ready() -> void:
    say(start_button.text)
    say(get_node("Menu/QuitButton").text)
    say(footer.text)
    for child in $Menu.get_children():
        say(child.name)
    say(start_button.get_parent().name)
    say(find_child("QuitButton").name)
    say(get_node_or_null("Missing") == null)
    say(has_node("Menu"))
```

What the output looks like:

```
Start
Quit
v1.0
StartButton
QuitButton
Menu
QuitButton
true
true
```

The ways to reach a node

Syntax	Meaning
`$Menu/StartButton`	shortcut for `get_node("Menu/StartButton")` : a path **relative** to this node
`get_node("Menu/StartButton")`	the long form; useful when the path is built from a variable
`get_node("../Sibling")`	`..` goes up to the parent
`get_node("/root/Game")`	an **absolute** path from the top of the tree (this is how autoloads are found)
`%Footer`	a node marked as **unique name** in this scene; found no matter where it sits
`get_parent()`	the parent node
`get_children()`	an array of the direct children
`get_child(0)`	the child at that position
`find_child("Name")`	searches all descendants by name (slower; use sparingly)
`get_node_or_null(path)`	like `get_node`, but returns `null` instead of an error when missing
`has_node(path)`	`true` if the path exists

The shorthand `$Name` and `%Name` are just text sugar for `get_node`. If a node name contains spaces, write `$"My Node"`.

Why unique names matter

Paths break when you reorganize the scene. If you put `Footer` inside a new `Panel`, the path `$Footer` is wrong and the game errors with "Node not found". A unique name (`%Footer`) survives such moves because Godot finds it by name inside the scene, wherever it is. Use `%` for important nodes that scripts depend on, especially in Labs like the one at the end of this chapter.

Limits of `%`: it only finds nodes in the **same scene file** as the script's scene root. Nodes inside an instanced child scene have their own unique-name space.

Typed node variables

Always write the node type on `@onready` variables: `@onready var start_button: Button = $Menu/StartButton`. The editor then offers `Button`'s properties after the dot and flags typos. If the node at the path is not actually a Button, the game stops with an error that points at the line.

> **Warning:** If you rename a node, every `$Path` that mentions the old name breaks. Godot does not update strings in scripts. When a "Node not found" error appears, compare the path to the Scene dock character by character, including capital letters.

5.4 Signals

How should the pieces of a game talk to each other? A Button should not have to know which function in which script wants to react when it is pressed. Godot's answer is the **signal**: a node announces "this happened" and anyone interested listens.

Think of a doorbell. The button does not know who is home. It rings, and whoever is listening responds.

Built-in signals

Every node type comes with signals. Some you will meet constantly:

Node	Signal	When
`Button`	`pressed`	the player clicks it
`Button`	`toggled(toggled_on)`	a toggle button changes state
`Timer`	`timeout`	the countdown reaches zero
`LineEdit`	`text_submitted(new_text)`	Enter pressed
`LineEdit`	`text_changed(new_text)`	every edit
`Area2D`	`body_entered(body)`	a physics body enters (Chapter 7)
`AnimationPlayer`	`animation_finished(anim_name)`	an animation ends (Chapter 9)

Look up a node's signals in the editor: select it and open the **Node** dock (next to the Inspector), **Signals** tab. In the script editor, hover over a class name and open its documentation to see its Signals section.

Connecting in the editor

1. Select the node that emits (for example `StartButton`).
2. Open the **Node** dock > **Signals** tab.
3. Double-click `pressed()`.
4. Choose the receiving node and the method name (Godot suggests `_on_start_button_pressed`), then click **Connect**.

Godot writes an empty function in the receiver's script, and shows a green icon next to the function and in the Signals tab. Fill it in:

```
func _on_start_button_pressed() -> void:
    say("Start was pressed")
```

The naming convention is `_on_<NodeName>_<signal_name>`.

Connecting in code

Code connections are easier to read, search, and review, and they work for nodes you create while the game runs. The pattern is `emitter.signal_name.connect(function)`. The function is given as a **Callable**: just its name, with no parentheses.

```
func _ready() -> void:
    start_button.pressed.connect(_on_start_pressed)

func _on_start_pressed() -> void:
    say("Start was pressed")
```

Notice `_on_start_pressed` has no `()`. With parentheses you would *call* the function and connect its result. Without them, you hand over the function itself.

Here is a timer, created entirely in code:

```
var timer: Timer
var ticks: int = 0

func _ready() -> void:
    timer = Timer.new()
    timer.wait_time = 1.0
    timer.timeout.connect(_on_tick)
    add_child(timer)
    timer.start()

func _on_tick() -> void:
    ticks += 1
    say("tick %d" % ticks)
    if ticks == 3:
        timer.stop()
        say("stopped")
```

What the output looks like (one line per second):

```
tick 1
tick 2
tick 3
stopped
```

(A node created with `.new()` is not part of the game until `add_child` puts it in the tree. Starting the timer before that would fail.)

Your own signals

Declare a custom signal with `signal`, giving the name and, optionally, typed parameters. Send it with `.emit(...)`.

```
class Door:
    signal opened(by: String)
    var is_open: bool = false

    func open(by: String) -> void:
        if is_open:
            return
        is_open = true
        opened.emit(by)

func _on_door_opened(by: String) -> void:
    say("Door opened by " + by)

func _on_first_open(_by: String) -> void:
    say("(first open only)")

func _ready() -> void:
    var door := Door.new()
    door.opened.connect(_on_door_opened)
    door.opened.connect(_on_first_open, CONNECT_ONE_SHOT)
    door.open("Mira")
    door.is_open = false
    door.open("Bo")
```

What the output looks like:

```
Door opened by Mira
(first open only)
Door opened by Bo
```

What happened:

- `signal opened(by: String)` declares that doors can announce an opening and that they say who did it.
- `opened.emit(by)` sends the message. Every connected function is called, in the order they were connected, with the same arguments.
- `CONNECT_ONE_SHOT` is a flag that automatically disconnects a listener after its first call. That is why `(first open only)` appears once.
- The Door does not know who is listening. It could have zero listeners or ten.

Naming advice: name signals for what *happened*, in the past tense or as an event: `opened`, `died`, `health_changed`, `toggled`. Never name them after what the listener should do.

Parameters must match

The receiving function must accept the same number of parameters the signal sends. If `opened` sends one argument (`by`), a function with no parameters raises an error when the signal is emitted. If you need extra information, **bind** it:

```gdscript
func _on_door_opened_at(by: String, place: String) -> void:
    say("%s opened the door in the %s" % [by, place])

func _ready() -> void:
    var door := Door.new()
    door.opened.connect(_on_door_opened_at.bind("hallway"))
    door.open("Mira")
```

`bind` attaches extra arguments that are added **after** the ones the signal sends. The output is `Mira opened the door in the hallway`.

You can also connect a lambda (Chapter 4) for tiny reactions:

```gdscript
func _ready() -> void:
    var door := Door.new()
    door.opened.connect(func(by: String): say(by + " came in"))
    door.open("Bo")
```

Disconnecting and checking

```gdscript
func _ready() -> void:
    var door := Door.new()
    door.opened.connect(_on_door_opened)
    say(door.opened.is_connected(_on_door_opened))
    door.opened.disconnect(_on_door_opened)
    say(door.opened.is_connected(_on_door_opened))
```

What the output looks like:

```
true
false
```

Connecting the same function twice, or disconnecting one that is not connected, raises an error. Guard it with `is_connected`:

```gdscript
if not door.opened.is_connected(_on_door_opened):
    door.opened.connect(_on_door_opened)
```

Godot automatically removes connections when either end is freed, so you usually do not need to disconnect by hand.

> **Note:** Older tutorials use `emit_signal("opened", by)` and `connect("opened", self, "_on_opened")`. The first still works in Godot 4, but the `signal.emit()` and `signal.connect()` style shown here is shorter, checked by the editor, and recommended. The form `connect("opened", self, "_on_opened")`, with the target object and the method name as text, is Godot 3 syntax and does not work in Godot 4.
>
> **Try it:** Add a `closed` signal to `Door` and a `close(by)` method. Connect both signals to functions that say what happened, then open and close the door with two different names.

5.5 Groups and `get_tree()`

Groups

A **group** is a text tag. You can put any number of nodes in a group and then ask the engine to find them all or call a method on all of them. This is perfect for "all enemies", "all lamps", or "everything that can be saved".

Add a node to a group in the editor (Node dock > **Groups** tab, type a name, **Add**) or in code:

```
func _ready() -> void:
    for i in 3:
        var lamp := Label.new()
        lamp.text = "Lamp %d" % i
        lamp.position = Vector2(20, 80 + i * 30)
        lamp.add_to_group("lamps")
        add_child(lamp)

    say(get_tree().get_nodes_in_group("lamps").size())
    say(get_child(get_child_count() - 1).is_in_group("lamps"))
    get_tree().call_group("lamps", "hide")
    await get_tree().create_timer(1.0).timeout
    get_tree().call_group("lamps", "show")
```

What happens: three labels appear below the output text. The output text shows

```
3
true
```

then the three labels vanish, and a second later they reappear.

The group tools:

Call	Meaning
`node.add_to_group("name")`	join a group
`node.remove_from_group("name")`	leave a group
`node.is_in_group("name")`	membership test
`get_tree().get_nodes_in_group("name")`	array of all nodes in the group
`get_tree().call_group("name", "method", args...)`	call a method on every member

`call_group` uses the method name as text, so a typo is not caught by the editor. Keep group names in `const` variables if you use them in many places.

`get_tree()`: the whole game

Every node can call `get_tree()` to get the **SceneTree**, the object that runs the game. It is the control panel for everything beyond one node.

Call	What it does
`get_tree().quit()`	close the game
`get_tree().reload_current_scene()`	restart the running scene

Call	What it does
`get_tree().change_scene_to_file("res://menu.tscn")`	switch to another scene
`get_tree().paused = true`	pause the game (nodes that are not set to run while paused freeze)
`get_tree().create_timer(2.0)`	make a one-use timer; has a `timeout` signal
`get_tree().current_scene`	the root node of the running scene
`get_tree().root`	the very top node of the tree (the Window)

`get_tree()` is only valid for a node that is **in** the tree. Calling it on a node you just made with `.new()` and have not yet added gives an error.

5.6 `await`: waiting without freezing

Games constantly say "do this, wait a moment, then do that." A `while` loop cannot wait without freezing the game (Chapter 4). `await` can. It pauses *this function only*, lets the rest of the game keep running, and resumes the function when something happens.

Awaiting a timer

```
func _ready() -> void:
    say("Starting countdown")
    for i in range(3, 0, -1):
        say(i)
        await get_tree().create_timer(1.0).timeout
    say("Liftoff!")
```

What the output looks like, one line per second:

```
Starting countdown
3
2
1
Liftoff!
```

`get_tree().create_timer(1.0)` makes a timer object, and `.timeout` is its signal. `await` followed by a signal means "stop here until that signal is emitted, then continue on the next line."

Other things you can await:

Expression	Resumes when
`await get_tree().create_timer(2.0).timeout`	2 seconds passed
`await get_tree().process_frame`	the next frame
`await some_button.pressed`	the button is pressed
`await some_node.some_signal`	any signal fires
`await my_coroutine()`	another function that awaits has finished

Awaiting a signal that carries values

If the signal has one parameter, `await` hands that value back. If it has several, you get an array.

```
signal damage_taken(amount: int)
signal hit(who: String, amount: int)

func fire_later() -> void:
    await get_tree().create_timer(0.5).timeout
    damage_taken.emit(5)
    await get_tree().create_timer(0.2).timeout
    hit.emit("Bo", 7)

func _ready() -> void:
    fire_later()
    var amount: int = await damage_taken
    say("Took %d damage" % amount)
    var result: Array = await hit
    say("%s was hit for %d" % [result[0], result[1]])
```

Because `fire_later()` awaits inside, calling it without `await` starts it running and returns immediately. `_ready` then waits for `damage_taken`.

What the output looks like (the first line after half a second, the second a little later):

```
Took 5 damage
Bo was hit for 7
```

> **Warning:** You cannot use `:=` with `await` (`var x := await sig` fails) because the type of the result is not known in advance. Write the type: `var amount: int = await damage_taken`.

Coroutines and return values

A function with `await` inside is a **coroutine**. Callers can wait for it, and it can return a value:

```
func wait_and_double(x: int) -> int:
    await get_tree().create_timer(0.5).timeout
    return x * 2

func _ready() -> void:
    var result: int = await wait_and_double(21)
    say(result)
```

The result `42` appears after half a second. Without `await` in front of the call, the line after it would run before the result exists.

Things to watch for

- If the signal you wait for never fires, the function stays paused forever. That is not a crash, but whatever came after is never done.
- If the node that is awaiting is freed (for example, the scene changes), the coroutine ends quietly and the rest of the function is skipped.
- A function paused in `await` still has its local variables. If other code changes the game while it waits, check the situation again after resuming.

> **Try it:** Make a "blinking" label: in a loop of 5 iterations, set `out.visible = false`, await 0.3 seconds, set `out.visible = true`, await 0.3 seconds.

5.7 Autoloads, call down and signal up, and composition

Autoloads: objects that live for the whole game

Normally a node exists only while its scene is loaded. Some things must outlive scenes: the player's score, game settings, a music player. An **autoload** is a script (or scene) that Godot loads when the game starts and keeps in the scene tree, as a child of the root, for the entire run. It is available by name from every script. This pattern is called a **singleton**.

Create one:

1. Create a script `res://stats.gd` (FileSystem dock: right-click > **Create New > Script...**; the default template is fine).
2. Open **Project > Project Settings** and click the **Globals** tab, then **Autoload**.
3. In **Path**, click the folder icon and pick `res://stats.gd`.
4. In **Node Name**, type `Stats` (PascalCase; this is the name you will use in code).
5. Click **Add**. In the list below, check that the **Global Variable** box next to `Stats` is ticked (it is by default); that is what lets you write `Stats` in code.

File: `res://stats.gd` (autoload named `Stats`).

```
extends Node

signal coins_changed(total: int)

var coins: int = 0

func add_coins(amount: int) -> void:
    coins += amount
    coins_changed.emit(coins)
```

Now any script in the project can use it:

```
func _ready() -> void:
    Stats.coins_changed.connect(func(total: int): say("Coins: %d" % total))
    Stats.add_coins(3)
    Stats.add_coins(2)
    say(Stats.coins)
```

What the output looks like:

```
Coins: 3
Coins: 5
5
```

When the game runs, the scene tree looks like this:

```
root (Window)
├── Stats        <- autoload, added before your scene
```

```
└── Playground   <- your scene
```

Autoload facts:

- Autoloads load in the order shown in the list and stay loaded when you change scenes. Their variables keep their values.
- They also exist when you run a single scene with F6.
- An autoload name must not be the same as a `class_name` or a Godot class.
- If you type the name wrongly, the editor says "Identifier not declared in the current scope."
- An autoload can also be a whole scene (`.tscn`), for example a pause menu that should always exist.

> **Warning:** Autoloads are global, and global state is easy to abuse. Every script can change `Stats.coins`, so bugs become hard to trace. Use autoloads for truly global services: game state, settings, audio, scene switching. Everything else should stay inside its scene.

Call down, signal up

A scene is more useful the less it knows about the world around it. A guideline keeps scenes independent:

> **Call down, signal up.** A parent may call functions and read properties on its **children**, because it created or owns them. A child must **not** reach upward to its parent. Instead it emits a signal and lets whoever is above decide what to do.

```
      Main                Main knows its children,
     /    \               connects their signals,
  Switch   Lamp            and calls their functions.
                           Switch and Lamp know nothing of Main or each other.
  calls go DOWN  ↓         signals go UP  ↑
```

Why bother? A `Switch` that does `get_parent().get_node("Lamp").turn_on()` only works inside one particular scene layout. A `Switch` that emits `toggled(on)` works anywhere: with one lamp, ten lamps, or none at all.

Rules of thumb:

1. Never write `get_parent().get_parent()...`. A long chain breaks as soon as the layout changes.
2. Siblings do not talk to each other directly. Their shared parent connects them.
3. For communication across the whole game (the player died, the level ended), use an autoload that only holds signals. This is called an **event bus**.
4. Parents hold typed references to children (`@onready` or `@export`) and call down.

Composition instead of deep inheritance

Suppose enemies, crates, and the player all need health. With inheritance you would force them all under one base class. With **composition** you build the behavior as a small separate node and add it wherever it is needed.

File: `res://health_component.gd`, attached to a `Node` named `HealthComponent`.

```gdscript
class_name HealthComponent
extends Node
```

```gdscript
signal changed(current: int, maximum: int)
signal died

@export var max_health: int = 10

var current: int = 0

func _ready() -> void:
    current = max_health

func damage(amount: int) -> void:
    current = maxi(current - amount, 0)
    changed.emit(current, max_health)
    if current == 0:
        died.emit()
```

Any scene can have a `HealthComponent` child. The parent listens to it:

```gdscript
func _ready() -> void:
    var hc := HealthComponent.new()
    hc.max_health = 5
    add_child(hc)
    hc.changed.connect(func(cur: int, mx: int): say("Health %d/%d" % [cur, mx]))
    hc.died.connect(func(): say("It died!"))
    hc.damage(3)
    hc.damage(3)
```

What the output looks like:

```
Health 2/5
Health 0/5
It died!
```

The component knows nothing about who owns it. A crate can connect `died` to "spawn loot", a player to "show game over". In Chapter 10 you will meet **Resources**, objects that store shared data; for now, composition with small nodes plus signals is all you need.

> **Why it works:** Small single-purpose nodes are easy to test, reuse, and replace. Because they communicate by signals, adding a new behavior never requires editing the old ones.

Lab 5: The Light Switch

Goal: Build a tiny system where a Switch node emits `toggled(on)`, two Lamps listen and light up, and a `Game` autoload counts how many times the switch was flipped. The Switch and the Lamp must know nothing about each other.

Time: 45-75 minutes.

Concepts used: classes and `class_name`, `@export`, `@onready`, unique names, custom signals, connecting with Callables, autoloads, call down and signal up.

Final scene tree:

```
Main            (Control)          <- main.gd
├── LightSwitch (Node)             <- light_switch.gd   (unique name)
└── Layout      (VBoxContainer)
```

```
    ├── Title        (Label)
    ├── Lamps        (HBoxContainer)
    │   ├── LampA    (ColorRect)   <- lamp.gd          (unique name)
    │   └── LampB    (ColorRect)   <- lamp.gd          (unique name)
    ├── FlipButton   (Button)                          (unique name)
    ├── ResetButton  (Button)                          (unique name)
    └── CountLabel   (Label)                           (unique name)
```

Part 1: the autoload (do this first)

The Switch script refers to `Game`; if the autoload does not exist yet the editor shows an error.

1. Create `res://game.gd` with the code below.
2. **Project > Project Settings > Globals > Autoload**: Path `res://game.gd`, Node Name `Game`, click **Add**.

File: `res://game.gd` (autoload named `Game`).

```gdscript
extends Node

signal count_changed(count: int)

var toggle_count: int = 0

func record_toggle() -> void:
    toggle_count += 1
    count_changed.emit(toggle_count)

func reset() -> void:
    toggle_count = 0
    count_changed.emit(toggle_count)
```

Part 2: the Switch and the Lamp

3. Create `res://light_switch.gd`.

File: `res://light_switch.gd`, attached to the `LightSwitch` (Node).

```gdscript
class_name LightSwitch
extends Node

signal toggled(on: bool)

@export var start_on: bool = false

var is_on: bool = false

func _ready() -> void:
    is_on = start_on

func flip() -> void:
    set_state(not is_on)

func set_state(on: bool) -> void:
    if on == is_on:
        return
    is_on = on
    Game.record_toggle()
    toggled.emit(is_on)
```

4. Create `res://lamp.gd`.

File: `res://lamp.gd`, attached to both `LampA` and `LampB` (ColorRect).

```
class_name Lamp
extends ColorRect

@export var on_color: Color = Color.YELLOW
@export var off_color: Color = Color(0.2, 0.2, 0.2)
@export var lamp_size: Vector2 = Vector2(120, 120)

var is_lit: bool = false

func _ready() -> void:
    add_to_group("lamps")
    custom_minimum_size = lamp_size
    set_on(false)

func set_on(on: bool) -> void:
    is_lit = on
    color = on_color if on else off_color
```

> **Note:** Our switch script is named `LightSwitch` and its signal `toggled`. A `Button` already has a signal called `toggled`, so we make the switch a plain `Node`, not a Button. If you extend a class and declare a signal with a name the parent already has, Godot reports an error.

Part 3: build the scene

5. New scene: root `Control` named `Main`, anchors **Full Rect**. Save it as `res://main.tscn`.
6. Add a `Node` child named `LightSwitch` and attach `light_switch.gd`.
7. Add a `VBoxContainer` child named `Layout`, **Full Rect**. Give it children: `Title` (Label, Text `Light Switch`), `Lamps` (HBoxContainer), `FlipButton` (Button, Text `Flip`), `ResetButton` (Button, Text `Reset`), `CountLabel` (Label).
8. Add two `ColorRect` children to `Lamps` named `LampA` and `LampB`. Attach `lamp.gd` to each. Select `LampB` and in the Inspector set **On Color** to a cyan or green so you can tell the lamps apart. This shows how `@export` lets two instances of one script differ.
9. Make the important nodes reachable by unique name: right-click each of `LightSwitch`, `LampA`, `LampB`, `FlipButton`, `ResetButton`, and `CountLabel`, and choose **Access as Unique Name**.
10. Attach a script to `Main`, saved as `res://main.gd`.

File: `res://main.gd`, attached to `Main` (Control).

```
extends Control

@onready var light_switch: LightSwitch = %LightSwitch
@onready var lamp_a: Lamp = %LampA
@onready var lamp_b: Lamp = %LampB
@onready var flip_button: Button = %FlipButton
@onready var reset_button: Button = %ResetButton
@onready var count_label: Label = %CountLabel

func _ready() -> void:
    light_switch.toggled.connect(lamp_a.set_on)
    light_switch.toggled.connect(lamp_b.set_on)
    flip_button.pressed.connect(light_switch.flip)
    reset_button.pressed.connect(_on_reset_pressed)
    Game.count_changed.connect(_on_count_changed)

    lamp_a.set_on(light_switch.is_on)
    lamp_b.set_on(light_switch.is_on)
```

```
    Game.reset()

func _on_count_changed(count: int) -> void:
    count_label.text = "Switch flips: %d" % count

func _on_reset_pressed() -> void:
    light_switch.set_state(false)
    Game.reset()
```

11. Run the scene with **F6**.

How the pieces fit

- `Main` is the only script that knows all the parts. It **calls down** (`light_switch.flip`, `lamp.set_on`) and connects the signals.
- `LightSwitch` emits `toggled(on)` and knows nothing about lamps or buttons. (It does call the global `Game`, which is a service, not a sibling.)
- `Lamp.set_on(on)` has exactly the shape of the signal `toggled(on: bool)`, so the lamp function can be connected directly.
- `Game` counts and announces with its own signal `count_changed`; `Main` shows the number.

Expected result

- Two dark gray squares side by side, a Flip button, a Reset button, and the text `Switch flips: 0`.
- Click **Flip**: LampA turns yellow and LampB turns its own color, and the label reads `Switch flips: 1`.
- Click **Flip** again: both go dark and the count is 2.
- Click **Flip** once, then **Reset**: both lamps are dark and the count returns to 0.
- Select `LightSwitch` and tick **Start On** in the Inspector, then run: the lamps start lit, and the count is 0.

Stretch goals

Third lamp. Duplicate `LampB` with Ctrl+D, rename it `LampC`, make it a unique name, and connect it in `main.gd`. Notice that you changed no code in `lamp.gd` or `light_switch.gd`.

Use a group. Replace the two direct lamp connections with a single listener in `Main`. Add this function to `main.gd`:

```
func _on_switch_toggled(on: bool) -> void:
    get_tree().call_group("lamps", "set_on", on)
```

Connect it with `light_switch.toggled.connect(_on_switch_toggled)` in `_ready`, and change the two startup `set_on` lines to `get_tree().call_group("lamps", "set_on", light_switch.is_on)`. Now any number of lamps in the group react without `Main` holding a reference to each.

Flicker with `await`. In `lamp.gd`, replace `set_on` with the version below and add `flicker`. The lamp blinks briefly when it turns on.

```
func set_on(on: bool) -> void:
    is_lit = on
    if on:
        flicker()
    else:
        color = off_color
```

```
func flicker() -> void:
    color = off_color
    await get_tree().create_timer(0.08).timeout
    if is_lit:
        color = on_color
```

The `if is_lit` check after the wait matters: if the player flips the switch off during the flicker, the lamp must not turn back on.

Chapter summary

- Every script is a class: `extends` names its parent, `class_name` gives it a global name, and `.new()` makes an instance.
- Inheritance lets a class reuse another's code; override functions by redefining them and use `super()` to include the parent's version. Inner classes (`class Name:`) hold small private helpers.
- `@export` puts variables in the Inspector (with ranges, enums, groups), and `@onready` delays a node lookup until the children exist.
- Reach nodes with `$Path`, `%UniqueName`, `get_node`, and `get_parent`; unique names survive scene reorganizing.
- Signals are announcements: declare with `signal`, send with `.emit()`, listen with `.connect(callable)`; bind extra arguments with `bind`, use `CONNECT_ONE_SHOT` for single use, and `disconnect` when needed.
- Groups tag nodes so `get_tree().get_nodes_in_group()` and `call_group()` can reach them all; `get_tree()` also quits, reloads, and changes scenes.
- `await` pauses a function until a timer or signal fires without freezing the game; a signal's values come back as the result.
- Autoloads are always-loaded singletons for truly global state; follow "call down, signal up" and prefer small composed nodes (like `HealthComponent`) over deep inheritance.

Review questions

1. What does `class_name Lamp` do, and why must the name not be `Timer`?
2. In a script that overrides `_ready()` from a parent class, why would you call `super()` and where?
3. What is the difference between `@export var speed: float = 100.0` and a plain `var speed: float = 100.0`?
4. Why does `var out: Label = $Output` (without `@onready`) fail at the top of a script?
5. Which node does `%Footer` find, and why is it more robust than `$Footer`?
6. Write the line that connects a button's `pressed` signal to a function named `_on_play_pressed`. Why are there no parentheses after the function name?
7. A signal is declared as `signal hit(who: String, amount: int)`. What does `await hit` return?
8. True or false: it is fine for a child node to call `get_parent().get_parent().add_score(5)`. Explain.
9. What is the difference between `get_tree().get_nodes_in_group("enemies")` and `get_tree().call_group("enemies", "flash")`?
10. Name two good uses and one bad use of an autoload.

Exercises

1. **Practice.** Write a `Counter` class with `class_name Counter extends RefCounted`. It has a `signal changed(value: int)`, a `value` variable, and `increment()`. In the Playground, connect a lambda that says the new value and call `increment()` three times.
2. **Practice.** Create a scene with a `Timer` node and a `Label`. Using the Node dock (not code), connect `timeout` to a function that increments a number and shows it. Set the Timer's Autostart property so it begins on its own.
3. **Apply.** Build a `Door` scene (a `ColorRect` that changes color) with an exported `locked: bool` and a method `try_open(has_key: bool)`. It emits `opened` when it succeeds and `refused` when it does not. In the parent scene connect both to a Label that reports the result. Use only "call down, signal up".
4. **Apply.** Write a `fade_out(node: Control, seconds: float)` coroutine that, using `await get_tree().process_frame` in a `while` loop and `delta` (`get_process_delta_time()`), lowers `node.modulate.a` from 1 to 0 over the given time. Check that you can use `await fade_out(...)` in another function.
5. **Challenge.** Create an event-bus autoload `Events` with signals `player_died` and `coin_collected(amount: int)`. Build two unrelated scenes: one emits both signals from buttons, the other displays a coin total and a "Game over" label. Show that the two scenes work together without referring to each other. Write two sentences about what could go wrong if every signal in a large game went through the bus.

Common mistakes

- **Symptom:** "Invalid access to property or key on a base object of type 'Nil'" in `_ready`. **Cause:** the variable holding a node is `null` because the path is wrong, or `@onready` is missing. **Fix:** check the node name in the Scene dock and use `@onready`.
- **Symptom:** "Node not found: ..." after you reorganize a scene. **Cause:** the `$Path` no longer matches. **Fix:** use unique names (`%Name`) for nodes scripts depend on.
- **Symptom:** A signal seems connected, yet the function never runs. **Cause:** the signal is never emitted (for example misspelled name, or `emit` missing), or the connect line ran in a different object than you thought. **Fix:** add a `print` before `emit`, and check the green connection icons in the Signals tab. The same two causes explain code after an `await` that never runs: the awaited signal never fires, or the awaiting node was freed.
- **Symptom:** Error mentioning the number of arguments when the signal is emitted. **Cause:** the receiving function has a different number of parameters than the signal sends. **Fix:** add the missing parameters (use `_name` if unused), or use `bind` / a lambda.
- **Symptom:** "Signal 'x' is already connected" error. **Cause:** the connect code runs more than once (for example in a function called repeatedly). **Fix:** connect once in `_ready`, or guard with `is_connected`.
- **Symptom:** "Identifier 'Game' not declared in the current scope". **Cause:** the autoload is missing, misspelled, or has a different Node Name than you typed. **Fix:** open Project Settings > Globals > Autoload and compare the name exactly.

Further reading

- Scripting languages overview and GDScript basics (classes, inheritance, signals, await): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

- GDScript exported properties: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_exports.html
- Using signals: https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html
- Nodes and scene instances (including unique names): https://docs.godotengine.org/en/stable/tutorials/scripting/nodes_and_scene_instances.html
- Groups: https://docs.godotengine.org/en/stable/tutorials/scripting/groups.html
- Singletons (autoload): https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html
- Scene organization (call down, signal up): https://docs.godotengine.org/en/stable/tutorials/best_practices/scene_organization.html

Part II: Making 2D Games

Chapter 6: 2D Basics and Input

Learning objectives

By the end of this chapter you will be able to:

- Describe Godot's 2D coordinate system and convert between local and global positions.
- Change a `Node2D`'s position, rotation, and scale from the Inspector and from code, and predict how a parent's transform affects its children.
- Show an image with `Sprite2D`, including sharp pixel-art filtering, regions, sprite sheets, and flipping.
- Build placeholder art from built-in nodes (`Polygon2D`, `ColorRect`, `_draw()`) without any image files.
- Move things at the same speed on every computer by using `delta` in `_process`.
- Define actions in the Input Map and read them with `Input.is_action_pressed`, `is_action_just_pressed`, and `Input.get_vector`.
- Use basic `Vector2` math (`length`, `normalized`, `distance_to`, `direction_to`, `lerp`, `clamp`) to aim, steer, and follow.

Key terms

- **Coordinate system**: the rule that says where (0, 0) is and which way the numbers grow. In Godot 2D, x grows to the right and y grows *down*.
- **Transform**: the position, rotation, and scale of a node, taken together.
- **Local vs. global**: local values are measured relative to the parent; global values are measured relative to the whole world.
- **Radian**: the unit Godot uses for angles in code. A half turn is `PI` radians (about 3.14159).
- **Texture**: an image that the GPU can draw. `Sprite2D` shows a texture.
- **Sprite sheet**: one image that holds many small pictures (frames) in a grid.
- **Delta**: the number of seconds since the previous frame. Multiplying by it makes movement independent of frame rate.
- **Action**: a named input such as `move_left`, mapped to keys, buttons, or sticks in the Input Map.
- **Input Map**: the table in Project Settings that connects physical inputs to action names.
- **Vector2**: a pair of numbers (x, y). It can mean a point, a direction, or a speed.
- **Normalized vector**: a vector scaled to length 1, so it keeps only the direction.
- **Camera2D**: a node that decides which part of the 2D world appears on screen.

6.1 The 2D world: coordinates and Node2D

Where is (0, 0)?

Every 2D game needs a way to say "where". Godot uses the same system as most screen-based software:

```
(0,0) ------------> +x
  |
  |
  v
 +y
```

The origin (0, 0) is the **top-left** corner of the screen (for the default view). The x value grows to the right. The y value grows **downward**. This surprises many beginners, because school graphs put positive y at the top.

> **Why it works:** Screens are drawn row by row starting at the top, so computers count rows downward. Godot follows that habit. Remember it with this rule: *up is negative y*.

The default game window is 1152 by 648 pixels. So the bottom-right corner is about (1152, 648). You can change the size in **Project > Project Settings > Display > Window**.

Node2D: the base of all 2D nodes

In Chapter 2 you learned that nodes form a tree. In 2D, almost everything you can see or move inherits from `Node2D`. A `Node2D` itself draws nothing. It only carries a **transform**, which has three parts:

Property	Type	Meaning
`position`	`Vector2`	Where the node is, relative to its parent
`rotation`	`float`	Turn angle in **radians**, clockwise on screen
`scale`	`Vector2`	Size multiplier on x and y (1, 1 is normal)

Because y points down, a *positive* rotation turns things **clockwise** as you look at the screen. That is the opposite of the math class convention, and it follows directly from y pointing down.

You can edit these in the Inspector under **Transform**. In the Inspector, rotation is shown in **degrees** (it is friendlier), but in code the property `rotation` is in **radians**. There is a second property, `rotation_degrees`, that uses degrees in code.

```
rotation = PI / 2.0            # a quarter turn, in radians
rotation_degrees = 90.0        # the same turn, in degrees
rotation = deg_to_rad(90.0)    # convert degrees to radians
var d: float = rad_to_deg(rotation)  # and back
```

Degrees	Radians	In code
0	0	`0.0`
90	about 1.571	`PI / 2.0`
180	about 3.142	`PI`
360	about 6.283	`TAU`

`TAU` is a built-in constant equal to a full turn (2 times PI).

Setting up a practice scene

Create a new project called `Chapter06` (or use a scratch project). Then:

1. Create a new scene with a **Node2D** as the root. Rename it `Main`.
2. Save it as `main.tscn`.
3. Add a child **Polygon2D** and rename it `Box`.

Typing polygon points by hand in the Inspector is slow, so we build the shape in code.

```
Main (Node2D)
└── Box (Polygon2D)      <- script: box.gd
```

`box.gd`, attached to the **Box** node (a `Polygon2D`):

```gdscript
extends Polygon2D

func _ready() -> void:
    polygon = PackedVector2Array([
        Vector2(-40, -40),
        Vector2(40, -40),
        Vector2(40, 40),
        Vector2(-40, 40),
    ])
    color = Color(0.9, 0.4, 0.2)
    position = Vector2(300, 200)
```

Run the scene (F6). You see an orange square, centered on (300, 200). The polygon is drawn around its own origin, so the origin sits at the middle of the square. That is exactly what we want for something we plan to rotate.

> **Try it:** Add `rotation_degrees = 45.0` as a line at the end of `_ready`. Run it. The square becomes a diamond. Now change it to `scale = Vector2(2, 0.5)` and run again. Which direction became bigger?

Expected result: the diamond appears rotated clockwise; the scaled square is twice as wide and half as tall.

6.2 Local vs. global, and parent-child transforms

Children move with their parents

When a node is a child of a `Node2D`, its `position` is measured from the **parent's origin**, and it is also affected by the parent's rotation and scale. This is called **transform inheritance**. It is one of the most useful ideas in 2D games.

Think of a sword held by a knight. If the sword is a child of the knight, then moving, turning, or shrinking the knight moves, turns, and shrinks the sword too. The sword only needs to know where it is *relative to the knight*.

```
Main (Node2D)
└── Parent (Node2D)          position (300, 200)
    └── Child (Polygon2D)    position (100, 0)
```

The child's **local** position is (100, 0). Its **global** position is where it really is on screen: (400, 200).

Now rotate the parent by 90 degrees. The child's local position is still (100, 0), but the child is swung around the parent. Its global position becomes (300, 300). The child orbits the parent like a planet.

Here is the same experiment as a script. The scene tree is above; attach this to **Parent**. The `Child` node needs a polygon of its own, so give it `child.gd` too (shown after).

`parent.gd`, attached to **Parent** (`Node2D`):

```gdscript
extends Node2D

func _ready() -> void:
    position = Vector2(300, 200)
    rotation_degrees = 90.0
    var child: Node2D = $Child
    print("Child local:  ", child.position)
    print("Child global: ", child.global_position)
```

`child.gd`, attached to **Child** (`Polygon2D`):

```gdscript
extends Polygon2D

func _ready() -> void:
    polygon = PackedVector2Array([
        Vector2(-15, -15),
        Vector2(15, -15),
        Vector2(15, 15),
        Vector2(-15, 15),
    ])
    color = Color(0.2, 0.7, 0.9)
    position = Vector2(100, 0)
```

> **Note:** Godot starts a node's `_ready` for children *before* the parent. So `child.gd` runs first and sets the child's position, then `parent.gd` runs and reads it. That is why the prints work.

Expected output in the Output panel (the numbers may show tiny rounding errors such as `300.0001`, because computers store decimals approximately):

```
Child local:  (100, 0)
Child global: (300, 300)
```

Global properties and conversion

Property / method	What it gives
`position`	Position relative to the parent
`global_position`	Position in the whole world
`rotation` / `global_rotation`	Local / world rotation in radians
`scale` / `global_scale`	Local / world scale
`to_global(local_point)`	Convert a point from this node's space to world space
`to_local(global_point)`	Convert a world point into this node's space

Use **local** values when you only care about "where am I inside my parent". Use **global** values when you compare two things that live in different parts of the tree, such as the distance between a player and an enemy.

> **Warning:** Setting `global_position` on a child, and `position` on its parent in the same frame, is a classic source of confusion. Pick one space per task. A good rule: *move things with* `position`*; compare things with* `global_position`.

Pivot points

A node rotates and scales around its own origin. If you want a door that swings on a hinge, put the node's origin at the hinge and the artwork off to one side. That is why we often build a shape around its origin (so it spins in place), or put the shape off to the side (so it swings around a pivot).

> **Try it:** In the scene above, change `Child`'s position to `Vector2(0, 0)` and rotate the parent. The child just spins in place. Now change it back to `(100, 0)` and watch it orbit. You have just built a pivot.

6.3 Sprite2D and textures

Showing an image

`Sprite2D` draws a **texture** (an image) at its node's position. Its main property is `texture`.

To use your own picture:

1. Copy a `.png` file into your project folder (or drag it into the **FileSystem** dock).
2. Godot **imports** it automatically. Click the file in the FileSystem dock and open the **Import** tab to see settings.
3. Drag the file from the FileSystem dock onto the `Texture` slot of a `Sprite2D` (or drag it into the 2D view to create a Sprite2D at once).

By default a sprite is **centered**: its position marks the middle of the picture. Turn off `Centered` in the Inspector if you want the position to be the picture's top-left corner. `Offset` shifts the picture without moving the node.

Pixel art needs "Nearest" filtering

When Godot scales a texture, it blends nearby pixels. That makes photos look smooth but makes pixel art look blurry. For pixel art, choose **Nearest** filtering:

- For one node: Inspector > **CanvasItem > Texture > Filter** > `Nearest`.
- For the whole project (recommended for pixel-art games): **Project Settings > Rendering > Textures > Canvas Textures > Default Texture Filter** > `Nearest`. You may need to turn on **Advanced Settings** (top right of the window) to see it.

Region: show only part of an image

If your image holds several items, you can show one piece. In the Inspector open **Region**, tick `Enabled`, and set `Rect` to `(x, y, width, height)` in pixels. Only that rectangle of the texture is drawn.

Sprite sheets: hframes, vframes, frame

A **sprite sheet** is a grid of equal-sized pictures in one file. Set `Animation > Hframes` to the number of columns and `Vframes` to the number of rows. Then `Frame` picks which cell to show. Frames are numbered left to right, then top to bottom, starting at 0.

```
Sheet with Hframes = 4, Vframes = 2

+----+----+----+----+
| 0  | 1  | 2  | 3  |
+----+----+----+----+
| 4  | 5  | 6  | 7  |
+----+----+----+----+
```

```
$Sprite2D.frame = 5             # second row, second column
$Sprite2D.frame_coords = Vector2i(1, 1)   # the same cell, by column and row
```

Chapter 9 turns these frames into animations. For now just know that `frame` is a number you can change.

Flipping

`Sprite2D` has `flip_h` and `flip_v` (in the Inspector under **Offset**). A character that walks left can reuse the right-facing picture:

```
$Sprite2D.flip_h = velocity.x < 0.0
```

Flipping the sprite is better than setting `scale.x = -1` for simple cases, because it does not disturb the physics shapes you will meet in Chapter 7.

Built-in textures for placeholder art

You do not need image files to see a sprite. A `Sprite2D`'s `Texture` slot can hold a **resource** (Chapter 2). In the Inspector click the `Texture` slot and choose **New GradientTexture2D**. Set its `Width` and `Height` in pixels. Then click the new texture's `Gradient` slot and choose **New Gradient** (without a gradient the texture draws nothing). A new gradient fades from black to white, and the texture runs it from left to right.

> **Note:** There is also `PlaceholderTexture2D`, which only stores a size. Its job is to reserve a size, not to guarantee a picture, so it may draw nothing when the game runs. We use `GradientTexture2D` when we need a visible stand-in.

Try it: a sprite from nothing

1. In `main.tscn` add a `Sprite2D` child named `Arrow`.
2. Set its `Texture` to a new `GradientTexture2D` with Width 96 and Height 32, and give it a **New Gradient**.
3. Set `Position` to (576, 324).
4. Run the scene.

Expected result: a rectangle that is dark on the left and bright on the right appears near the middle of the window. The bright end will become the "nose" of things that point somewhere.

6.4 Placeholder art with built-in nodes

Real art takes time. Professional teams build the game with placeholder shapes first, then replace them. You can too, using only built-in tools.

Tool	Best for	Notes
`ColorRect`	Panels, bars, backgrounds	It is a UI node (a `Control`); its origin is the **top-left** corner
`Polygon2D`	Any flat shape	You choose the points; origin is wherever you place it
`GradientTexture2D` in `Sprite2D`	A "sprite" that behaves like a real one	Good for practicing sprite properties
`_draw()`	Shapes made by script	Fully controlled by code

ColorRect

A `ColorRect` has `color` and `size`. It works inside a 2D scene, but its position is its *top-left corner*, not its center. To center one, shift it by half its size:

```
extends ColorRect

func _ready() -> void:
    size = Vector2(60, 60)
    position = -size / 2.0     # now the middle sits at the parent's origin
    color = Color.CORNFLOWER_BLUE
```

> **Note:** `Color` has many named constants, such as `Color.RED`, `Color.WHITE`, `Color.CORNFLOWER_BLUE`, or you can write `Color(r, g, b)` with each value from 0.0 to 1.0.

Drawing with `_draw()`

Every 2D node (every `CanvasItem`) can draw shapes by itself. Define `_draw()`, and Godot calls it when the node needs to be drawn. Inside it, call `draw_*` functions. Their coordinates are **local** to the node.

```
extends Node2D

func _draw() -> void:
    draw_rect(Rect2(-30, -30, 60, 60), Color.ORANGE)           # a square
    draw_circle(Vector2(80, 0), 20.0, Color.SKY_BLUE)          # a circle
    draw_line(Vector2(-50, 50), Vector2(50, 50), Color.WHITE, 3.0)  # a line, 3 px wide
```

Attach this to a `Node2D` and run. You see an orange square, a blue circle to its right, and a white line under the square.

`_draw()` runs once, then Godot remembers the result. If the picture must change, call `queue_redraw()` and Godot will call `_draw()` again on the next frame.

```
extends Node2D

var radius: float = 10.0

func _process(delta: float) -> void:
    radius += 20.0 * delta
    if radius > 60.0:
        radius = 10.0
    queue_redraw()
```

```
func _draw() -> void:
    draw_circle(Vector2.ZERO, radius, Color.TOMATO)
```

Run it: a red circle grows and snaps back, over and over.

> **Why it works:** `_process` changes the data (`radius`). `queue_redraw()` asks for a new picture. `_draw()` reads the data and paints it. Keeping *data* and *drawing* separate is a habit that pays off in every chapter.

6.5 Time and movement: `_process` and delta

Two kinds of "every frame"

A game is a loop: read input, update the world, draw the picture, repeat. Each pass is a **frame**. Godot gives you a function that runs every frame:

```
func _process(delta: float) -> void:
    pass
```

It runs as often as the screen refreshes: 60 times a second on many screens, 144 on others, and fewer when the computer is busy. In Chapter 7 you will meet its sibling `_physics_process`, which runs at a fixed rate. For now, think of `_process` as "once per drawn frame".

The problem: different computers, different speeds

Suppose we move a square 5 pixels every frame:

```
extends Node2D

func _process(delta: float) -> void:
    position.x += 5.0
```

On a 60 frames-per-second (FPS) machine the square moves 5 x 60 = 300 pixels per second. On a 144 FPS machine it moves 720 pixels per second. The game is more than twice as fast! The same bug makes games unplayable when the frame rate dips.

The solution: multiply by delta

`delta` is the time since the last frame in seconds. At 60 FPS it is about 0.0167. At 144 FPS it is about 0.0069. If you describe speed in **pixels per second** and multiply by `delta`, the result is the right distance for that frame:

```
extends Node2D

var speed: float = 300.0   # pixels per second

func _process(delta: float) -> void:
    position.x += speed * delta
```

FPS	delta (s)	Distance per frame	Distance per second
30	0.0333	10.0 px	300 px

FPS	delta (s)	Distance per frame	Distance per second
60	0.0167	5.0 px	300 px
144	0.0069	2.08 px	300 px

The speed per second is always 300. This is **frame-rate independence**.

> **Why it works:** Distance = speed x time. `speed` is pixels per second, `delta` is seconds, so `speed * delta` is pixels. The units match.

The rule: **anything that changes smoothly over time (movement, rotation, fading, timers) is multiplied by `delta`.** Things that happen once (a button press) are not.

```
extends Sprite2D

var spin_speed: float = PI   # radians per second: half a turn each second

func _process(delta: float) -> void:
    rotation += spin_speed * delta
```

> **Try it:** Put the spin script on the `Arrow` sprite from section 6.3. It turns once every two seconds. Change `spin_speed` to `TAU` and it turns once per second.

Expected result: the arrow rotates clockwise at a steady pace, no matter the computer.

A timer you build yourself

```
extends Node2D

var elapsed: float = 0.0

func _process(delta: float) -> void:
    elapsed += delta
    if elapsed >= 2.0:
        elapsed = 0.0
        print("Two seconds passed")
```

Chapter 9 introduces the `Timer` node, which is usually better. But seeing this pattern shows you what a timer really is: a number that grows by `delta`.

6.6 Camera2D

By default the screen shows the world starting from (0, 0) at the top-left. A `Camera2D` moves that window. Add one as a child of the thing you want to follow, and the camera moves with it automatically. This works because of transform inheritance (section 6.2).

Useful properties:

Property	Meaning
`zoom`	`Vector2(2, 2)` zooms in (things look twice as big)
`position_smoothing_enabled`	Camera glides instead of snapping

Property	Meaning
`position_smoothing_speed`	How quickly it catches up (default 5.0)
`limit_left`, `limit_right`, `limit_top`, `limit_bottom`	World edges the camera will not cross
`offset`	Shift the view without moving the camera node

The camera is centered on its node: the node's position is the middle of the screen. Only one camera is active at a time.

```
Main (Node2D)
└── Player (Node2D)
    ├── Body
    └── Camera2D       <- child of Player: follows it
```

```gdscript
# Fragment: set up a smooth camera from a script on the Player.
func _ready() -> void:
    var cam: Camera2D = $Camera2D
    cam.position_smoothing_enabled = true
    cam.position_smoothing_speed = 6.0
    cam.zoom = Vector2(1.5, 1.5)
```

> **Note:** Smoothing is a matter of taste. Turn it off if you want the camera locked tightly to the target, or lower `position_smoothing_speed` for a lazier camera.

You will use a camera in the lab.

6.7 Input: the Input Map and reading actions

Why actions, not keys

You could write `if key == W`. But then players with other keyboards, controllers, or different preferences are stuck. Godot's solution is the **Input Map**: you invent a named **action** such as `move_up`, and in a settings table you attach any number of keys, buttons, or stick directions to it. Code only mentions the action name.

To open it: **Project > Project Settings > Input Map** tab.

Create actions

For each action below:

1. Type the name in the **Add New Action** box and press **Add** (or Enter).
2. Click the + button at the right of the new row.
3. In the dialog, press the key you want. (If your window offers a choice about **Physical Keycode**, pick it: it uses the key's *position*, so WASD stays in the same place on French or German keyboards.)
4. Click **OK**.

Action	Key to add
`move_left`	A (you may also add Left arrow)
`move_right`	D (and Right arrow)

Action	Key to add
`move_up`	W (and Up arrow)
`move_down`	S (and Down arrow)
`sprint`	Shift

Godot also comes with built-in actions starting with `ui_` (such as `ui_left`, `ui_accept`). They are meant for menus. Make your own for gameplay.

> **Note:** You can create actions in code too, which is useful for tools but unneeded for this course. A fragment:
> ```gdscript

Fragment: add an action named "jump" mapped to the Space bar.

func _ready() -> void: if not InputMap.has_action("jump"): InputMap.add_action("jump") var ev := InputEventKey.new() ev.physical_keycode = KEY_SPACE InputMap.action_add_event("jump", ev) ```

Polling: asking "is it held right now?"

The `Input` singleton (a ready-made global object, like the autoloads from Chapter 5) answers questions about the current state:

Call	True when…
`Input.is_action_pressed("a")`	the action is **held down** this frame
`Input.is_action_just_pressed("a")`	the action went down **this frame** (one frame only)
`Input.is_action_just_released("a")`	the action was let go this frame
`Input.get_action_strength("a")`	0.0 to 1.0 (analog sticks give in-between values)

Use *pressed* for things that continue while held (walking). Use *just_pressed* for one-time events (jumping, shooting once, opening a menu).

```
extends Node2D

func _process(delta: float) -> void:
    if Input.is_action_pressed("move_right"):
        position.x += 200.0 * delta
    if Input.is_action_just_pressed("sprint"):
        print("Sprint button went down!")
```

Attach this to a node in `main.tscn`, and run. Hold D: the node slides right. Tap Shift: one message per tap, not a flood.

One number: `Input.get_axis`

`Input.get_axis("negative_action", "positive_action")` returns -1, 0, or +1 (or in between for sticks). It saves two `if` lines:

```
var direction: float = Input.get_axis("move_left", "move_right")
position.x += direction * 200.0 * delta
```

Two numbers: `Input.get_vector`

For free 2D movement use `Input.get_vector(negative_x, positive_x, negative_y, positive_y)`. The argument order is **left, right, up, down**. It returns a `Vector2`:

```
var direction: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
```

Pressing W gives `(0, -1)` (up is negative y!). Pressing D gives `(1, 0)`. Pressing W and D together gives about `(0.707, -0.707)`: its length is 1, not 1.414. Godot already normalized it, so diagonal movement is not faster than straight movement.

> **Why it works:** Without normalizing, "right 1 and up 1" would have length about 1.41, so diagonal speed would be 41% faster. `get_vector` fixes that for you (and also handles a small dead zone for sticks).

6.8 Events: `_input` and `_unhandled_input`

Polling is great for continuous movement. For *events* (a click, a key tap), Godot sends an `InputEvent` object to two functions you can define:

Function	Called when...
`_input(event)`	Every input event, first, before the UI sees it
`_unhandled_input(event)`	Only events that nothing else (like a button) used

Use `_unhandled_input` for gameplay. Then a click on a pause-menu button does not also fire your sword. Use `_input` only when you must see everything.

An event has helper methods to ask about actions:

```
extends Node2D

func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("sprint"):
        print("Shift pressed (event)")
    elif event.is_action_released("sprint"):
        print("Shift released (event)")
```

Mouse clicks arrive as `InputEventMouseButton`. To check type and fields safely, cast with `as`; it yields `null` when the type is wrong:

```
extends Node2D

func _unhandled_input(event: InputEvent) -> void:
    var click := event as InputEventMouseButton
    if click and click.pressed and click.button_index == MOUSE_BUTTON_LEFT:
        print("Left click at ", get_global_mouse_position())
```

A pattern to remember:

```
# Fragment: inside _unhandled_input(event: InputEvent).
# Cast to the event type you want; the result is null if it is a different type.
var click := event as InputEventMouseButton
if click and click.pressed:
    print("Clicked at ", click.position)
```

An event can be marked "used up" so no one else gets it: `get_viewport().set_input_as_handled()`. You rarely need it at this stage.

> **Try it:** Attach the click script to `Main`. Run and click several places. The Output panel prints different positions each time. Then right-click: nothing prints, because the button index is not left.

6.9 The mouse and Vector2 math

Mouse position

Any `CanvasItem` (every 2D node) has two helpful functions:

Call	Meaning
`get_global_mouse_position()`	Mouse in **world** coordinates (follows the camera)
`get_local_mouse_position()`	Mouse relative to this node's own origin

When there is a camera, always use `get_global_mouse_position()` for gameplay, because the window position and the world position differ once the camera moves.

Vector2 toolbox

A `Vector2` is a pair (x, y). It might be a position, a direction, or a speed. Here are the pieces you will use constantly:

Expression	Result
`Vector2(3, 4).length()`	`5.0` (the Pythagorean theorem)
`Vector2(3, 4).normalized()`	`(0.6, 0.8)`: same direction, length 1
`a.distance_to(b)`	Distance between two points
`a.direction_to(b)`	Normalized vector pointing from `a` toward `b`
`a.angle_to_point(b)`	Angle in radians of the direction from `a` to `b`
`a.lerp(b, 0.25)`	A point 25% of the way from `a` to `b`
`v.clamp(lo, hi)`	Keep each part of `v` between the vectors `lo` and `hi`
`v.limit_length(100.0)`	Shorten `v` to at most 100 long
`a.move_toward(b, step)`	Move `a` toward `b` by at most `step`
`Vector2.ZERO`, `UP`, `DOWN`, `LEFT`, `RIGHT`	Handy constants. `Vector2.UP` is `(0, -1)`

Plain numbers have matching helpers: `clampf(x, lo, hi)`, `lerpf(a, b, t)`, `absf`, `minf`, `maxf`.

Arithmetic works on vectors directly: `a + b`, `a - b`, `a * 2.0`, `a / 2.0`. The vector from `a` to `b` is `b - a`.

Worked example 1: walk toward the mouse

```
extends Node2D

var speed: float = 250.0

func _process(delta: float) -> void:
    var target: Vector2 = get_global_mouse_position()
```

```
    if global_position.distance_to(target) > 4.0:
        global_position += global_position.direction_to(target) * speed * delta
```

The `> 4.0` check stops the node from shaking when it arrives. Without it, it would jump back and forth over the target.

Worked example 2: smooth follow with lerp

```
extends Node2D

func _process(delta: float) -> void:
    var target: Vector2 = get_global_mouse_position()
    global_position = global_position.lerp(target, 5.0 * delta)
```

Each frame it covers part of the remaining gap, so it speeds up far away and eases in near the target. This is a handy way to make a smooth glide.

> **Warning:** `lerp(target, 0.1)` without `delta` follows faster at high frame rates. Multiply the weight by `delta`, as above, for rough frame-rate independence. (It is not exact, but close enough for visual smoothing.)

Worked example 3: clamp inside the screen

```
extends Node2D

func _process(delta: float) -> void:
    position += Input.get_vector("move_left", "move_right", "move_up", "move_down") * 300.0 * delta
    var screen: Vector2 = get_viewport_rect().size
    position = position.clamp(Vector2.ZERO, screen)
```

`get_viewport_rect().size` is the window size in world units. `clamp` keeps both x and y inside the rectangle from (0, 0) to the bottom-right corner.

Worked example 4: face the mouse

To point a node at the mouse, there are two ways:

```
# Way 1: a ready-made helper on Node2D
look_at(get_global_mouse_position())

# Way 2: compute the angle yourself
rotation = global_position.angle_to_point(get_global_mouse_position())
```

Both turn the node so its **positive x direction (to the right of its picture)** points at the mouse. So draw your art pointing right.

> **Try it:** Attach this to the `Arrow` sprite (centered at (576, 324)):
>
> ```gdscript extends Sprite2D
>
> func _process(_delta: float) -> void: look_at(get_global_mouse_position()) ```
>
> Run it and move the mouse around. The bright end of the arrow should chase the mouse cursor.

The name `_delta` (with an underscore) tells Godot and readers "I know this parameter is unused". It avoids a warning.

Lab 6: Move and Aim

Time: 40 to 60 minutes.

Goal: A blue square you steer with WASD across a big floor, a camera that follows it, and a pointer sprite on the square that always faces the mouse. Hold Shift to run faster.

Step 1. Project and input

1. Create a new project named `MoveAndAim` (Compatibility or Forward+ both work).
2. Open **Project Settings > Input Map** and add the five actions from the table in section 6.7: `move_left` (A), `move_right` (D), `move_up` (W), `move_down` (S), `sprint` (Shift).

Step 2. The scene tree

Create these nodes. The root is a `Node2D`. Save the scene as `main.tscn`, and set it as the main scene when Godot asks on first run.

```
Main (Node2D)
├── Floor (Node2D)          <- script: floor_grid.gd
└── Player (Node2D)         <- script: player.gd
    ├── Pointer (Sprite2D)
    └── Camera2D (Camera2D)
```

Create the scripts in a folder such as `res://scripts/` by right-clicking a node and choosing **Attach Script**.

Step 3. The floor

Without a background you cannot tell the camera is moving. This script draws a grid that is bigger than the screen.

`floor_grid.gd`, attached to **Floor** (`Node2D`):

```gdscript
extends Node2D

const CELL: int = 100
const HALF_SIZE: int = 1500

func _draw() -> void:
    draw_rect(Rect2(-HALF_SIZE, -HALF_SIZE, HALF_SIZE * 2, HALF_SIZE * 2), Color(0.12, 0.14, 0.18))
    for i in range(-HALF_SIZE, HALF_SIZE + 1, CELL):
        var line_color := Color(0.25, 0.28, 0.35)
        if i == 0:
            line_color = Color(0.8, 0.5, 0.2)   # the world axes
        draw_line(Vector2(i, -HALF_SIZE), Vector2(i, HALF_SIZE), line_color, 2.0)
        draw_line(Vector2(-HALF_SIZE, i), Vector2(HALF_SIZE, i), line_color, 2.0)
```

The floor is 3000 by 3000 pixels, with lines every 100 pixels. Two orange lines cross at the world origin (0, 0).

Step 4. The pointer sprite

Select the **Pointer** node and set up its Inspector properties:

- **Texture**: New `GradientTexture2D`, Width `48`, Height `16`, with its `Gradient` set to **New Gradient**.
- **Offset**: `(32, 0)`. This shifts the picture to the right of the node's origin, so the pointer orbits around the player's center like the hand of a clock.

Leave `Position` at (0, 0). Leave `Centered` on.

Step 5. The camera

Select **Camera2D** and tick **Position Smoothing > Enabled**. Leave the other settings as they are.

Step 6. The player script

`player.gd`, attached to **Player** (`Node2D`):

```
extends Node2D

@export var walk_speed: float = 280.0
@export var sprint_speed: float = 480.0

@onready var pointer: Sprite2D = $Pointer

func _ready() -> void:
    position = Vector2.ZERO

func _process(delta: float) -> void:
    # 1. Move
    var direction: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    var speed: float = walk_speed
    if Input.is_action_pressed("sprint"):
        speed = sprint_speed
    position += direction * speed * delta

    # 2. Keep the player on the floor
    position = position.clamp(Vector2(-1450, -1450), Vector2(1450, 1450))

    # 3. Aim the pointer at the mouse
    pointer.look_at(get_global_mouse_position())

func _draw() -> void:
    draw_rect(Rect2(-20, -20, 40, 40), Color(0.25, 0.6, 1.0))
    draw_rect(Rect2(-20, -20, 40, 40), Color.WHITE, false, 2.0)
```

How it works:

- `Input.get_vector` returns a ready-to-use direction (length at most 1).
- `speed * delta` keeps the movement the same on every computer.
- `clamp` stops the player at the edge of the floor.
- `pointer.look_at(...)` turns only the pointer, not the player's square, because the player's own rotation stays 0.
- `_draw()` paints the square. The second `draw_rect` call draws a white outline: the third argument `false` means "not filled", and the fourth is the line width.

> **Note:** `Camera2D` is a child of `Player`, so it follows automatically. Because `Pointer` is also a child of `Player`, it moves with the square too.

Step 7. Run it

Press **F5**.

Test checklist

1. Press W, A, S, D: the square moves and the grid scrolls past. Up is up.
2. Hold two keys, such as W and D: you move diagonally at normal speed.
3. Hold Shift while moving: you go faster.
4. Move the mouse in circles: the pointer's bright end follows it, even while you move.
5. Walk to the edge of the floor: the player stops and you see the dark void outside.

Expected result: a blue square with a bright arrow that always points at your mouse, a camera that glides after the square, and a scrolling grid that proves the camera works.

If something is wrong

Symptom	Likely cause
Output says `The InputMap action "move_left" doesn't exist`	Typo in the Input Map name or in the script
Pointer rotates but always points away	The art is drawn pointing left; draw it pointing right
Player doesn't show	`_draw()` is spelled wrong, or the script is on the wrong node
Camera does not move	`Camera2D` is not a child of `Player`
Pointer is not orbiting	`Offset` not set, or the Pointer's `Position` is not (0, 0)

Stretch goals

- Add a second sprite that lags behind the player using `lerp` (a "pet").
- Make left click move a small `Polygon2D` marker to the mouse position using `_unhandled_input`.
- Zoom the camera in and out with the mouse wheel (`MOUSE_BUTTON_WHEEL_UP` and `MOUSE_BUTTON_WHEEL_DOWN` in an `InputEventMouseButton`).

Chapter summary

- Godot 2D uses (0, 0) at the top-left, x to the right, and **y downward**. Up is negative y.
- A `Node2D` has `position`, `rotation` (radians, clockwise), and `scale`. Children inherit the parent's transform, so local and global values differ.
- `Sprite2D` shows a texture. Use Nearest filtering for pixel art, `region` to crop, and `hframes` / `vframes` / `frame` for sprite sheets.
- Placeholder art can come from `Polygon2D`, `ColorRect`, `GradientTexture2D`, or `_draw()`, so you can build gameplay before art exists.
- Multiply continuous changes by `delta` so speed is measured in units per second, not per frame.

- Define **actions** in the Input Map. Read them with `is_action_pressed` (held), `is_action_just_pressed` (one frame), and `Input.get_vector` (a normalized 2D direction).
- Use `_unhandled_input` for gameplay events so the UI gets first pick.
- `Vector2` math (`normalized`, `distance_to`, `direction_to`, `lerp`, `clamp`, `look_at`) handles aiming, chasing, and boundaries.

Review questions

1. In Godot's 2D coordinate system, does a larger y value move a node up or down on the screen?
2. In code, what unit does the `rotation` property use? Give the value for a quarter turn.
3. A parent `Node2D` sits at (100, 100) and is rotated 90 degrees (clockwise). Its child has local position (50, 0). What is the child's global position?
4. True or false: the `delta` passed to `_process` is always exactly the same number.
5. A sprite moves at 200 pixels per second. In one frame, `delta` is 0.02. How many pixels should it move in that frame?
6. Which filter setting keeps pixel art sharp when scaled, and where can you set it for the whole project?
7. A sprite sheet has 4 columns and 2 rows. What values do `hframes` and `vframes` take, and which row and column does `frame = 5` show?
8. What is the difference between `Input.is_action_pressed` and `Input.is_action_just_pressed`? Give one use for each.
9. Why is `_unhandled_input` usually better than `_input` for gameplay controls?
10. What are `Vector2(3, 4).length()` and `Vector2(3, 4).normalized()`? And what is the length of the vector `Input.get_vector` returns when two perpendicular keys are held?

Exercises

1. **Practice (easy): Spinning logo.** Make a `Polygon2D` triangle that rotates once every 4 seconds and slowly pulses between scale 1 and scale 1.5. Use `_process`, `delta`, and the `sin` function (for the pulse: `scale = Vector2.ONE * (1.25 + 0.25 * sin(elapsed * 2.0))`, where `elapsed` is a float you increase by `delta`). Check that it runs at the same speed if you change **Project Settings > Application > Run > Max FPS** to 30.
2. **Practice (easy): Input Map practice.** Add an action `reset` mapped to R. In a script, when it is pressed (just pressed, not held), set the node's position back to the center of the window, using `get_viewport_rect().size / 2.0`.
3. **Apply (medium): Mouse chaser.** Create a node that smoothly follows the mouse with `lerp`, rotates to face the direction it is moving (`rotation = velocity.angle()` where `velocity` is the movement this frame), and leaves the pointer alone when the mouse is within 5 pixels of the node.
4. **Apply (medium): Orbiting moon.** Build a planet with a child "moon" at local position (120, 0). Rotate the *planet* only, and watch the moon orbit. Then add a grandchild "moon pebble" to the moon, and predict what happens when you rotate the moon as well. Write down your prediction before running.
5. **Challenge (hard): Mini twin-stick prototype.** Extend the lab: pressing left click spawns a small `Polygon2D` bullet at the player's position, moving in the pointer's direction at 600 pixels per second, and removing itself with `queue_free()` after 1.5 seconds or when it leaves the floor. Hint: write a `bullet.gd` with `var velocity: Vector2` and spawn it with `Polygon2D.new()` or an instanced scene (Chapter 2). Set its `global_position` when you spawn it.

Common mistakes

- **The square moves much faster on my friend's computer** -> movement written as `position.x += 5` without `delta` -> write `position.x += speed * delta` with speed in pixels per second.
- **Everything is rotated the wrong way, or the angle is huge** -> passing degrees to `rotation` -> use `rotation_degrees`, or `deg_to_rad(...)`.
- **Pressing W moves the player down** -> forgetting that positive y points down, or swapping the up/down arguments of `get_vector` -> remember the order is left, right, **up**, down, and that up is negative.
- **Error:** `The InputMap action "move_left" doesn't exist` -> the name in the script differs from the Input Map, or the action was never added -> check spelling in both places; names are case-sensitive.
- **The mouse aim is off once the camera moves** -> using the window position instead of the world position -> use `get_global_mouse_position()`.
- **A held key triggers a one-time action every frame (for example, a jump repeats)** -> using `is_action_pressed` for an event -> use `is_action_just_pressed`.

Further reading

- Godot docs: *2D movement overview*: https://docs.godotengine.org/en/stable/tutorials/2d/2d_movement.html
- Godot docs: *Using InputEvent*: https://docs.godotengine.org/en/stable/tutorials/inputs/inputevent.html
- Godot docs: *Input examples*: https://docs.godotengine.org/en/stable/tutorials/inputs/input_examples.html
- Godot docs: *Vector math*: https://docs.godotengine.org/en/stable/tutorials/math/vector_math.html
- Godot docs: *Matrices and transforms*: https://docs.godotengine.org/en/stable/tutorials/math/matrices_and_transforms.html
- Class reference: `Node2D`, `Sprite2D`, `Camera2D`, `Vector2` (search at https://docs.godotengine.org/en/stable/classes/)

Chapter 7: Physics and Collisions

Learning objectives

By the end of this chapter you will be able to:

- Explain the difference between `_process` and `_physics_process`, and choose the right one.
- Pick the correct physics body (`StaticBody2D`, `CharacterBody2D`, `RigidBody2D`, `Area2D`) for a given job.
- Give a body a collision shape with `CollisionShape2D`, and set collision layers and masks so only the right things touch.
- Move a `CharacterBody2D` with `velocity` and `move_and_slide()`, for both top-down and platformer games.
- React to overlaps with `Area2D` signals such as `body_entered`, `area_entered`, `mouse_entered`, and `input_event`.
- Describe `TileMapLayer` and `RayCast2D`, and find collision bugs with the "Visible Collision Shapes" debug option.
- Build a complete small game, "Coin Dash", with a score, a win condition, and a restart.

Key terms

- **Physics engine**: the part of Godot that moves bodies, detects touches, and resolves them.
- **Physics tick**: one fixed step of the physics engine (60 per second by default). `_physics_process` runs once per tick.
- **Collision shape**: an invisible outline (rectangle, circle, capsule...) that tells physics where a body is solid.
- **Body**: a node that takes part in physics: `StaticBody2D`, `CharacterBody2D`, `RigidBody2D`.
- **Area2D**: a region that *detects* things overlapping it but does not block them.
- **Collision layer**: the "I am on these channels" setting of an object.
- **Collision mask**: the "I look for things on these channels" setting of an object.
- **Velocity**: how fast and in what direction a body moves, in pixels per second.
- **Signal**: an event a node emits that other code can listen to (Chapter 5).
- **TileMapLayer**: a node that paints a grid of tiles, with optional collision, from a `TileSet`.
- **Raycast**: an invisible line that reports what it hits first.

7.1 Physics versus process

In Chapter 6 you used `_process(delta)`, which runs once per drawn frame. Its rate changes with the computer. That is fine for visuals, but it is a poor base for physics. Physics calculations are stable when the time step is the same every time.

Godot therefore runs the physics engine at a **fixed rate**: **60 ticks per second** by default. You can see it in **Project Settings > Physics > Common > Physics Ticks per Second**. To run your own code on those ticks, use `_physics_process`:

```
extends Node
```

```gdscript
func _physics_process(delta: float) -> void:
    print("physics tick, delta = ", delta)
```

Here `delta` is always the same value (about 0.01667 s at 60 ticks), unlike in `_process`.

	`_process(delta)`	`_physics_process(delta)`
Runs	once per rendered frame	once per physics tick
Rate	varies with the computer	fixed (60/s by default)
Use for	animation, visuals, UI, camera effects	moving physics bodies, `move_and_slide`, forces

> **Why it works:** If a body moves a different distance each step, it can skip through a thin wall at low frame rates (called **tunneling**) or behave differently on faster machines. A fixed step makes behavior repeatable.

Rule of thumb: if the code touches `velocity`, `move_and_slide()`, or applies forces, put it in `_physics_process`. Everything cosmetic goes in `_process`.

> **Try it:** Create a scene with one `Node` and the script above. Run it and look at the Output panel. Then change **Physics Ticks per Second** to 30, run again, and see how `delta` changes to about 0.0333.

Expected result: `delta` is almost the same on every line, and it changes when you change the tick rate. Put the setting back to 60 afterward.

7.2 The four physics bodies

Godot 2D has four main nodes for physics. They all inherit from `CollisionObject2D`, which gives them layers, masks, and the ability to hold shapes.

Node	Moved by	Blocks others?	Typical use
`StaticBody2D`	nothing (it stays put)	Yes	walls, floors, platforms
`CharacterBody2D`	**your code** (`velocity`)	Yes	player, enemies, anything you steer
`RigidBody2D`	**the physics engine** (forces, gravity)	Yes	crates, balls, falling debris
`Area2D`	your code, or nothing	**No**	coins, hit zones, triggers, sensors

Choose by asking: *Who decides where this thing goes?*

- **Not moving at all:** `StaticBody2D`.
- **I decide, frame by frame, using input or AI:** `CharacterBody2D`.
- **Physics decides (it bounces, rolls, falls):** `RigidBody2D`.
- **I only want to know when something is inside a region:** `Area2D`.

There is also `AnimatableBody2D`, a static body that you move with animation (moving platforms). We do not use it in this chapter.

Every body needs a shape

A body does nothing until it has at least one `CollisionShape2D` child with a shape. The body is the *thing*; the shape is its *outline for physics*. The editor shows a warning triangle next to a body that has no shape.

Common shapes (assign in the Inspector under `Shape`):

Shape	Properties	Good for
`RectangleShape2D`	`size`	boxes, walls, simple characters
`CircleShape2D`	`radius`	balls, coins, pickups
`CapsuleShape2D`	`radius`, `height`	upright characters (smooth on slopes and edges)
`WorldBoundaryShape2D`	normal, distance	an endless floor
`ConvexPolygonShape2D`	points	custom outlines (convex only)

Different tools exist for more complex outlines, such as `CollisionPolygon2D`. For beginners, rectangles, circles, and capsules cover nearly everything.

The scene tree for a typical solid object looks like this:

```
Wall (StaticBody2D)
├── CollisionShape2D       <- shape: RectangleShape2D
└── Polygon2D              <- the visible picture (optional)
```

> **Warning:** Never scale a `CollisionShape2D` node (or its parent body) to resize it. Change the shape's `size` or `radius` instead. A scaled shape can behave oddly. Also, the shape does not have to match the picture exactly; make it a little smaller for forgiving gameplay.

Shapes made in code

You can create shapes in a script as well. This fragment sizes a rectangle shape at runtime:

```gdscript
# Fragment: runs inside a script on a body that has a CollisionShape2D child.
func _ready() -> void:
    var rect := RectangleShape2D.new()
    rect.size = Vector2(64, 32)
    $CollisionShape2D.shape = rect
```

Remember that shapes are **resources** (Chapter 2). If you assign the *same* shape resource to many scene instances, they share it, so changing one changes them all. Creating a new shape per instance, as above, avoids that.

7.3 Collision layers and masks

Imagine a crowded school with many hallways. You decide which hallways you *stand in* (your layer) and which hallways you *watch for others* (your mask). A collision is detected when one object's mask contains a layer the other object occupies.

Godot gives each object two settings:

- `collision_layer`: the layers this object **is on**.
- `collision_mask`: the layers this object **scans for**.

There are 32 layers, shown as numbered squares in the Inspector under **Collision**. Layer 1 is on by default for both.

Naming layers

Numbers are hard to remember. Give them names: **Project Settings > Layer Names > 2D Physics**. Name layer 1 `player`, layer 2 `world`, layer 3 `coins`. The Inspector then shows the names when you hover the squares.

Example setup for Coin Dash

Object	Node type	Layer (is on)	Mask (looks for)	Why
Player	`CharacterBody2D`	1 `player`	2 `world`	Bumps into walls only; coins do not stop it
Wall	`StaticBody2D`	2 `world`	none	Walls just exist; the player scans for them
Coin	`Area2D`	3 `coins`	1 `player`	Notices the player when it overlaps

Read the table as sentences: "The player is on layer 1 and looks for layer 2, so it collides with walls." "The coin is on layer 3 and looks for layer 1, so it notices the player."

> **Note:** For two physics bodies, a collision happens if *either* object's mask includes the other's layer. In practice, set the mask on the one doing the moving and keep the other's mask empty. That keeps your thinking simple: *the mover decides what it hits*.

The fastest way to break your game is to leave everything on layer 1 and mask 1. Then everything hits everything, and you cannot tell why. Spend 30 seconds setting layers; it saves hours.

Layers and masks in code

```gdscript
# Fragment: runs inside a script on a CollisionObject2D.
func _ready() -> void:
    set_collision_layer_value(3, true)    # be on layer 3
    set_collision_layer_value(1, false)   # leave layer 1
    set_collision_mask_value(1, true)     # look for layer 1
```

The functions take the layer **number** (1 to 32), not a bit mask. Use these helpers; they are far clearer than the raw `collision_layer` integer.

> **Try it:** Make a scene with two `StaticBody2D` nodes, one on layer 1 and one on layer 2, and a `CharacterBody2D` (you will build this in 7.4). Change the character's mask between 1, 2, and both. Note which wall it walks through.

7.4 CharacterBody2D and `move_and_slide`

`CharacterBody2D` is made for characters you steer. It has a built-in property `velocity`, a `Vector2` in **pixels per second**. You set `velocity`, then call `move_and_slide()`. Godot moves the body, stops it at solid things, and slides it along walls instead of sticking to them.

```
velocity = Vector2(200, 0)
move_and_slide()
```

Three facts to memorize:

1. `velocity` already exists. Do not declare your own `var velocity`.
2. `move_and_slide()` takes **no arguments**. It uses `velocity`.
3. `move_and_slide()` multiplies by the physics `delta` for you. So **do not multiply** `velocity` **by** `delta` when you set it.

After the call, Godot may have changed `velocity` (for example, it sets the part pushing into a wall to zero). That is why the same variable is used both as input and output.

Top-down movement

In a top-down game (seen from above, like classic Zelda) there is no gravity and no "floor". Set the body to **floating** mode:

```
TopDownTest (Node2D)
└── Player (CharacterBody2D)     <- script: top_down_player.gd
    └── CollisionShape2D          <- RectangleShape2D, size (32, 32)
```

`top_down_player.gd`, attached to **Player** (`CharacterBody2D`):

```
extends CharacterBody2D

@export var speed: float = 250.0

func _ready() -> void:
    motion_mode = MOTION_MODE_FLOATING

func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    velocity = direction * speed
    move_and_slide()

func _draw() -> void:
    draw_rect(Rect2(-16, -16, 32, 32), Color(0.2, 0.6, 1.0))
```

You need the `move_left`, `move_right`, `move_up`, and `move_down` actions from Chapter 6 in your Input Map.

Add a `StaticBody2D` wall next to the player (with a `CollisionShape2D` that has a `RectangleShape2D`). Run the scene and walk into it. You stop, and when you press diagonally against the wall, you slide along it.

> **Why it works:** `get_vector` gives a direction with length up to 1. Multiplying by `speed` makes `velocity` pixels per second. `move_and_slide` does the rest. When you let go of the keys, `get_vector` returns `(0, 0)`, so `velocity` becomes zero and you stop.
>
> **Note:** The default `MOTION_MODE_GROUNDED` is intended for platformers. It treats "up" specially (the floor). Use `MOTION_MODE_FLOATING` for top-down games. You can also set `Motion Mode` in the Inspector instead of in `_ready`.

Which layers does the player need?

In this test, leave the default layer 1 on both nodes and mask 1. You are testing movement only. In the lab you will set proper layers.

7.5 Gravity, jumping, and platformer movement

In a **side-view platformer**, there is gravity: a constant downward acceleration. Because y points down, gravity adds to `velocity.y`. A jump gives `velocity.y` a **negative** value (up), and gravity slowly cancels it, which produces an arc.

Where gravity comes from

The project has a default gravity: **Project Settings > Physics > 2D > Default Gravity** (980 pixels per second squared). Read it in code like this:

```
var gravity: float = ProjectSettings.get_setting("physics/2d/default_gravity")
```

> **Note:** Newer Godot 4 versions (4.3 and later) also give bodies a `get_gravity()` method that returns a gravity `Vector2`. Both ways work; the setting above works in every Godot 4 version.

`is_on_floor()`

`is_on_floor()` tells you if the body was standing on something at the end of its latest `move_and_slide()` call. `move_and_slide()` is what updates it, so when you read it near the top of `_physics_process` (before calling `move_and_slide()` again) it reflects the *previous* tick. That is fine for normal use.

Similar functions: `is_on_wall()` and `is_on_ceiling()`. For these to work the body needs a floor direction, which is `Vector2.UP` by default (the `up_direction` property).

A full platformer test

Add actions `jump` (Space), plus `move_left` and `move_right` (A/D, from Chapter 6) to the Input Map.

```
PlatformTest (Node2D)
├── Ground (StaticBody2D)          <- position (576, 600)
│   ├── CollisionShape2D           <- RectangleShape2D, size (1152, 40)
│   └── Polygon2D                  <- visible floor (see below)
├── Ledge (StaticBody2D)           <- position (700, 500)
│   ├── CollisionShape2D           <- RectangleShape2D, size (240, 20)
│   └── Polygon2D
```

```
└── Player (CharacterBody2D)        <- position (200, 500), script: platform_player.gd
    └── CollisionShape2D            <- RectangleShape2D, size (32, 48)
```

For the floor pictures, give each `Polygon2D` a color and points matching its shape: for the Ground use `(-576, -20)`, `(576, -20)`, `(576, 20)`, `(-576, 20)`; for the Ledge use `(-120, -10)`, `(120, -10)`, `(120, 10)`, `(-120, 10)`. (In the Inspector choose `Polygon`, set the array size to 4, and type the points.)

`platform_player.gd`, attached to **Player** (`CharacterBody2D`):

```gdscript
extends CharacterBody2D

const SPEED: float = 300.0
const JUMP_VELOCITY: float = -520.0

var gravity: float = ProjectSettings.get_setting("physics/2d/default_gravity")

func _physics_process(delta: float) -> void:
    # Gravity pulls down when we are in the air.
    if not is_on_floor():
        velocity.y += gravity * delta

    # Jump only when standing on something.
    if Input.is_action_just_pressed("jump") and is_on_floor():
        velocity.y = JUMP_VELOCITY

    # Let go of jump early for a short hop.
    if Input.is_action_just_released("jump") and velocity.y < 0.0:
        velocity.y *= 0.4

    # Left and right.
    var direction: float = Input.get_axis("move_left", "move_right")
    velocity.x = direction * SPEED

    move_and_slide()

func _draw() -> void:
    draw_rect(Rect2(-16, -24, 32, 48), Color(1.0, 0.6, 0.2))
```

Notice what *is* and *is not* multiplied by `delta`:

- `gravity * delta`: **yes**. Gravity is an acceleration (change in speed per second); to get this frame's change in speed, multiply by `delta`.
- `velocity.x = direction * SPEED`: **no**. This is a speed; `move_and_slide` handles the time step.

Run it: you walk, fall to the ground, jump with Space, land on the ledge, and can cut a jump short.

> **Try it:** Set `JUMP_VELOCITY` to `-800.0` and see the difference. Then lower the `gravity` by writing `velocity.y += gravity * 0.5 * delta`. Which feels "floatier"? Games feel good because designers tune these numbers.
>
> **Why it works:** A jump is just "set upward speed once". Every tick gravity adds a little downward speed. The speed goes from -520 up through 0 at the top of the arc and then becomes positive, so you fall. Landing sets `is_on_floor()` to true, and `move_and_slide` zeroes `velocity.y`.

7.6 RigidBody2D, briefly

A `RigidBody2D` is controlled by the physics engine. You do not set its position. You **push** it, and gravity, bounces, and friction do the rest. Use it for crates, balls, or physics puzzles.

```
BallTest (Node2D)
├── Floor (StaticBody2D)           <- position (576, 620), shape: RectangleShape2D (1152, 40)
└── Ball (RigidBody2D)             <- position (576, 100), script: ball.gd
    └── CollisionShape2D           <- CircleShape2D, radius 24
```

`ball.gd`, attached to **Ball** (`RigidBody2D`):

```gdscript
extends RigidBody2D

func _ready() -> void:
    var bouncy := PhysicsMaterial.new()
    bouncy.bounce = 0.7
    physics_material_override = bouncy

func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("jump"):
        apply_central_impulse(Vector2(0, -600))

func _draw() -> void:
    draw_circle(Vector2.ZERO, 24.0, Color.TOMATO)
```

The script reads the `jump` action, so add `jump` (Space) to the Input Map as in 7.5. The floor has no picture, so turn on **Debug > Visible Collision Shapes** (section 7.8) before running to see it, or give the Floor a `Polygon2D` child.

Run it: the ball falls and bounces. Each Space press kicks it upward. The `bounce` value of 0.7 means it keeps about 70 percent of its speed on each bounce.

Key properties: `mass`, `gravity_scale` (0 turns gravity off), `linear_velocity`, `linear_damp`, `freeze`.

> **Warning:** Do not set `position` on a `RigidBody2D` every frame. You fight the physics engine and get jitter. Use `apply_central_impulse`, `apply_force`, or set `linear_velocity`. (To teleport one, use `freeze` or a custom integrator; both are beyond this course.)

7.7 Area2D: detecting overlaps

An `Area2D` is a detector. Things pass through it, but it knows when they are inside. It needs a `CollisionShape2D` child, like any other collision object.

Signals you will use most:

Signal	Emitted when	Argument
`body_entered(body)`	a physics body (character, rigid, static, tilemap) enters	`Node2D`
`body_exited(body)`	such a body leaves	`Node2D`
`area_entered(area)`	another `Area2D` enters	`Area2D`

Signal	Emitted when	Argument
`area_exited(area)`	another `Area2D` leaves	`Area2D`
`mouse_entered()`	the mouse pointer moves over the area	none
`mouse_exited()`	the pointer leaves	none
`input_event(viewport, event, shape_idx)`	the mouse or touch interacts with the area	`Node`, `InputEvent`, `int`

Connect them in code (or with the Node dock's **Signals** tab, from Chapter 5):

```
func _ready() -> void:
    body_entered.connect(_on_body_entered)

func _on_body_entered(body: Node2D) -> void:
    print(body.name, " entered")
```

A pickup zone

```
PickupTest (Node2D)
├── Player (CharacterBody2D)       <- the top_down_player.gd from 7.4
│   └── CollisionShape2D
└── Zone (Area2D)                  <- position (600, 300), script: zone.gd
    └── CollisionShape2D           <- CircleShape2D, radius 50
```

`zone.gd`, attached to **Zone** (`Area2D`):

```
extends Area2D

var inside: bool = false

func _ready() -> void:
    body_entered.connect(_on_body_entered)
    body_exited.connect(_on_body_exited)

func _on_body_entered(body: Node2D) -> void:
    inside = true
    print(body.name, " entered the zone")
    queue_redraw()

func _on_body_exited(body: Node2D) -> void:
    inside = false
    print(body.name, " left the zone")
    queue_redraw()

func _draw() -> void:
    var c := Color(0.2, 0.9, 0.3, 0.5) if inside else Color(0.9, 0.2, 0.2, 0.5)
    draw_circle(Vector2.ZERO, 50.0, c)
```

Run it. Walk the player into the circle: it turns green, and the Output prints a message. Walk out: red again.

> **Why it works:** The player's mask must include the layer the Zone is on, or the Zone's mask must include the player's layer. With default settings (everything on layer 1, mask 1), they see each other. In the lab you set explicit layers.
>
> **Warning:** If you change physics state (turn off an area, add a body, change a shape's `disabled`) *inside* a physics callback like `body_entered`, Godot may print: *"Can't change this state while flushing queries."* Use a deferred change: `set_deferred("monitoring", false)` or `$CollisionShape2D.set_deferred("disabled", true)`. Calling `queue_free()` is safe.

Mouse input on an Area2D

An `Area2D` can respond to the mouse, which is handy for clickable hotspots (you will see a lot of this in the adventure game chapters). It needs `input_pickable` to be true (it is true by default) and a shape.

```
ClickTest (Node2D)
└── Button (Area2D)              <- position (576, 324), script: click_zone.gd
    └── CollisionShape2D         <- RectangleShape2D, size (200, 80)
```

`click_zone.gd`, attached to **Button** (`Area2D`):

```gdscript
extends Area2D

var hovered: bool = false
var clicks: int = 0

func _ready() -> void:
    mouse_entered.connect(_on_mouse_entered)
    mouse_exited.connect(_on_mouse_exited)
    input_event.connect(_on_input_event)

func _on_mouse_entered() -> void:
    hovered = true
    queue_redraw()

func _on_mouse_exited() -> void:
    hovered = false
    queue_redraw()

func _on_input_event(_viewport: Node, event: InputEvent, _shape_idx: int) -> void:
    var click := event as InputEventMouseButton
    if click and click.pressed and click.button_index == MOUSE_BUTTON_LEFT:
        clicks += 1
        print("Clicked ", clicks, " times")

func _draw() -> void:
    var c := Color(1.0, 0.8, 0.2) if hovered else Color(0.5, 0.5, 0.6)
    draw_rect(Rect2(-100, -40, 200, 80), c)
```

Run it. Hover over the rectangle: it brightens. Click it: the Output counts the clicks.

> **Note:** `input_event` only fires when the click lands inside the area's shape. Another collision object or a UI `Control` on top can get the click first.

7.8 TileMapLayer, raycasts, and debugging

TileMapLayer: building levels from tiles

Painting a level out of single sprites is slow. A **tile map** divides the world into a grid and fills cells with small pictures called **tiles**. It is fast to draw and edit, and it can add collision to every tile at once.

> **Note:** Before Godot 4.3 there was a node called `TileMap`. In **Godot 4.3 and newer** you use `TileMapLayer` instead (one node per layer). The old `TileMap` node still exists but is **deprecated**. This book uses `TileMapLayer`. If you see old tutorials with `TileMap`, the ideas carry over.

The workflow:

1. Add a `TileMapLayer` node to your scene.
2. In the Inspector, set its `Tile Set` to **New TileSet**, and set the tile size (for example 32 by 32 pixels).
3. Open the **TileSet** panel (bottom of the editor) and add your atlas image as a source (drag a PNG tilesheet there).
4. In the TileSet's Inspector, add a **Physics Layer** under *Physics Layers*. This makes the tiles able to have collision.
5. Select tiles and paint a collision polygon on each solid one.
6. Switch to the **TileMap** panel, pick tiles, and paint cells in the 2D view.

The TileMapLayer is itself a physics collider. A `CharacterBody2D` that collides with that layer will stop at your painted tiles.

In code you can read and write cells. This is a fragment for a script on a `TileMapLayer` that already has a `TileSet` whose source ID 0 is an atlas:

```gdscript
# Fragment: a script on a TileMapLayer node.
func _ready() -> void:
    set_cell(Vector2i(3, 2), 0, Vector2i(0, 0))   # cell (3,2), source 0, atlas tile (0,0)
    var cell: Vector2i = local_to_map(Vector2(100, 70))   # pixels to cell coordinates
    print(cell)
    erase_cell(Vector2i(3, 2))
```

Cells use `Vector2i` (whole numbers). We do not use tile maps in the lab, because they need an image file, but you will meet them again in your own projects.

RayCast2D: looking along a line

A `RayCast2D` shoots an invisible line from its position to `target_position` (local coordinates) and tells you the first thing it hits. Uses: ground checks, line of sight, enemy "vision", laser sights.

```
SightTest (Node2D)
├── Wall (StaticBody2D)            <- position (500, 300), CollisionShape2D (RectangleShape2D 20x200)
└── Player (CharacterBody2D)       <- position (200, 300), script: sight_player.gd
    ├── CollisionShape2D           <- RectangleShape2D, size (32, 32)
    └── RayCast2D                  <- Target Position (200, 0), Enabled on
```

Neither node draws a picture, so turn on **Debug > Visible Collision Shapes** to see the player and wall while you test. This scene uses the `move_left`, `move_right`, `move_up` and `move_down` actions from Chapter 6.

`sight_player.gd`, attached to **Player** (`CharacterBody2D`):

```gdscript
extends CharacterBody2D

@onready var ray: RayCast2D = $RayCast2D

func _ready() -> void:
    motion_mode = MOTION_MODE_FLOATING
    ray.add_exception(self)

func _physics_process(_delta: float) -> void:
    velocity = Input.get_vector("move_left", "move_right", "move_up", "move_down") * 200.0
    move_and_slide()
    if ray.is_colliding():
        var hit := ray.get_collider() as Node
        print("Ray sees ", hit.name, " at ", ray.get_collision_point())
```

Move the player toward the wall until the line touches it. The Output starts printing the wall's name and the hit point. The ray extends 200 pixels to the right of the player.

Useful ray functions: `is_colliding()`, `get_collider()`, `get_collision_point()`, `get_collision_normal()`, and `force_raycast_update()`. By default rays hit bodies but not areas; the properties `collide_with_bodies` and `collide_with_areas` change this.

You can also shoot a ray directly from code without a node:

```gdscript
# Fragment: inside a physics body's script.
func can_see(point: Vector2) -> bool:
    var space := get_world_2d().direct_space_state
    var query := PhysicsRayQueryParameters2D.create(global_position, point)
    query.exclude = [get_rid()]
    var result: Dictionary = space.intersect_ray(query)
    return result.is_empty()    # nothing in the way means we can see the point
```

`intersect_ray` returns a `Dictionary` with keys like `position`, `normal`, and `collider`. An **empty** dictionary means the ray hit nothing. Call a function like this from `_physics_process`, because the physics space is only safe to query there.

Debugging collisions

When "I walk through walls" or "the coin doesn't trigger" happens, **look at the shapes**. In the editor menu: **Debug > Visible Collision Shapes**. Turn it on, then run the game. Godot draws every shape in translucent blue (areas and bodies) so you can see their real size and place.

Checklist when a collision does not work:

1. Does the body have a `CollisionShape2D` *with a shape assigned*? (Look for the warning triangle.)
2. Is the shape `disabled`?
3. Do the layer and mask match (section 7.3)? Print them: `print(collision_layer, " ", collision_mask)`.
4. Is the shape in the right place (turn on Visible Collision Shapes)?
5. Is the body moving with `position` instead of `move_and_slide`? Setting `position` can teleport through walls.
6. Is the signal connected? Add a `print` at the top of the callback.

> **Try it:** Turn on **Debug > Visible Collision Shapes** and run the platformer test. You can see that the player's rectangle is slightly bigger than the visible drawn square if you made them different, and that the ledge's shape matches its picture. Turn it off again when done.

Lab 7: Coin Dash

Time: 60 to 90 minutes.

Goal: A top-down mini-game. You steer a blue square around an arena with walls. Eight gold coins sit around the arena. Walk into a coin to collect it. A score label counts up. When you collect all coins, "You win! Press R to play again" appears. Pressing R restarts the game at any time.

Step 1. Project settings

1. Create a new project named `CoinDash`.
2. **Input Map**: add `move_left` (A), `move_right` (D), `move_up` (W), `move_down` (S), and `restart` (R).
3. **Layer Names > 2D Physics**: name layer 1 `player`, layer 2 `world`, layer 3 `coins`.

Step 2. Plan the scenes

We make three scenes. Save them in `res://`.

- `wall.tscn`: a reusable rectangular wall.
- `coin.tscn`: a reusable coin.
- `main.tscn`: the arena (set it as the main scene).

Step 3. The Wall scene

```
Wall (StaticBody2D)            <- script: wall.gd
└── CollisionShape2D           <- shape is created by the script
```

In the Inspector for the **Wall** root: under **Collision**, set **Layer** to only `2`, and **Mask** to nothing (turn all squares off).

The editor will warn that `CollisionShape2D` has no shape. That is expected: the script supplies one. This lets each wall instance have its own `size`.

`wall.gd`, attached to **Wall** (`StaticBody2D`):

```gdscript
extends StaticBody2D

@export var size: Vector2 = Vector2(100, 100)

func _ready() -> void:
    var rect := RectangleShape2D.new()
    rect.size = size
    $CollisionShape2D.shape = rect

func _draw() -> void:
    draw_rect(Rect2(-size / 2.0, size), Color(0.35, 0.35, 0.5))
```

Why create the shape in code? Each instance has its own `size`. If we set the shape in the editor, all walls would share one shape resource and one size.

Save the scene as `wall.tscn`. (The script is not a tool script, so walls show up only when the game runs, not in the editor view. Use the position and size tables in Step 5 to place them.)

Step 4. The Coin scene

```
Coin (Area2D)                 <- script: coin.gd
└── CollisionShape2D          <- shape: CircleShape2D, radius 12
```

On the **Coin** root: **Layer** only `3`, **Mask** only `1` (the player). On `CollisionShape2D`, create a **New CircleShape2D** and set `Radius` to `12`.

`coin.gd`, attached to **Coin** (`Area2D`):

```gdscript
class_name Coin
extends Area2D

signal collected

func _ready() -> void:
    add_to_group("coins")
    body_entered.connect(_on_body_entered)

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        collected.emit()
        queue_free()

func _draw() -> void:
    draw_circle(Vector2.ZERO, 12.0, Color(1.0, 0.82, 0.1))
    draw_arc(Vector2.ZERO, 12.0, 0.0, TAU, 24, Color(0.7, 0.5, 0.0), 2.0)
```

How it works: when a body overlaps the coin, `_on_body_entered` checks that it is the player (a member of the `player` group, Chapter 5). If so, the coin announces `collected` and removes itself.

Save the scene as `coin.tscn`.

Step 5. The Main scene

Create the arena. Use a `Node2D` as the root named `Main`. Instance the other scenes under it with **Ctrl+Shift+A** (Instantiate Child Scene) or by dragging `wall.tscn` and `coin.tscn` from the FileSystem dock.

```
Main (Node2D)                     <- script: main.gd
├── Walls (Node2D)
│   ├── WallTop    (Wall)
│   ├── WallBottom (Wall)
│   ├── WallLeft   (Wall)
│   ├── WallRight  (Wall)
│   ├── WallA      (Wall)
│   ├── WallB      (Wall)
│   └── WallC      (Wall)
├── Coins (Node2D)
│   ├── Coin1 (Coin)
│   ├── Coin2 (Coin)
│   ├── Coin3 (Coin)
│   ├── Coin4 (Coin)
│   ├── Coin5 (Coin)
```

```
│   ├── Coin6 (Coin)
│   ├── Coin7 (Coin)
│   └── Coin8 (Coin)
├── Player (CharacterBody2D)       <- script: player.gd
│   └── CollisionShape2D
└── HUD (CanvasLayer)
    ├── ScoreLabel (Label)
    └── MessageLabel (Label)
```

Set each instance's **Position** (under Transform), and for walls also the exported **Size** (it appears in the Inspector under the script variables). Use Ctrl+D to duplicate the first wall and coin to save time.

Walls (window is 1152 by 648; the position is the *center* of each wall):

Node	Position	Size
WallTop	(576, 10)	(1152, 20)
WallBottom	(576, 638)	(1152, 20)
WallLeft	(10, 324)	(20, 648)
WallRight	(1142, 324)	(20, 648)
WallA	(400, 250)	(30, 300)
WallB	(750, 400)	(300, 30)
WallC	(800, 150)	(30, 200)

Coins:

Node	Position
Coin1	(100, 100)
Coin2	(250, 500)
Coin3	(576, 100)
Coin4	(550, 300)
Coin5	(900, 300)
Coin6	(1050, 100)
Coin7	(700, 500)
Coin8	(1000, 550)

Player: Position `(100, 300)`. Layer only `1`, Mask only `2`. Its `CollisionShape2D` gets a **New RectangleShape2D** with `Size` `(32, 32)`.

HUD: a `CanvasLayer` keeps the labels fixed on screen. Set `ScoreLabel` `Text` to `Coins: 0 / 8`. Leave `MessageLabel` text empty; the script fills it.

Now the scripts.

`player.gd`, attached to **Player** (`CharacterBody2D`):

```gdscript
extends CharacterBody2D

@export var speed: float = 280.0
```

```
func _ready() -> void:
    motion_mode = MOTION_MODE_FLOATING
    add_to_group("player")

func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    velocity = direction * speed
    move_and_slide()

func _draw() -> void:
    draw_rect(Rect2(-16, -16, 32, 32), Color(0.25, 0.6, 1.0))
```

`main.gd`, attached to **Main** (`Node2D`):

```
extends Node2D

@onready var player: CharacterBody2D = $Player
@onready var score_label: Label = $HUD/ScoreLabel
@onready var message_label: Label = $HUD/MessageLabel

var total_coins: int = 0
var collected: int = 0
var finished: bool = false

func _ready() -> void:
    score_label.add_theme_font_size_override("font_size", 28)
    score_label.position = Vector2(30, 30)
    message_label.add_theme_font_size_override("font_size", 40)
    message_label.position = Vector2(300, 280)
    message_label.text = ""

    for node in get_tree().get_nodes_in_group("coins"):
        var coin := node as Coin
        coin.collected.connect(_on_coin_collected)
        total_coins += 1
    _update_score()

func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("restart"):
        get_tree().reload_current_scene()

func _on_coin_collected() -> void:
    collected += 1
    _update_score()
    if collected >= total_coins:
        _win()

func _update_score() -> void:
    score_label.text = "Coins: %d / %d" % [collected, total_coins]

func _win() -> void:
    finished = true
    player.set_physics_process(false)
    message_label.text = "You win! Press R to play again"
```

Key ideas in `main.gd`:

- `get_nodes_in_group("coins")` collects every coin, so adding a ninth coin needs no code change.
- Each coin's `collected` signal is connected to `_on_coin_collected`.
- `reload_current_scene()` throws away the current scene and creates a fresh one. All variables reset, all coins return.
- `set_physics_process(false)` switches off `player.gd`'s `_physics_process`, so the player stops responding after the win.

- `"Coins: %d / %d" % [collected, total_coins]` is string formatting from Chapter 3.

> **Note:** `score_label` is a `Label`, and `add_theme_font_size_override` is how you change a label's text size from a script. Chapter 8 teaches themes in depth.

Step 6. Run it

Save everything, then press **F5**.

Test checklist

1. The player starts at the left; walls and coins are visible; the score reads `Coins: 0 / 8`.
2. WASD moves the square. Diagonal movement is not faster.
3. The square stops at the border walls and at the three inner walls, and it slides along them.
4. Touching a coin removes it, and the score goes up by one.
5. After the eighth coin, the message "You win! Press R to play again" appears and the player stops.
6. Pressing R at any time restarts: all coins return, score is 0 again.
7. Turn on **Debug > Visible Collision Shapes** and run. You can see the player's square, the wall rectangles, and the coin circles.

Expected result: a playable arena game with a score, a win message, and a working restart.

If something is wrong

Symptom	Likely cause
Player walks through walls	Player Mask does not include layer 2, or wall Layer is not 2
Coins never disappear	Coin Mask does not include layer 1, or the player is not in group `player` (check `player.gd`'s `_ready`)
Error `Invalid access to property 'collected'` in `main.gd`	A node in the `coins` group is not an instance of `coin.tscn` with `coin.gd` attached
Walls are invisible	`wall.gd` is not attached, or `size` is (0, 0)
Score stays `Coins: 0 / 0`	Group name mismatch: use exactly `coins`
Win never triggers	A coin sits inside a wall and cannot be reached; move it

Stretch goals

- Add a countdown `Timer` and lose if time runs out (Chapter 9 covers Timers).
- Add a moving enemy (a second `CharacterBody2D`) that restarts the game on contact, using an `Area2D` on its body.
- Use `randf_range` to place coins at random positions when the scene starts.

Chapter summary

- Physics runs on a fixed tick. Use `_physics_process` for anything involving bodies, velocity, or `move_and_slide`.

- Four main bodies: `StaticBody2D` (fixed), `CharacterBody2D` (code-controlled), `RigidBody2D` (engine-controlled), `Area2D` (detector that does not block).
- Every body needs a `CollisionShape2D` with a shape; resize shapes, never scale them.
- **Layer** is where an object lives; **mask** is what it looks for. Name layers in Project Settings and give each kind of object its own layer.
- In a `CharacterBody2D`, set `velocity` and call `move_and_slide()` (no arguments, no `delta` on velocity). Add `gravity * delta` to `velocity.y` for platformers and use `is_on_floor()` for jumps.
- `Area2D` signals (`body_entered`, `area_entered`, `mouse_entered`, `input_event`) drive pickups, triggers, and clickable zones.
- Godot 4.3+ uses `TileMapLayer`; `RayCast2D` reports what lies along a line.
- Debug collision problems with **Debug > Visible Collision Shapes**, then check shape, layer, mask, and signal connection.

Review questions

1. Why should movement of physics bodies go in `_physics_process` instead of `_process`?
2. Match each node to its best use: (a) `StaticBody2D`, (b) `CharacterBody2D`, (c) `RigidBody2D`, (d) `Area2D` with uses (1) a bouncing crate, (2) a coin pickup zone, (3) a player steered by the keyboard, (4) a solid wall.
3. A coin `Area2D` is on layer 3 with mask 1. The player `CharacterBody2D` is on layer 1 with mask 2. Will the coin's `body_entered` signal fire when the player touches it? Explain.
4. True or false: you must pass `velocity` into `move_and_slide()` as an argument.
5. In a platformer, should a jump set `velocity.y` to a positive or a negative number? Why?
6. What does `is_on_floor()` tell you, and which function updates it?
7. Which Area2D signal tells you when a `CharacterBody2D` has overlapped the area, and what argument does it give your function?
8. Why should you not set `position` every frame on a `RigidBody2D`? What should you use instead?
9. In Godot 4.3 and newer, which node replaces `TileMap` for painting tile levels?
10. Which editor menu option draws the collision shapes while the game runs?

Exercises

1. **Practice (easy): Name your layers.** In a new project, name layers 1 to 4 `player`, `world`, `coins`, `enemies`. Create a `StaticBody2D` and set its layer to `world` using the Inspector. Then write a script that prints `collision_layer` and `collision_mask` for it and explain the numbers (hint: layer 2 prints `2`, layer 3 prints `4`, because each layer is a bit).
2. **Practice (easy): Platformer tuning.** Take the platformer test from 7.5 and make three versions of the jump: floaty (low gravity), heavy (high gravity), and tight (high gravity and a high jump velocity). Write down the constants for each and describe how each feels.
3. **Apply (medium): Spike trap.** In the platformer test add an `Area2D` named `Spike` in red. When the player touches it, print "Ouch" and move the player back to its start position, `Vector2(200, 500)`. Use `body_entered`. (Hint: set `body.global_position` and reset `body.velocity = Vector2.ZERO` in the signal function, using a typed variable such as `var player := body as CharacterBody2D`.)
4. **Apply (medium): Ground check with a ray.** Add a `RayCast2D` pointing down by 30 pixels to the platformer player and print `ray.is_colliding()` each tick. Compare it to `is_on_floor()`. Which one changes first when you step off the ledge?

5. **Challenge (hard): Coin Dash upgrade.** Add (a) a `Timer`-free stopwatch in `main.gd` that adds `delta` in `_process` and shows the elapsed time under the score, stopping at the win; (b) a "best time" kept in a variable that survives restarts. Hint: `reload_current_scene` resets scene variables, so store the best time in an **autoload** (Chapter 5), such as `Stats.gd` with `var best_time: float = INF`.

Common mistakes

- **The player passes straight through walls** -> missing or misplaced `CollisionShape2D`, or layer and mask do not match -> assign a shape, check the layer numbers on both sides, and turn on **Debug > Visible Collision Shapes**.
- **The character moves extremely fast or extremely slow** -> multiplying `velocity` by `delta`, or forgetting that velocity is in pixels per second -> assign `velocity = direction * speed` with no `delta`, then call `move_and_slide()`.
- **The Output shows** `Can't change this state while flushing queries` -> changing an `Area2D` or a shape inside `body_entered` -> use `set_deferred("monitoring", false)` or `call_deferred(...)`.
- **The character falls forever or cannot jump** -> gravity is added but the floor has no shape, or `is_on_floor()` is false because `up_direction` or the floor shape is wrong -> check the floor's shape and that the player's `motion_mode` is the default grounded mode.
- **A coin never triggers** -> its mask does not contain the player's layer, or the signal is not connected -> check the layer table, and add a `print` at the top of the signal function.
- **The platform shape looks right but the collision is off** -> resizing the node with the scale handles instead of the shape -> reset `scale` to (1, 1) and resize the shape's `size` or `radius`.

Further reading

- Godot docs: *Physics introduction*: https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html
- Godot docs: *Using CharacterBody2D/3D*: https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html
- Godot docs: *Collision shapes (2D)*: https://docs.godotengine.org/en/stable/tutorials/physics/collision_shapes_2d.html
- Godot docs: *Using Area2D*: https://docs.godotengine.org/en/stable/tutorials/physics/using_area_2d.html
- Godot docs: *Ray-casting*: https://docs.godotengine.org/en/stable/tutorials/physics/ray-casting.html
- Godot docs: *Using TileMaps*: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilemaps.html
- Class reference: `CharacterBody2D`, `Area2D`, `RigidBody2D`, `TileMapLayer`, `RayCast2D` at https://docs.godotengine.org/en/stable/classes/

Chapter 8: User Interface

Learning objectives

By the end of this chapter you will be able to:

- Explain the difference between `Control` nodes and `Node2D` nodes and choose the right one for a menu, a HUD or a game object.
- Place and resize interface elements with anchors, offsets, size flags and container nodes so that a layout survives different window sizes.
- Build interactive screens using `Label`, `Button`, `TextureButton`, `LineEdit`, `CheckBox`, `OptionButton`, `HSlider`, `ProgressBar` and `RichTextLabel`, and react to their signals.
- Style a whole interface with a `Theme` resource and adjust single nodes with theme overrides.
- Make menus usable with keyboard and gamepad by managing focus.
- Build a HUD on a `CanvasLayer`, pause the game with `get_tree().paused` and `process_mode`, switch scenes with `change_scene_to_file`, and set up window stretching.

Key terms

- **Control**: the base class of every user interface node; it has a size, anchors, a theme and focus.
- **Anchor**: a number from 0 to 1 that says which point of the parent rectangle an edge of a Control is attached to.
- **Offset**: the pixel distance from a Control's edge to its anchor point.
- **Container**: a Control that arranges its children automatically and ignores their own anchors and positions.
- **Size flags**: settings that tell a container whether a child should fill, expand, shrink or center.
- **Theme**: a resource that stores the fonts, colors, icons, sizes and style boxes used by Controls.
- **Theme override**: a setting on one node that replaces the theme value for that node only.
- **Focus**: the state of being the one Control that receives keyboard or gamepad input such as "accept".
- **CanvasLayer**: a node that draws its children on a separate layer that does not move with the camera.
- **Process mode**: a node setting that decides whether the node runs while the game is paused.
- **BBCode**: a small markup language, such as `[b]bold[/b]`, used by `RichTextLabel` for formatted text.
- **Stretch mode**: a project setting that decides how the game picture is scaled when the window size changes.

8.1 Two families of nodes: Control and Node2D

So far you have used `Node2D` and its children: `Sprite2D`, `CharacterBody2D`, `Area2D`. A `Node2D` has a **position**, a **rotation** and a **scale**. That is enough for game objects that live in the game world, because the world has no edges and no "window size".

An interface is different. A menu must stay inside the window, a health bar must stay in the top-left corner, and a "Quit" button must be big enough to click. Interface nodes need to know about **size** and about **the space around them**. That is the job of the `Control` class.

Every interface node inherits from `Control`. In the Create Node dialog you can tell them apart by the green icons (Node2D family nodes have blue icons).

Feature	`Node2D`	`Control`
Position	`position` (a point)	`position` and `size` (a rectangle)
Layout help	none	anchors, offsets, size flags, containers
Look	none	`Theme`, theme overrides
Input	you write it	mouse filter, focus, built-in signals
Typical use	sprites, players, enemies	menus, HUD, dialog boxes

You can mix them in one project, and often you will. The usual pattern is a `Node2D` world with a UI layer on top (Section 8.6). Avoid putting a `Control` as a child of a `Node2D` and expecting anchors to work; anchors need a `Control` or a `CanvasLayer`/viewport parent to measure against.

> **Note:** A `Control` is a rectangle. Its top-left corner is `position`, and `size` says how far the rectangle extends right and down. Everything else in this chapter is about choosing those two values in a smart way.

Your first Control scene

1. Create a new project. In **Project > Project Settings > Display > Window** leave the size at 1152 x 648.
2. Create a scene with a **Control** root node, and rename it `Hello`.
3. Add a **Label** child. In the Inspector set `Text` to `Hello, Godot UI`.
4. Press **F6** (run current scene).

You see the text at the top-left corner. If you resize the window, the text does not move. It is attached to the top-left of its parent. The next section changes that.

8.2 Anchors, offsets and size flags

Anchors and offsets

A Control has four **anchors**: `anchor_left`, `anchor_top`, `anchor_right`, `anchor_bottom`. Each is a number between 0 and 1 that points to a place in the **parent's rectangle**. For example `anchor_left = 0.5` means "the left edge is attached to the horizontal middle of my parent".

Each edge also has an **offset** in pixels: `offset_left`, `offset_top`, `offset_right`, `offset_bottom`. The final edge position is:

```
edge = parent_size * anchor + offset    (measured from the parent's top-left corner)
```

When the parent resizes, the anchor point moves with it, and the offset stays the same. This is how a button can stay "20 pixels from the right edge".

Here are the three common cases:

Anchors	What happens when the parent grows
All four at the top-left (0, 0, 0, 0)	The Control keeps its size and stays at the corner

Anchors	What happens when the parent grows
Left and right both at 0.5 and top and bottom both at 0.5	The Control stays centered, same size
Left 0, right 1, top 0, bottom 1 (full rect)	The Control stretches to fill the parent

Anchor presets

You almost never type anchor numbers. Select a Control and click the **Anchors** button in the toolbar above the 2D viewport (the "Layout" menu in some versions). It shows presets such as *Top Left*, *Center*, *Top Wide*, *Full Rect*. Pick one and Godot sets all anchors and offsets for you.

In code you can do the same:

```
# Fragment: inside any Control script
func _ready() -> void:
    set_anchors_and_offsets_preset(Control.PRESET_FULL_RECT)
```

Use `set_anchors_and_offsets_preset` (it changes both anchors and offsets). The shorter `set_anchors_preset` changes only anchors and can leave odd offsets.

> **Try it:** In the `Hello` scene, select the `Label`. Open the Anchors menu and choose *Center*. Run with F6 and drag the window edges. The label stays near the middle. Now choose *Bottom Right* and run again. Notice that the label sits in the corner at every window size.

Minimum size

A Control can be told "never be smaller than this" with `custom_minimum_size` (Inspector: **Layout > Custom Minimum Size**). Buttons and labels also compute their own minimum from their text. The final size is never smaller than the larger of the two.

Size flags

Anchors work for loose Controls. Inside a **container** (next section) anchors are ignored and the container decides. You influence it with **size flags**, in the Inspector under **Layout > Container Sizing**:

Flag	Meaning
`Fill`	Use the whole slot given by the container
`Expand`	Ask the container for extra free space
`Fill + Expand` (`SIZE_EXPAND_FILL`)	Take free space and fill it. The most useful one
`Shrink Center`	Stay at minimum size, centered in the slot
`Shrink Begin` / `Shrink End`	Stay at minimum size, at the start or end

Horizontal and vertical flags are separate: `size_flags_horizontal` and `size_flags_vertical`. In code:

```
# Fragment: make a child take all spare horizontal space
child.size_flags_horizontal = Control.SIZE_EXPAND_FILL
```

If two children both use Expand, they share the spare space. The `Stretch Ratio` property sets the proportion (ratio 2 gets twice as much as ratio 1).

8.3 Containers

Positioning by hand is fragile. A **container** is a Control whose job is to place its children. You add children, set size flags, and the container does the math.

Container	What it does
`VBoxContainer`	Stacks children vertically
`HBoxContainer`	Places children side by side
`GridContainer`	Rows and columns; you set `columns`
`MarginContainer`	Adds empty space around a single child
`CenterContainer`	Keeps its child centered
`PanelContainer`	Draws a background panel and puts the child inside
`ScrollContainer`	Adds scroll bars when the child is bigger than the container

There are also flow containers (`HFlowContainer`, `VFlowContainer`) that wrap to the next line. You can nest containers as deep as you like. A normal menu is "CenterContainer > VBoxContainer > buttons".

Worked example: a vertical menu

Scene tree (save as `container_demo.tscn`):

```
ContainerDemo (Control)       <- anchors preset: Full Rect
└── CenterContainer           <- anchors preset: Full Rect
    └── VBoxContainer
        ├── Title (Label)     <- Text: "My Game"
        ├── Play (Button)     <- Text: "Play"
        └── Quit (Button)     <- Text: "Quit"
```

Steps:

1. Make the root `Control` and apply **Full Rect**.
2. Add `CenterContainer` as a child and apply **Full Rect** too (a container still needs its own size).
3. Add `VBoxContainer` inside it, then the three children.
4. Select `Title` and set **Horizontal Alignment** to *Center*.
5. Select `VBoxContainer`, open **Theme Overrides > Constants** and set `Separation` to `12`.

Expected result: a column of three items in the exact middle of the window at any size.

> **Why it works:** `CenterContainer` gives its child its minimum size and puts it in the middle. The `VBoxContainer` computes its minimum size from the widest child and the sum of the heights.

Grids and margins

A `GridContainer` with `columns = 3` and nine children gives a 3 x 3 grid, which is ideal for an inventory. `MarginContainer` has no inspector property for the margin size. It uses theme constants named `margin_left`, `margin_top`, `margin_right` and `margin_bottom`. Open **Theme Overrides > Constants** and type a number into each. In code:

```
# Fragment: give a MarginContainer 16 pixels on every side
margin_container.add_theme_constant_override("margin_left", 16)
margin_container.add_theme_constant_override("margin_top", 16)
margin_container.add_theme_constant_override("margin_right", 16)
margin_container.add_theme_constant_override("margin_bottom", 16)
```

A settings row

Rows of "label on the left, control on the right" are everywhere. Use an `HBoxContainer` per row:

```
HBoxContainer
├── VolumeLabel (Label)    <- Text: "Volume", custom minimum size x = 120
└── VolumeSlider (HSlider) <- Size flags horizontal: Fill + Expand
```

The slider takes the leftover width. The label has a fixed minimum so rows line up.

> **Warning:** Children of a container cannot be moved by dragging them in the viewport. Their `position` and anchors are overwritten by the container. If you cannot move a node, check whether its parent is a container.
>
> **Try it:** Change the `VBoxContainer` in the demo into a `GridContainer` (right-click > Change Type) with `columns = 2` and add four more buttons. Watch them fill a grid.

8.4 The common widgets

Label

`Label` shows text. Important properties: `text`, `horizontal_alignment`, `vertical_alignment`, `autowrap_mode`, `uppercase`. To make long text wrap you must set a width too (a minimum size or a container width), otherwise there is nothing to wrap against.

```
# Attach to: a Label node
extends Label

func _ready() -> void:
    autowrap_mode = TextServer.AUTOWRAP_WORD_SMART
    custom_minimum_size.x = 300.0
    text = "This sentence is long enough that it needs to wrap onto several lines."
```

Expected result: the text appears in a block 300 pixels wide.

Button

`Button` has `text`, `icon`, `disabled`, `flat` and `toggle_mode`. Its main signal is `pressed`. A button with `toggle_mode = true` stays down and emits `toggled(toggled_on: bool)`.

Connect in the editor (Node dock > Signals > double-click `pressed`) or in code. Code is easier to read in a book and easier to find later:

```
# Attach to: a Control that has a child Button named "GreetButton"
extends Control

@onready var greet_button: Button = $GreetButton

func _ready() -> void:
    greet_button.pressed.connect(_on_greet_pressed)

func _on_greet_pressed() -> void:
    print("Hello from the button")
```

Scene tree:

```
Main (Control)
└── GreetButton (Button)
```

Expected result: each click prints one line in the Output panel.

TextureButton

`TextureButton` is a button without a built-in look: you give it pictures. The inspector has slots `Texture Normal`, `Texture Pressed`, `Texture Hover`, `Texture Disabled` and `Texture Focused`. If you only fill `Texture Normal` it still works. For a button that scales, enable `Ignore Texture Size` and set `Stretch Mode`. Use it for icon buttons and picture-based menus. The `pressed` signal works the same as for `Button`.

LineEdit

`LineEdit` is a one-line text box. Useful members: `text`, `placeholder_text`, `max_length`, `secret` (hides characters), and the signals `text_changed(new_text)` and `text_submitted(new_text)` (when Enter is pressed).

```
# Attach to: a Control with a LineEdit child named "NameEdit" and a Label child named "Greeting"
extends Control

@onready var name_edit: LineEdit = $NameEdit
@onready var greeting: Label = $Greeting

func _ready() -> void:
    name_edit.placeholder_text = "Your name"
    name_edit.text_submitted.connect(_on_name_submitted)

func _on_name_submitted(new_text: String) -> void:
    greeting.text = "Welcome, %s!" % new_text
```

Scene tree:

```
Main (Control)
├── NameEdit (LineEdit)
└── Greeting (Label)
```

Move `Greeting` below `NameEdit` in the viewport (or put both in a `VBoxContainer` and update the paths to `$VBoxContainer/NameEdit`). Type a name, press Enter, and the label updates.

CheckBox, OptionButton and sliders

Node	Value to read	Signal
`CheckBox`	`button_pressed` (bool)	`toggled(toggled_on: bool)`
`OptionButton`	`selected` (index)	`item_selected(index: int)`
`HSlider` / `VSlider`	`value` (float)	`value_changed(value: float)`

`HSlider` has `min_value`, `max_value` and `step`. An `OptionButton` is a drop-down list. You fill it in the inspector (**Items**) or in code:

```
# Attach to: an OptionButton named "QualityOption"
extends OptionButton

func _ready() -> void:
    add_item("Low")
    add_item("Medium")
    add_item("High")
    select(1)
    item_selected.connect(_on_item_selected)

func _on_item_selected(index: int) -> void:
    print("Quality is now ", get_item_text(index))
```

Expected result: the box starts on "Medium"; choosing another entry prints its name.

> **Note:** When you want to set the value of a slider or checkbox from code without triggering its signal, use `set_value_no_signal(v)` on the slider and `set_pressed_no_signal(b)` on the checkbox. Otherwise your own set-up code may run the "changed" handler.

ProgressBar

`ProgressBar` shows how full something is. Set `min_value`, `max_value`, `value` and turn `show_percentage` on or off. In code you simply change `value`. For a smooth change, animate it with a Tween (Chapter 9):

```
# Fragment (needs a ProgressBar variable named health_bar)
func set_health(new_health: float) -> void:
    health_bar.value = new_health
```

Anything that has a "value between a minimum and a maximum" is a good fit: health, time left, loading progress, stamina.

RichTextLabel and BBCode

`RichTextLabel` can show **bold**, *italic*, colors, images, links and effects. You must turn on `bbcode_enabled`; then you write BBCode tags in `text`:

Tag	Result
`[b]...[/b]`	bold
`[i]...[/i]`	italic
`[color=red]...[/color]`	colored (names or `#ff8800`)

Tag	Result
`[center]...[/center]`	centered paragraph
`[wave amp=40 freq=5]...[/wave]`	text that waves
`[url=help]...[/url]`	clickable link; emits `meta_clicked`

```
# Attach to: a RichTextLabel
extends RichTextLabel

func _ready() -> void:
    bbcode_enabled = true
    fit_content = true
    text = "[center][b]Quest complete![/b]\nYou earned [color=gold]50 coins[/color].[/center]"
```

Expected result: a centered bold headline and a gold-colored number.

A famous use of `RichTextLabel` is the **typewriter effect**, using the `visible_ratio` property (0 hides everything, 1 shows everything):

```
# Attach to: a RichTextLabel
extends RichTextLabel

func _ready() -> void:
    bbcode_enabled = true
    text = "The old hotel [color=orange]creaked[/color] in the night."
    visible_ratio = 0.0
    var tween: Tween = create_tween()
    tween.tween_property(self, "visible_ratio", 1.0, 2.0)
```

Expected result: the sentence types itself over two seconds. (Tweens are explained in Chapter 9; for now treat it as "change this number over 2 seconds".)

Mouse filter: why my button does not click

Every Control has `mouse_filter` with three values:

- **Stop**: the control takes the click and nothing behind it gets it. This is the default for buttons.
- **Pass**: the control sees the click and also lets controls behind it see it.
- **Ignore**: the control is invisible to the mouse.

A full-screen `ColorRect` used as a dark background defaults to *Stop* and will block every button behind it. Set decorative Controls to **Ignore**.

8.5 Themes and theme overrides

If you changed the font size of ten buttons one by one, you would waste an afternoon and change your mind tomorrow. A **Theme** is a resource (a file ending in `.tres`) that stores the look of each Control type in one place: fonts, font sizes, colors, icons, constants and **style boxes** (the backgrounds drawn behind widgets).

How a theme is found

A Control looks for a value (say the font size of a `Button`) in this order:

1. A theme override on the node itself.
2. The `theme` property of the node.
3. The theme of its nearest ancestor Control that has one. The theme flows down the tree to children.
4. The project-wide custom theme, if you set one.
5. The built-in default theme.

So you assign one Theme to the root Control of a menu, and every child follows it.

Making a theme

1. In the FileSystem dock, right-click > **New Resource...** > **Theme**. Save it as `ui_theme.tres`.
2. Double-click it. The bottom panel shows the Theme editor.
3. Set **Default Font Size** (in the Theme's inspector) to `24`.
4. Click **Add Type** in the Theme editor, pick `Button`, and edit its `normal`, `hover` and `pressed` style boxes: choose **New StyleBoxFlat**, set a background color, corner radius and content margins.
5. Select the root Control of your menu and drag `ui_theme.tres` into its **Theme** slot.

To make the theme apply to the whole game, set **Project Settings > GUI > Theme > Custom** to your file.

> **Why it works:** The theme is only data. Buttons ask "what does a Button's `normal` style look like?" every time they draw, and the answer comes from the first place in the list above that has one.

Theme overrides

On a single node, **Theme Overrides** in the Inspector let you change just that node. Typical use: a red "Delete" button or a bigger title label. In code, the methods start with `add_theme_` and end with `_override`:

```gdscript
# Attach to: a Label named "Title"
extends Label

func _ready() -> void:
    add_theme_font_size_override("font_size", 48)
    add_theme_color_override("font_color", Color.GOLD)
    add_theme_color_override("font_outline_color", Color.BLACK)
    add_theme_constant_override("outline_size", 6)
```

Expected result: a big gold title with a black outline.

Style boxes work the same way:

```gdscript
# Attach to: a Button
extends Button

func _ready() -> void:
    var style: StyleBoxFlat = StyleBoxFlat.new()
    style.bg_color = Color("#2b3a67")
    style.set_corner_radius_all(10)
    style.set_content_margin_all(12.0)
    add_theme_stylebox_override("normal", style)
```

Expected result: a navy rounded button. It will look the same when hovered unless you also override `"hover"`.

> **Try it:** Create a theme with a larger default font size and a rounded `normal` style for Buttons. Assign it to the root of your `container_demo` scene. Every button now looks the same without touching the buttons.

Keep it consistent

Pick three colors (background, accent, text) and one font size scale (for example 20, 28, 48). Put them in the theme. When you later change your mind, you change one file.

8.6 Focus, keyboard and gamepad navigation

A mouse is only one way to play. Keyboard and gamepad users move a **focus** highlight from button to button with the directional keys or stick and press "accept" (Enter, Space, or the A button).

Godot gives you this for free as long as you do three things:

1. **Give one Control focus when the screen opens.** Call `grab_focus()` on the first button. Without this, a keyboard user sees nothing selected and the arrow keys appear dead.
2. **Make sure `focus_mode` is not "None".** Buttons default to *All*. Decorative Controls should be *None*.
3. **Use the built-in actions.** `ui_up`, `ui_down`, `ui_left`, `ui_right`, `ui_accept`, `ui_cancel`, `ui_focus_next` already exist in the Input Map. Godot moves focus to the nearest neighbor in the pressed direction.

```gdscript
# Fragment: in the _ready() of a menu script
func _ready() -> void:
    $VBoxContainer/PlayButton.grab_focus()
```

When the automatic order is wrong, set the **Focus** section in the Inspector: `Focus Neighbor Top/Bottom/Left/Right` take a node path. In code: `button.focus_neighbor_bottom = button.get_path_to(other_button)`.

The focused button is drawn with a `focus` style box (a thin outline in the default theme). If you restyle buttons with a custom theme, **always style `focus` too**, or players will not see where they are.

> **Try it:** Run your `container_demo` with F6. Click once on empty space, then press Tab and the arrow keys. Add `grab_focus()` for the Play button and compare.
>
> **Warning:** A hidden Control cannot hold focus. When you `show()` a menu, call `grab_focus()` *after* showing it.

8.7 CanvasLayer, pausing, scene switching and stretching

CanvasLayer for HUDs

If your game has a `Camera2D` that follows the player, a `Label` placed in the world scrolls away with everything else. A `CanvasLayer` draws its children in screen space, unaffected by cameras. Anything on it stays put. Its `layer` number decides the draw order: a higher number is drawn on top (the default world is layer 0). Typical numbers: HUD at 1, pause menu at 10.

```
Game (Node2D)
├── World stuff...
├── Hud (CanvasLayer)        <- layer 1
│   └── ScoreLabel (Label)
└── PauseMenu (CanvasLayer)  <- layer 10
```

A Control that is a child of a `CanvasLayer` uses the whole window as its parent rectangle, so anchors work exactly as in a menu scene.

Pausing the game

`get_tree().paused = true` freezes the game: `_process`, `_physics_process`, physics, animations and timers stop. To decide which nodes still run, every node has a **process mode** (Inspector: **Node > Process > Mode**, in code `process_mode`):

Mode	Behavior
`Inherit` (default)	Use the parent's mode
`Pausable`	Runs only when not paused
`When Paused`	Runs only while paused
`Always`	Runs always
`Disabled`	Never runs

Every node defaults to *Inherit*, and the tree's root acts as *Pausable*, so by default everything pauses. Your pause menu must be set to **Always** (or *When Paused*), or it freezes together with the game and cannot receive a click to resume. In code the constants are `Node.PROCESS_MODE_ALWAYS`, `Node.PROCESS_MODE_PAUSABLE`, `Node.PROCESS_MODE_WHEN_PAUSED`, and so on.

```gdscript
# Attach to: a CanvasLayer named "PauseMenu" (set Process > Mode to Always in the Inspector,
# or uncomment the line in _ready)
extends CanvasLayer

func _ready() -> void:
    # process_mode = Node.PROCESS_MODE_ALWAYS
    pass

func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("ui_cancel"):
        get_tree().paused = not get_tree().paused
```

Expected result (when the pause menu is part of a running game): pressing Escape alternately freezes and unfreezes the game.

> **Warning:** `paused` is a property of the whole `SceneTree`, not of a scene. If you change scenes while paused, the next scene starts paused too. Always set `get_tree().paused = false` before leaving.

Switching scenes

```gdscript
get_tree().change_scene_to_file("res://game.tscn")
```

This unloads the current scene and loads the one at that path. It happens at the end of the current frame, so code after the call in the same function still runs once. Related calls:

Call	Use
`get_tree().change_scene_to_file(path)`	Switch to a scene by path
`get_tree().change_scene_to_packed(packed_scene)`	Switch to an already loaded `PackedScene`
`get_tree().reload_current_scene()`	Restart the current scene
`get_tree().quit()`	Close the game

`change_scene_to_file` returns an `Error` code; if the path is wrong it returns `ERR_CANT_OPEN` and also prints an error. Copy the path with right-click > **Copy Path** in the FileSystem dock to avoid typing mistakes.

Window stretch settings

By default, when the player drags the window larger, the game keeps its design size and shows extra empty space. Fix that in **Project > Project Settings > Display > Window**:

Setting	Recommended value	What it does
Size > Viewport Width / Height	1152 x 648 (or your design size)	The size you design for
Stretch > Mode	`canvas_items`	Scales the whole interface and 2D world up crisply to fit
Stretch > Aspect	`keep`	Keeps the shape and adds black bars if the window has a different shape

`canvas_items` re-renders at the larger size, so text and vector shapes stay sharp. Use `viewport` instead for pixel-art games where you want big chunky pixels. Turn on **Advanced Settings** (top-right of the Project Settings window) if you do not see the Stretch section.

> **Try it:** Set the Stretch Mode to `canvas_items` and run your `Hello` scene. Drag the window to be twice as large. The text now scales instead of staying small in the corner.

Lab 8: Main menu, options and HUD for a small game

Time: 60 to 90 minutes.

Goal: build a small coin-grabbing game ("Coin Dash") with a main menu, an options screen with a volume slider and a fullscreen checkbox, a HUD, and a pause menu. You will use ideas from Chapters 1 to 7 (scenes, signals, groups, `CharacterBody2D`, `Area2D`).

Project files you will create:

```
res://
├── coin.tscn / coin.gd
├── player.gd
├── hud.tscn / hud.gd
├── options_menu.tscn / options_menu.gd
├── pause_menu.tscn / pause_menu.gd
```

```
├── main_menu.tscn / main_menu.gd
└── game.tscn / game.gd
```

Step 0: project settings

1. Create a project named `CoinDashUI` (Forward+ or Mobile or Compatibility renderer are all fine).
2. In **Project Settings > Display > Window**: Viewport Width `1152`, Height `648`, Stretch Mode `canvas_items`, Aspect `keep`.
3. Remember about **unique names**: right-click a node in the Scene dock and choose **Access as Unique Name**. A `%` appears beside the name. In the script of the same scene you can write `%Name` to reach that node, no matter how deep it sits.

Step 1: the coin

Scene tree for `coin.tscn`:

```
Coin (Area2D)              <- script coin.gd
├── Sprite2D               <- Texture: res://icon.svg, Scale (0.1, 0.1), Self Modulate: gold
└── CollisionShape2D       <- Shape: CircleShape2D, Radius 13
```

`coin.gd`:

```gdscript
class_name Coin
extends Area2D

signal collected

func _ready() -> void:
    body_entered.connect(_on_body_entered)

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        collected.emit()
        queue_free()
```

Leave the coin and the player on the default collision layer 1 and mask 1; this lab has no walls, so they will see each other.

Step 2: the player script

The player lives directly in the game scene (Step 6). Create `player.gd` now:

```gdscript
extends CharacterBody2D

@export var speed: float = 320.0

func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")
    velocity = direction * speed
    move_and_slide()
    position = position.clamp(Vector2(16.0, 16.0), Vector2(1136.0, 632.0))
```

The arrow keys are the built-in `ui_` actions, so you do not need to edit the Input Map.

Step 3: the HUD

Scene tree for `hud.tscn`:

```
Hud (CanvasLayer)               <- script hud.gd, Layer: 1
├── TopBar (MarginContainer)    <- Anchors: Top Wide. Theme constants margin_* = 16
│   └── HBoxContainer
│       ├── ScoreLabel (Label)  <- unique name, Text: "Score: 0"
│       └── TimeBar (ProgressBar) <- unique name, Size Flags Horizontal: Fill + Expand,
│                                    Show Percentage: off, Max Value 30, Value 30
└── CenterBox (CenterContainer) <- Anchors: Full Rect, Mouse Filter: Ignore
    └── MessageLabel (Label)    <- unique name, Visible: off, font size 48
```

`hud.gd`:

```gdscript
class_name GameHud
extends CanvasLayer

@onready var score_label: Label = %ScoreLabel
@onready var time_bar: ProgressBar = %TimeBar
@onready var message_label: Label = %MessageLabel

func set_score(value: int) -> void:
    score_label.text = "Score: %d" % value

func set_time(left: float, total: float) -> void:
    time_bar.max_value = total
    time_bar.value = left

func show_message(text: String) -> void:
    message_label.text = text
    message_label.show()
```

Step 4: the options screen

Scene tree for `options_menu.tscn`:

```
OptionsMenu (Control)                <- script options_menu.gd, Anchors: Full Rect
├── Dim (ColorRect)                  <- Anchors: Full Rect, Color (0, 0, 0, 0.8), Mouse Filter: Stop
└── CenterContainer                  <- Anchors: Full Rect
    └── PanelContainer
        └── MarginContainer          <- theme constants margin_* = 24
            └── VBoxContainer        <- Theme constant Separation = 16
                ├── Title (Label)    <- Text: "Options", centered
                ├── HBoxContainer
                │   ├── VolumeLabel (Label)    <- Text: "Volume", Custom Minimum Size x = 120
                │   └── VolumeSlider (HSlider) <- unique name; Min 0, Max 1, Step 0.01,
                │                                 Size Flags Horizontal: Fill + Expand,
                │                                 Custom Minimum Size x = 220
                ├── FullscreenCheck (CheckBox) <- unique name, Text: "Fullscreen"
                └── BackButton (Button)        <- unique name, Text: "Back"
```

`options_menu.gd`:

```gdscript
class_name OptionsMenu
extends Control

signal closed

@onready var volume_slider: HSlider = %VolumeSlider
@onready var fullscreen_check: CheckBox = %FullscreenCheck
@onready var back_button: Button = %BackButton
```

```
func _ready() -> void:
    var bus: int = AudioServer.get_bus_index("Master")
    volume_slider.set_value_no_signal(db_to_linear(AudioServer.get_bus_volume_db(bus)))
    var is_fullscreen: bool = DisplayServer.window_get_mode() == DisplayServer.WINDOW_MODE_FULLSCREEN
    fullscreen_check.set_pressed_no_signal(is_fullscreen)

    volume_slider.value_changed.connect(_on_volume_changed)
    fullscreen_check.toggled.connect(_on_fullscreen_toggled)
    back_button.pressed.connect(close)
    volume_slider.grab_focus()

func _on_volume_changed(value: float) -> void:
    var bus: int = AudioServer.get_bus_index("Master")
    AudioServer.set_bus_volume_db(bus, linear_to_db(value))

func _on_fullscreen_toggled(toggled_on: bool) -> void:
    if toggled_on:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_FULLSCREEN)
    else:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_WINDOWED)

func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("ui_cancel"):
        get_viewport().set_input_as_handled()
        close()

func close() -> void:
    closed.emit()
    queue_free()
```

The slider works in the range 0 to 1 (what people understand), and `linear_to_db` converts it to decibels (what the audio system uses). A value of 0 gives silence. You have no sound yet, so you will hear the effect in Chapter 9; for now check that the numbers move.

Step 5: the main menu

Scene tree for `main_menu.tscn`:

```
MainMenu (Control)                    <- script main_menu.gd, Anchors: Full Rect
├── Background (ColorRect)            <- Anchors: Full Rect, Color: dark blue, Mouse Filter: Ignore
└── CenterContainer                   <- Anchors: Full Rect
    └── VBoxContainer                 <- Theme constant Separation = 14
        ├── Title (Label)             <- Text: "Coin Dash", font size 64 (override), centered
        ├── StartButton (Button)      <- unique name, Text: "Start"
        ├── OptionsButton (Button)    <- unique name, Text: "Options"
        └── QuitButton (Button)       <- unique name, Text: "Quit"
```

Set a minimum width of 220 on each button so they look even (or let the `VBoxContainer`'s widest child decide).

`main_menu.gd`:

```
extends Control

const GAME_PATH: String = "res://game.tscn"
const OPTIONS_SCENE: PackedScene = preload("res://options_menu.tscn")

@onready var start_button: Button = %StartButton
@onready var options_button: Button = %OptionsButton
@onready var quit_button: Button = %QuitButton

func _ready() -> void:
    start_button.pressed.connect(_on_start_pressed)
```

```
    options_button.pressed.connect(_on_options_pressed)
    quit_button.pressed.connect(_on_quit_pressed)
    start_button.grab_focus()

func _on_start_pressed() -> void:
    get_tree().change_scene_to_file(GAME_PATH)

func _on_options_pressed() -> void:
    var options: OptionsMenu = OPTIONS_SCENE.instantiate()
    options.closed.connect(_on_options_closed)
    add_child(options)

func _on_options_closed() -> void:
    start_button.grab_focus()

func _on_quit_pressed() -> void:
    get_tree().quit()
```

`preload` loads the scene while the script is compiled, and `instantiate()` makes a fresh copy each time. Because `OptionsMenu` is a `class_name`, `options.closed` is checked by the editor.

Set `main_menu.tscn` as the main scene: **Project Settings > Application > Run > Main Scene**, or right-click the scene tab > **Set as Main Scene**.

Step 6: the pause menu

Scene tree for `pause_menu.tscn`:

```
PauseMenu (CanvasLayer)              <- script pause_menu.gd, Layer: 10, Process > Mode: Always
└── Root (Control)                   <- unique name, Anchors: Full Rect, Mouse Filter: Ignore
    ├── Dim (ColorRect)              <- Anchors: Full Rect, Color (0, 0, 0, 0.6)
    └── CenterContainer              <- Anchors: Full Rect
        └── VBoxContainer
            ├── Title (Label)        <- Text: "Paused", centered
            ├── ResumeButton (Button)   <- unique name, Text: "Resume"
            ├── OptionsButton (Button)  <- unique name, Text: "Options"
            └── ToMenuButton (Button)   <- unique name, Text: "Main menu"
```

The script hides `Root` at start, so you can leave it visible while you design the layout.

> **Note:** We avoid the node name `MenuButton` because `MenuButton` is also a built-in Godot class. A node name could be anything, but the same word for two things is confusing.

`pause_menu.gd`:

```
extends CanvasLayer

const MAIN_MENU_PATH: String = "res://main_menu.tscn"
const OPTIONS_SCENE: PackedScene = preload("res://options_menu.tscn")

@onready var root: Control = %Root
@onready var resume_button: Button = %ResumeButton
@onready var options_button: Button = %OptionsButton
@onready var to_menu_button: Button = %ToMenuButton

func _ready() -> void:
    root.hide()
    resume_button.pressed.connect(resume)
    options_button.pressed.connect(_on_options_pressed)
    to_menu_button.pressed.connect(_on_to_menu_pressed)
```

```
func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("ui_cancel"):
        get_viewport().set_input_as_handled()
        if get_tree().paused:
            resume()
        else:
            pause()

func pause() -> void:
    get_tree().paused = true
    root.show()
    resume_button.grab_focus()

func resume() -> void:
    get_tree().paused = false
    root.hide()

func _on_options_pressed() -> void:
    var options: OptionsMenu = OPTIONS_SCENE.instantiate()
    options.closed.connect(resume_button.grab_focus)
    add_child(options)

func _on_to_menu_pressed() -> void:
    get_tree().paused = false
    get_tree().change_scene_to_file(MAIN_MENU_PATH)
```

The options menu is added as a child of the pause menu, so it inherits the *Always* mode and works while the game is frozen. It is also later in the tree than `Root`, so it draws on top and receives the Escape key first.

Step 7: the game scene

Scene tree for `game.tscn`:

```
Game (Node2D)                       <- script game.gd
├── Player (CharacterBody2D)        <- script player.gd, Position (576, 324), Group: player
│   ├── Sprite2D                    <- Texture: res://icon.svg, Scale (0.25, 0.25)
│   └── CollisionShape2D            <- RectangleShape2D, Size (32, 32)
├── Coins (Node2D)
├── Hud                             <- instance of hud.tscn
└── PauseMenu                       <- instance of pause_menu.tscn
```

To add the group: select `Player`, open the **Node** dock (next to Inspector), the **Groups** tab, type `player` and press Add. Use **Instantiate Child Scene** (the chain icon) to add `Hud` and `PauseMenu`.

`game.gd`:

```
extends Node2D

const COIN_SCENE: PackedScene = preload("res://coin.tscn")
const MAIN_MENU_PATH: String = "res://main_menu.tscn"
const AREA_SIZE: Vector2 = Vector2(1152.0, 648.0)
const ROUND_TIME: float = 30.0
const START_COINS: int = 5

var score: int = 0
var time_left: float = ROUND_TIME
var round_over: bool = false

@onready var coins: Node2D = $Coins
@onready var hud: GameHud = $Hud

func _ready() -> void:
    hud.set_score(score)
```

```
    hud.set_time(time_left, ROUND_TIME)
    for _i in START_COINS:
        spawn_coin()

func _process(delta: float) -> void:
    if round_over:
        return
    time_left = maxf(time_left - delta, 0.0)
    hud.set_time(time_left, ROUND_TIME)
    if time_left <= 0.0:
        _end_round()

func spawn_coin() -> void:
    var coin: Coin = COIN_SCENE.instantiate()
    coin.position = Vector2(
        randf_range(40.0, AREA_SIZE.x - 40.0),
        randf_range(60.0, AREA_SIZE.y - 40.0)
    )
    coin.collected.connect(_on_coin_collected)
    coins.add_child(coin)

func _on_coin_collected() -> void:
    score += 1
    hud.set_score(score)
    # Adding a physics node while physics is busy gives an error, so wait one moment.
    spawn_coin.call_deferred()

func _end_round() -> void:
    round_over = true
    hud.show_message("Time's up!  Score: %d" % score)
    await get_tree().create_timer(2.5).timeout
    get_tree().change_scene_to_file(MAIN_MENU_PATH)
```

`await get_tree().create_timer(2.5).timeout` simply waits 2.5 seconds (`await` is from Chapter 5; timers get a full treatment in Chapter 9).

Step 8: test

Run the project (**F5**). Check each item:

1. The main menu shows with "Start" already highlighted. Press Enter: the game starts.
2. Arrow keys move the Godot-icon player. Touching a gold coin raises the score and a new coin appears.
3. The time bar shrinks. At zero, a "Time's up" message shows, then the menu returns.
4. Press **Escape** during play: the game freezes (the bar stops!), the pause menu appears, "Resume" has focus.
5. In the pause menu choose **Options**, drag the slider, tick Fullscreen, press **Escape**: only the options close. Press Escape again: the game resumes.
6. **Main menu** returns to the menu and the next game starts unpaused.
7. Drag the window bigger: everything scales.
8. **Quit** closes the game.

Expected result: a small but complete game flow: menu -> game with HUD -> pause -> options -> back to the menu, all working with mouse and keyboard.

Chapter summary

- `Control` nodes are for interface; they have size, anchors, a theme, and focus. `Node2D` nodes are for the game world.

- Anchors attach an edge to a fraction of the parent; offsets are pixel distances from that anchor point. Use the Anchors presets menu.
- Containers (`VBox`, `HBox`, `Grid`, `Margin`, `Center`, `Panel`) arrange children for you; children use size flags such as Fill + Expand.
- Widgets report events through signals: `pressed`, `toggled`, `value_changed`, `item_selected`, `text_submitted`.
- A `Theme` resource styles the whole interface; theme overrides change a single node.
- Call `grab_focus()` on the first control of every screen so keyboard and gamepad players can navigate.
- Put the HUD on a `CanvasLayer`; pause with `get_tree().paused`; give the pause menu `process_mode` Always; unpause before changing scenes.
- Set Stretch Mode to `canvas_items` so the game scales with the window.

Review questions

1. What is the main difference between a `Node2D` and a `Control`?
2. A button has anchors all set to 1.0 (bottom-right) and `offset_left = -120`. Describe where its left edge is when the window grows wider.
3. Which size flag makes a child take up the remaining space of an `HBoxContainer`?
4. True or false: you can freely drag a `Button` that is a child of a `VBoxContainer` to any place in the viewport.
5. Which signal does a `CheckBox` emit when the player ticks it, and what does its parameter contain?
6. You made a full-screen dark `ColorRect` as a background, and now none of the buttons react to clicks. What property should you change and to which value?
7. Name the order in which a Control looks for a theme value, starting with the most specific.
8. Why must you call `grab_focus()` when a menu opens?
9. After `get_tree().paused = true`, your pause menu buttons do not respond. What setting is wrong?
10. Choose: which Stretch Mode keeps text and vector shapes crisp when the window gets larger? (a) `disabled` (b) `canvas_items` (c) `viewport`

Exercises

1. **Practice (easy):** Make a scene with a `GridContainer` of 3 columns and 9 buttons labelled 1 to 9, centered in the window. Print the label of the pressed button with one shared handler (hint: `button.pressed.connect(_on_number_pressed.bind(i))`).
2. **Practice (easy):** Add a `RichTextLabel` to the main menu that shows a one-line credit with a colored word and a typewriter effect.
3. **Apply (medium):** Make a theme file `ui_theme.tres` with rounded buttons (normal, hover, pressed and focus style boxes) and a bigger default font size. Apply it to the root of all three menu scenes and compare the look before and after.
4. **Apply (medium):** Add an `OptionButton` named `ResolutionOption` to the options menu with the entries "1152 x 648" and "1920 x 1080", and make it call `DisplayServer.window_set_size(Vector2i(w, h))`.
5. **Challenge (hard):** Add a scene transition: a black `ColorRect` on a high `CanvasLayer` in an autoload script that fades in, changes the scene, and fades out. The menu's Start button should use it instead of calling `change_scene_to_file` directly.

Common mistakes

- **The label or button sits in the top-left corner and ignores the window size.** -> No anchor preset or container. -> Apply an anchor preset or put it inside a container.
- **A child inside a container will not move or resize as set.** -> The container overrides position and size. -> Use size flags and custom minimum size instead.
- **Nothing happens when I click a button.** -> Another Control (often a `ColorRect`) is in front with *Mouse Filter: Stop*. -> Set it to Ignore.
- **The pause menu appears but is frozen.** -> Its `process_mode` is still Inherit/Pausable. -> Set it to Always.
- **The next game starts frozen after leaving through the pause menu.** -> `get_tree().paused` was still true. -> Set it to false before `change_scene_to_file`.
- **Keyboard arrows do nothing in the menu.** -> No control has focus. -> Call `grab_focus()` on the first button, after the menu is visible.
- **Error "Invalid access to property or key" on a `$Node` path.** -> The path does not match the scene tree (maybe the node moved into a container). -> Use unique names (`%Name`) so the path survives re-arranging.

Further reading

- Godot docs, User Interface: https://docs.godotengine.org/en/stable/tutorials/ui/index.html
- Size and anchors: https://docs.godotengine.org/en/stable/tutorials/ui/size_and_anchors.html
- Using containers: https://docs.godotengine.org/en/stable/tutorials/ui/gui_containers.html
- Control node gallery: https://docs.godotengine.org/en/stable/tutorials/ui/control_node_gallery.html
- GUI skinning (themes): https://docs.godotengine.org/en/stable/tutorials/ui/gui_skinning.html
- BBCode in RichTextLabel: https://docs.godotengine.org/en/stable/tutorials/ui/bbcode_in_richtextlabel.html
- Pausing games: https://docs.godotengine.org/en/stable/tutorials/scripting/pausing_games.html
- Multiple resolutions: https://docs.godotengine.org/en/stable/tutorials/rendering/multiple_resolutions.html

Chapter 9: Animation and Audio

Learning objectives

By the end of this chapter you will be able to:

- Build animations in an `AnimationPlayer` using property tracks, call-method tracks, looping and a RESET track, and control them from code.
- Play frame-by-frame animation with `AnimatedSprite2D` and `SpriteFrames`.
- Animate properties from a script with a `Tween`: easing, chaining, parallel steps and the `finished` signal.
- Choose between a `Timer` node and `get_tree().create_timer()` for delays and repeating events.
- Create simple effects with `CPUParticles2D` and a screen shake.
- Play sounds and music with `AudioStreamPlayer` and `AudioStreamPlayer2D`, route them through audio buses, and control volume with `AudioServer`, `linear_to_db` and fades.

Key terms

- **Animation**: a resource that stores keyframes for properties over time.
- **Track**: one row of an animation that controls a single property or method call.
- **Keyframe**: a value placed at a point in time on a track; Godot fills in the values between keyframes.
- **Interpolation**: how Godot computes the in-between values (linear, cubic, nearest).
- **SpriteFrames**: a resource holding named lists of pictures for `AnimatedSprite2D`.
- **Tween**: an object that changes a property from one value to another over time, created by code.
- **Easing**: a rule that makes a change start slowly, end slowly, or bounce, instead of moving at constant speed.
- **Particles**: many tiny, short-lived sprites used for sparks, smoke and dust.
- **Audio bus**: a channel in the audio system; sounds are routed to a bus, and the bus has its own volume and effects.
- **Decibel (dB)**: the unit Godot uses for volume; 0 dB is "unchanged", negative values are quieter.
- **Stream**: an audio resource (a sound file) that a player node plays.

9.1 AnimationPlayer: animating anything

The idea

A game without motion feels dead. `AnimationPlayer` is Godot's general animation tool. It does not only move sprites. It can change **any property of any node in the same scene**: position, rotation, color, `visible`, a light's energy, a label's text, even a call to one of your functions.

The core idea is simple:

1. An `AnimationPlayer` node holds one or more **animations**.
2. An animation has a **length** (in seconds) and a list of **tracks**.
3. Each track targets a node and a property, and has **keyframes** at certain times.
4. When you play the animation, Godot moves along the timeline and sets the properties to the interpolated values.

Making your first animation

Scene tree:

```
Bouncer (Node2D)
├── Sprite2D                <- Texture: res://icon.svg
└── AnimationPlayer
```

Steps:

1. Select `AnimationPlayer`. The **Animation** panel opens at the bottom.
2. Click **Animation** (the dropdown) > **New**, and name it `bounce`.
3. Set **Length** to `1.0` (the number box near the top right of the panel).
4. Select the `Sprite2D`. In the Inspector, click the small **key icon** beside `Position`. Godot asks to create a track; confirm. A keyframe appears at time 0.
5. Drag the time cursor (the red line in the timeline ruler) to `0.5`. Change `Position` to `(0, -80)` and click the key icon again.
6. Move to `1.0`, set `Position` back to `(0, 0)`, click the key icon.
7. Turn on the **loop** button (the circle-arrows icon near the length box).
8. Press the **play** button in the panel to preview.

Expected result: the logo goes up and comes back down forever, in the editor preview.

> **Note:** The key icon in the Inspector is the quickest way to create property tracks. You can also press **Add Track** and choose *Property Track*, then pick the node and the property.

Interpolation and easing on a track

At the right end of each track there is a small icon for **Interpolation**. *Linear* moves at a constant speed. *Cubic* is smooth and makes motion look natural. *Nearest* jumps from key to key with no in-between (used for things like `visible` or `frame`). Click the icon to change it. For booleans and integers Godot chooses sensible defaults.

Track types you will use

Track type	What it changes	Example
Property	A property on a node	`position`, `modulate`, `visible`, `scale`
Call Method	Calls a function on a node at a given time	Play a footstep sound, spawn an effect
Audio Playback	Starts and stops sounds	A cutscene with music
Animation Playback	Starts another animation	Combining animations
Bezier Curve	A property with custom curves	Fine control of one number

The **Call Method track** is very useful for events at exact moments. Add a *Call Method Track*, choose a node, then right-click the track at a time position and pick **Insert Key**. A list of that node's functions appears; choose one (it must exist in the node's script). Now at that moment, your function runs. You will use this in the lab.

The RESET track

When you scrub an animation in the editor, the properties really change. If you save the scene while the sprite is at the end of the animation, the scene is saved in the wrong pose. That is why Godot supports a special animation named **RESET**. It holds a single frame with the *default* values for every property that other animations change. The editor uses it to put the nodes back into their resting pose when you save or reopen the scene, so the node looks right.

To create it: finish your animation, then open the **Edit** menu in the animation panel and choose **Create RESET Track(s) From Animation**. Godot adds the animation `RESET` using the first keyframe values. Keep the RESET track up to date when you add new tracks.

> **Why it works:** The first keyframe usually shows the resting pose. RESET guarantees that "not playing anything" means "resting pose", even if you were scrubbing at the last frame when you saved.

Controlling animations from code

```gdscript
# Attach to: Node2D "Bouncer" (see the scene tree above)
extends Node2D

@onready var anim: AnimationPlayer = $AnimationPlayer

func _ready() -> void:
    anim.animation_finished.connect(_on_animation_finished)
    anim.play("bounce")

func _on_animation_finished(anim_name: StringName) -> void:
    print("Finished: ", anim_name)
```

Notes about this code:

- If the animation loops, `animation_finished` does not fire (it never finishes). Turn looping off to see the message.
- The signal passes the animation name as a `StringName`. Compare it with `&"bounce"` or `"bounce"`; both work.

Frequently used members:

Member	Meaning
`play("name")`	Start from the beginning
`play_backwards("name")`	Start at the end and run in reverse
`queue("name")`	Play after the current one finishes
`stop()`	Stop where it is
`is_playing()`	`true` while an animation is running
`current_animation`	The name of the running animation
`speed_scale`	Playback speed multiplier (`2.0` = twice as fast)
`autoplay`	Property that plays an animation when the scene starts

You can also wait for an animation inside a function with `await`:

```gdscript
# Attach to: Node2D "Bouncer"
extends Node2D

@onready var anim: AnimationPlayer = $AnimationPlayer

func _ready() -> void:
    anim.play("bounce")
    await anim.animation_finished
    print("The bounce is over")
```

For this you need the `bounce` animation to have its loop turned **off**.

> **Warning:** `play_backwards` always starts from the end of the animation, not from the current position. If an animation is half-way and you call `play_backwards`, it jumps to the end first. In the lab we avoid the problem by ignoring requests while an animation is running.
>
> **Try it:** Add a second track to `bounce` that changes `Sprite2D > Modulate` from white to red at time 0.5 and back to white at 1.0. Run the scene with F6.

AnimationTree: a very short introduction

Large characters have many animations (idle, walk, run, jump, attack) and rules about which can follow which. `AnimationTree` is a node that sits on top of an `AnimationPlayer` and blends animations using a **state machine** or **blend space**. You draw the states and arrows in the editor and ask the tree to travel to a state. A fragment of how code uses it:

```gdscript
# Fragment: needs an AnimationTree node with a state machine that has states "idle" and "run",
# and the tree's Active property turned on.
@onready var tree: AnimationTree = $AnimationTree

func _physics_process(_delta: float) -> void:
    var playback: AnimationNodeStateMachinePlayback = tree.get("parameters/playback")
    if velocity.length() > 0.0:
        playback.travel("run")
    else:
        playback.travel("idle")
```

You do not need `AnimationTree` for this course. Start with `AnimationPlayer` and `AnimatedSprite2D`, and look at the tree when you have a character with more than four or five animations.

9.2 AnimatedSprite2D and SpriteFrames

Classic 2D characters are drawn as a list of pictures (frames) shown one after another. The simplest node for that is `AnimatedSprite2D`. It uses a **SpriteFrames** resource: a set of named animations, each one a list of frames with a speed (frames per second) and a loop flag.

Setting it up

1. Add an `AnimatedSprite2D` node.
2. In the Inspector, next to **Sprite Frames**, choose **New SpriteFrames**, then click it to open the **SpriteFrames** panel at the bottom.
3. The default animation is called `default`. Rename it (double-click) to `idle`. Click **Add Animation** (the first icon) to create `run`.

4. To add frames from separate pictures, drag the image files from the FileSystem dock into the frame list. To add frames from one sheet of equal-sized cells, click the **Add frames from sprite sheet** icon, choose the image, set the number of horizontal and vertical frames, select the cells, and confirm.
5. Set **FPS** (frames per second) for each animation, typically 6 to 12, and the loop icon on or off.
6. Tick **Autoplay on load** (the icon next to the loop button) for the `idle` animation if you want it to start by itself.

Using it from code

```
# Attach to: CharacterBody2D "Hero"
# Scene tree:
# Hero (CharacterBody2D)
# ├── AnimatedSprite2D     <- SpriteFrames with animations "idle" and "run"
# └── CollisionShape2D
extends CharacterBody2D

@export var speed: float = 200.0

@onready var sprite: AnimatedSprite2D = $AnimatedSprite2D

func _physics_process(_delta: float) -> void:
    var direction: float = Input.get_axis("ui_left", "ui_right")
    velocity.x = direction * speed
    move_and_slide()
    if direction != 0.0:
        sprite.flip_h = direction < 0.0
        sprite.play("run")
    else:
        sprite.play("idle")
```

Calling `play("run")` every frame is fine: if `run` is already playing, it simply continues. If an animation name does not exist, Godot prints an error and nothing plays, so match the names exactly.

Member	Meaning
`play("name")`, `stop()`, `pause()`	Control playback
`animation`	The name of the current animation
`frame`	The current frame number
`speed_scale`	Speed multiplier
`flip_h`, `flip_v`	Mirror the picture
`animation_finished`	Emitted when a non-looping animation ends
`frame_changed`	Emitted every time the frame changes

A one-shot animation (loop off) can be awaited:

```
# Fragment: inside a function of a script that has a "sprite" variable (AnimatedSprite2D)
sprite.play("attack")
await sprite.animation_finished
sprite.play("idle")
```

> **Note:** When to use which? `AnimatedSprite2D` is perfect for picture-by-picture animation and has nothing else to learn. As soon as you want to animate other properties at the same time (a hitbox, a sound, a particle effect), switch to `AnimationPlayer`; it can animate a `Sprite2D`'s `frame` property *and* everything else on the same timeline.
>
> **Try it:** Create an `AnimatedSprite2D` with `idle` made of two or three pictures (use any images; even the same icon tinted differently is fine). Set FPS to 2 and run the scene. Change the FPS to 8 and compare.

9.3 Tween: animation by code

Sometimes you do not want to build a timeline. You only want "fade this out in half a second" or "move this label up by 20 pixels". A **Tween** does exactly that from a script.

You create a tween with `create_tween()` (a method of every node). Then you add steps to it:

```gdscript
# Attach to: Sprite2D
extends Sprite2D

func _ready() -> void:
    var tween: Tween = create_tween()
    tween.tween_property(self, "modulate:a", 0.0, 1.0)
```

`tween_property(object, "property", final_value, duration)` changes the property from its current value to the final value over the given seconds. The text `"modulate:a"` means "the `a` (alpha) part of `modulate`"; you can reach into vectors and colors with a colon.

Expected result: the sprite fades out to invisible in one second.

Chaining: steps happen one after another

By default, a tween runs its steps in **sequence**:

```gdscript
# Attach to: Sprite2D
extends Sprite2D

func _ready() -> void:
    var tween: Tween = create_tween()
    tween.tween_property(self, "position:x", position.x + 200.0, 0.5)
    tween.tween_interval(0.3)
    tween.tween_property(self, "position:x", position.x, 0.5)
    tween.tween_callback(queue_free)
```

Expected result: the sprite slides right by 200 pixels, waits 0.3 seconds, slides back, then removes itself from the game.

Step types:

Method	Does
`tween_property(obj, "prop", value, time)`	Changes a property
`tween_interval(time)`	Waits
`tween_callback(callable)`	Calls a function

Method	Does
`tween_method(callable, from, to, time)`	Calls a function with a value that goes from `from` to `to`

Parallel steps

Call `set_parallel(true)` and all later steps start together:

```
# Attach to: Sprite2D
extends Sprite2D

func pop_in() -> void:
    scale = Vector2.ZERO
    var tween: Tween = create_tween()
    tween.set_parallel(true)
    tween.tween_property(self, "scale", Vector2.ONE, 0.3)
    tween.tween_property(self, "modulate:a", 1.0, 0.3)
```

You can also mix: leave the tween sequential and call `.parallel()` on a single step to start it together with the previous one.

Easing: the feel of the motion

Constant speed looks robotic. Add `set_trans` (the shape of the curve) and `set_ease` (where the curve is applied):

```
# Fragment: inside a function of a Node2D script
var tween: Tween = create_tween()
tween.tween_property(self, "position:y", 300.0, 0.6)\
        .set_trans(Tween.TRANS_BOUNCE)\
        .set_ease(Tween.EASE_OUT)
```

Transition	Look
`TRANS_LINEAR`	Constant speed
`TRANS_QUAD`, `TRANS_CUBIC`	Smooth, natural
`TRANS_BACK`	Overshoots a little, then settles
`TRANS_ELASTIC`	Springy wobble
`TRANS_BOUNCE`	Bounces like a ball

Ease	Meaning
`EASE_IN`	Starts slowly
`EASE_OUT`	Ends slowly (the most common for movement that arrives)
`EASE_IN_OUT`	Both

Other useful settings: `.from(value)` (start from this value instead of the current one), `.as_relative()` (treat the final value as an amount to add), `.set_delay(sec)`, and `tween.set_loops(n)` (`0` loops forever).

finished, kill and reuse

A tween emits `finished` when its steps are all done:

```gdscript
# Attach to: Sprite2D
extends Sprite2D

func _ready() -> void:
    var tween: Tween = create_tween()
    tween.tween_property(self, "modulate:a", 0.0, 1.0)
    await tween.finished
    print("Faded out")
```

Three rules to remember:

1. **A tween plays once.** After it finishes it is dead. Create a new tween each time.
2. `create_tween()` **ties the tween to the node.** If the node is freed, the tween stops.
3. **Kill the old tween before starting a new one on the same property.** Two tweens fighting over `position` produce jitter. Store the tween in a variable and call `kill()` first:

```gdscript
# Attach to: Sprite2D
extends Sprite2D

var _tween: Tween

func move_to(target: Vector2) -> void:
    if _tween != null and _tween.is_valid():
        _tween.kill()
    _tween = create_tween()
    _tween.tween_property(self, "position", target,
0.4).set_trans(Tween.TRANS_CUBIC).set_ease(Tween.EASE_OUT)
```

> **Warning:** If you `await tween.finished` and the tween is killed, do not count on the awaiting code resuming: a killed tween does not finish normally. If you need a callback that is cancelled together with the tween, use `tween_callback()` as the last step instead.
>
> **Try it:** Make a `Button` that, when pressed, makes a `Label` grow from scale 0.5 to 1 with `TRANS_BACK` and `EASE_OUT`. Does it overshoot? Change to `TRANS_ELASTIC`.

Tween or AnimationPlayer?

Situation	Better tool
Values depend on runtime data (target position is computed)	Tween
One-off effects: fade, pop, flash, count-up	Tween
Authored animation with many properties, designers tweak it by eye	AnimationPlayer
Need to trigger sounds or functions at precise times	AnimationPlayer (Call Method track)
Same animation reused with different timing	Either

9.4 Timers, particles and screen shake

Timers: two ways

A **Timer node** is a node you add to the scene. It counts down `wait_time` seconds, then emits `timeout`. Properties: `wait_time`, `one_shot` (stop after one timeout), `autostart` (start when the scene starts). Methods: `start()`, `stop()`. It is best when the timer is part of the design: a spawn timer you can see and adjust in the

Inspector.

```gdscript
# Attach to: Node2D "Spawner"
# Scene tree:
# Spawner (Node2D)
# └── SpawnTimer (Timer)    <- Wait Time 1.5, Autostart: on, One Shot: off
extends Node2D

@onready var spawn_timer: Timer = $SpawnTimer

func _ready() -> void:
    spawn_timer.timeout.connect(_on_spawn_timer_timeout)

func _on_spawn_timer_timeout() -> void:
    print("Spawn something now")
```

Expected result: a message every 1.5 seconds.

A quick one-off delay does not need a node. `get_tree().create_timer(seconds)` makes a temporary timer that frees itself:

```gdscript
# Fragment: inside any function of a script
print("Get ready...")
await get_tree().create_timer(2.0).timeout
print("Go!")
```

	`Timer` **node**	`create_timer()`
Visible in Inspector	yes	no
Repeats	yes (`one_shot` off)	no, once
Can be stopped or restarted	yes	not easily
Best for	spawners, cooldowns, repeated events	a single "wait then continue"

> **Note:** A `Timer` node obeys the pause rules from Chapter 8: it stops when the game is paused, unless its `process_mode` is Always. A timer made by `create_timer(time_sec)` has a second argument, `process_always`, which defaults to `true`, so by default it keeps counting while the game is paused. Pass `false` if the delay should freeze with the game.

Particles: a first effect

Particles are many small sprites created and destroyed quickly. `CPUParticles2D` is simple, runs on any computer and is plenty for small effects. (`GPUParticles2D` is the faster, more powerful version that needs a process material; start with the CPU one.)

Add a `CPUParticles2D` node and play with these properties in the Inspector:

Property	**Meaning**
`emitting`	Turn the effect on or off
`amount`	Number of particles
`lifetime`	Seconds each particle lives
`one_shot`	Emit one burst, then stop

Property	Meaning
`explosiveness`	0 = steady stream, 1 = all at once
`direction`, `spread`	Direction and angle of the fan
`initial_velocity_min/max`	Starting speed range
`gravity`	A constant pull, for example `(0, 300)`
`scale_amount_min/max`	Particle size
`color`	Tint

Without a texture, each particle is a small square, which is enough for dust and sparks. In code you can build a burst when something happens:

```gdscript
# Attach to: any Node2D. Call burst(Vector2(100, 100)) when something is hit.
extends Node2D

func burst(at: Vector2) -> void:
    var sparks: CPUParticles2D = CPUParticles2D.new()
    sparks.position = at
    sparks.one_shot = true
    sparks.amount = 16
    sparks.lifetime = 0.6
    sparks.explosiveness = 1.0
    sparks.direction = Vector2.UP
    sparks.spread = 180.0
    sparks.initial_velocity_min = 80.0
    sparks.initial_velocity_max = 160.0
    sparks.gravity = Vector2(0.0, 300.0)
    sparks.scale_amount_min = 3.0
    sparks.scale_amount_max = 5.0
    sparks.color = Color.YELLOW
    add_child(sparks)
    sparks.emitting = true
    await get_tree().create_timer(sparks.lifetime + 0.2).timeout
    sparks.queue_free()
```

Expected result: a short yellow burst that cleans itself up. Always remove one-shot particle nodes when done, otherwise they pile up.

Screen shake

A screen shake makes impacts feel strong. The simplest version moves the `Camera2D`'s `offset` property by a random amount that fades to zero. A `Tween` with `tween_method` is perfect for the fade:

```gdscript
# Attach to: Camera2D
class_name ShakeCamera
extends Camera2D

var _shake_tween: Tween

func shake(strength: float = 10.0, duration: float = 0.3) -> void:
    if _shake_tween != null and _shake_tween.is_valid():
        _shake_tween.kill()
    _shake_tween = create_tween()
    _shake_tween.tween_method(_apply_shake.bind(strength), 1.0, 0.0, duration)
    _shake_tween.tween_callback(func() -> void: offset = Vector2.ZERO)

func _apply_shake(amount: float, strength: float) -> void:
    offset = Vector2(randf_range(-1.0, 1.0), randf_range(-1.0, 1.0)) * strength * amount
```

How it works: `tween_method` calls `_apply_shake` many times, passing a value that goes from 1.0 down to 0.0. `bind(strength)` adds the second argument. The camera jumps to a random point within a shrinking radius. At the end, the offset is set to zero.

Call it from anywhere with a reference to the camera: `camera.shake(8.0, 0.25)`. Use small values: 4 to 12 pixels for small hits, 20 for explosions. Heavy shake makes some players feel sick, so offer a setting to reduce it later.

9.5 Audio

Players and streams

Sound in Godot has three parts:

1. A **stream**: the audio file loaded as a resource (`AudioStream`).
2. A **player node** that plays a stream.
3. An **audio bus** that carries the sound to the speakers.

Node	Use
`AudioStreamPlayer`	Sound with no position: music, menu clicks, UI
`AudioStreamPlayer2D`	Sound that comes from a place in the 2D world: louder when the camera is near, panned left and right

Both have `stream`, `volume_db`, `pitch_scale`, `bus`, `autoplay`, `playing`, and methods `play()` and `stop()`. They emit `finished` when the sound ends.

File formats

Format	Good for	Notes
`.wav`	Short sound effects	Uncompressed, instant to play, big file
`.ogg` (Vorbis)	Music, long sounds	Compressed, small, can loop
`.mp3`	Music	Works, but `.ogg` is usually the better choice in Godot

Where to get sounds: make your own beeps and effects with free tools like **sfxr** or its browser version **jsfxr** (you click "Pickup coin" or "Blip" and export a `.wav`), or download CC0 (public domain) packs from **Kenney.nl** or **OpenGameArt.org**. Always read the license, and keep a note of where each file came from for your credits (Chapter 11).

Put sounds in a folder such as `res://sounds/`. When you copy a file into the project, Godot imports it automatically. Select the file and open the **Import** dock to see options.

Looping

For music you want a loop. Select the `.ogg` or `.mp3` file, open the **Import** dock, tick **Loop** (and optionally set a **Loop Offset**), then press **Reimport**. For `.wav` files, set **Edit > Loop Mode** to *Forward* in the Import dock. This is a property of the imported stream, not of the player node.

Playing a sound safely

In a lab you might not have a sound file in the player node yet. A call to `play()` with an empty `stream` does nothing useful and may print an error. Get in the habit of guarding:

```
# Attach to: AudioStreamPlayer named "ClickSound"
extends AudioStreamPlayer

func play_click() -> void:
    if stream:
        play()
```

`if stream:` is `true` only when a stream is assigned. Your game will run silently instead of printing errors if a file is missing.

To load a sound from code, use `load` (see Chapter 10 for `load` vs `preload`):

```
# Attach to: AudioStreamPlayer
extends AudioStreamPlayer

func play_file(path: String) -> void:
    if ResourceLoader.exists(path):
        stream = load(path)
        play()
```

Sound effects with variation

The same sound repeated a hundred times gets annoying. Change `pitch_scale` slightly each time:

```
# Attach to: AudioStreamPlayer2D named "StepSound"
extends AudioStreamPlayer2D

func play_step() -> void:
    if stream:
        pitch_scale = randf_range(0.9, 1.1)
        play()
```

Audio buses

Open the **Audio** panel at the bottom of the editor. There is one bus, **Master**. Everything is mixed into it. Add more buses with **Add Bus** and name them **Music** and **SFX**. Both send to Master by default. Why bother?

- The player gets separate sliders for music and sound effects.
- You can quiet the music while a dialogue plays.
- You can add effects (reverb, low-pass filter) to one bus.

Godot keeps the layout in `res://default_bus_layout.tres` (the Audio panel also has **Load**, **Save As** and **Default** buttons if you want to manage the file yourself). Assign a player to a bus with its `bus` property (a drop-down in the Inspector).

Volume in code

Volume is measured in **decibels (dB)**. 0 dB is the original volume. -6 dB is about half as loud. -80 dB is silence. Sliders and humans think in a 0 to 1 scale, so convert:

```gdscript
# Attach to: any Node. Call set_bus_volume("Music", 0.5) for half volume.
extends Node

func set_bus_volume(bus_name: String, linear_value: float) -> void:
    var index: int = AudioServer.get_bus_index(bus_name)
    if index == -1:
        push_warning("No audio bus named " + bus_name)
        return
    AudioServer.set_bus_volume_db(index, linear_to_db(linear_value))
```

Function	Use
`AudioServer.get_bus_index("Music")`	Find a bus number by name; returns `-1` if missing
`AudioServer.set_bus_volume_db(index, db)`	Set volume
`AudioServer.get_bus_volume_db(index)`	Read volume
`AudioServer.set_bus_mute(index, true)`	Mute a bus
`linear_to_db(x)`	0..1 to decibels (`0` gives silence)
`db_to_linear(db)`	Decibels back to 0..1

Fading

A hard stop of music sounds rough. Use a tween on `volume_db`:

```gdscript
# Attach to: AudioStreamPlayer named "Music"
extends AudioStreamPlayer

func fade_out(seconds: float = 1.0) -> void:
    var tween: Tween = create_tween()
    tween.tween_property(self, "volume_db", -40.0, seconds)
    tween.tween_callback(stop)

func fade_in(seconds: float = 1.0) -> void:
    if not stream:
        return
    volume_db = -40.0
    play()
    var tween: Tween = create_tween()
    tween.tween_property(self, "volume_db", 0.0, seconds)
```

Expected result: music glides to quiet over `seconds`, then stops; or starts quietly and rises. After `fade_out`, remember to reset `volume_db` to 0 before playing again (`fade_in` does that for you).

> **Try it:** Make a scene with an `AudioStreamPlayer`, a `Button` labelled "Beep" and a `Button` labelled "Fade". Make the first call `play_click()` and the second call `fade_out(2.0)` on a looping sound.

9.6 Putting it together: picking the right tool

I want to…	Use
Show a walk cycle made of pictures	`AnimatedSprite2D`
Open a door, with a sound at the right moment	`AnimationPlayer` with a Call Method track
Fade out a label	`Tween`

I want to...	Use
Wait two seconds then continue	`await get_tree().create_timer(2.0).timeout`
Spawn an enemy every 3 seconds	`Timer` node
A burst of sparks on impact	`CPUParticles2D`
Short sound on a button	`AudioStreamPlayer` with `.wav`
Background music with a volume slider	`AudioStreamPlayer` on the `Music` bus + `AudioServer`

Good animation rules:

1. **Keep it short.** Interface animations should take 0.1 to 0.3 seconds. Players wait for them every time.
2. **Ease everything.** `TRANS_QUAD` or `TRANS_CUBIC` with `EASE_OUT` make most motion feel right.
3. **Give feedback for every action.** A button should react to hover and press. A pickup should make a sound and a small movement.

Lab 9: An animated, sound-enabled door and button

Time: 60 to 90 minutes.

Goal: build a room with a floor button and **two** doors that open when you step on the button. One door is animated with a **script-driven Tween**, the other with an **AnimationPlayer**. Both play a sound, the room shakes and spits dust when a door finishes opening.

Sound files: the code runs without sounds (every `play()` is guarded by `if stream`). To hear them, create two short `.wav` files with jsfxr or sfxr (one "blip" for the button, one "hit" or "powerup" for the doors) or download a free CC0 pack from Kenney.nl or OpenGameArt.org, and put them in `res://sounds/`. Optionally find one looping music `.ogg`.

Files:

```
res://
├── player.gd
├── floor_button.gd
├── shake_camera.gd
├── door_tween.tscn / door_tween.gd
├── door_anim.tscn  / door_anim.gd
├── door_room.tscn  / door_room.gd
├── default_bus_layout.tres      (created by the Audio panel)
└── sounds/
```

Step 1: buses

1. Open the **Audio** panel (bottom). Click **Add Bus** twice. Name the new buses `Music` and `SFX`.
2. Save the layout if the panel asks you to (it goes to `default_bus_layout.tres`; the **Save As** button in the panel does the same).

Step 2: the player

Scene `door_room.tscn` will include the player. Create it first as part of the room (Step 6). The script `player.gd`:

```gdscript
extends CharacterBody2D

@export var speed: float = 260.0

func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")
    velocity = direction * speed
    move_and_slide()
```

Step 3: the camera

`shake_camera.gd` is exactly the `ShakeCamera` script from Section 9.4. Create the file and paste it in:

```gdscript
class_name ShakeCamera
extends Camera2D

var _shake_tween: Tween

func shake(strength: float = 10.0, duration: float = 0.3) -> void:
    if _shake_tween != null and _shake_tween.is_valid():
        _shake_tween.kill()
    _shake_tween = create_tween()
    _shake_tween.tween_method(_apply_shake.bind(strength), 1.0, 0.0, duration)
    _shake_tween.tween_callback(func() -> void: offset = Vector2.ZERO)

func _apply_shake(amount: float, strength: float) -> void:
    offset = Vector2(randf_range(-1.0, 1.0), randf_range(-1.0, 1.0)) * strength * amount
```

Step 4: the floor button

Scene tree for the button (you can build it inside `door_room.tscn`, or as `floor_button.tscn`; the lab uses the node name `FloorButton`):

```
FloorButton (Area2D)            <- script floor_button.gd
├── Pad (Sprite2D)              <- Texture res://icon.svg, Scale (0.5, 0.15), Modulate #c0392b
├── CollisionShape2D            <- RectangleShape2D, Size (64, 24)
└── PressSound (AudioStreamPlayer2D)   <- Bus: SFX
```

`floor_button.gd`:

```gdscript
class_name FloorButton
extends Area2D

signal stepped_on
signal stepped_off

const COLOR_UP: Color = Color("#c0392b")
const COLOR_DOWN: Color = Color("#4caf50")

var _tween: Tween

@onready var pad: Sprite2D = $Pad
@onready var sound: AudioStreamPlayer2D = $PressSound

func _ready() -> void:
    body_entered.connect(_on_body_entered)
    body_exited.connect(_on_body_exited)

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        _set_pressed(true)
```

```
        if sound.stream:
            sound.play()
        stepped_on.emit()

func _on_body_exited(body: Node2D) -> void:
    if body.is_in_group("player"):
        _set_pressed(false)
        stepped_off.emit()

func _set_pressed(is_pressed: bool) -> void:
    if _tween != null and _tween.is_valid():
        _tween.kill()
    _tween = create_tween()
    _tween.set_parallel(true)
    var target_scale_y: float = 0.08 if is_pressed else 0.15
    var target_color: Color = COLOR_DOWN if is_pressed else COLOR_UP
    _tween.tween_property(pad, "scale:y", target_scale_y, 0.1)
    _tween.tween_property(pad, "modulate", target_color, 0.1)
```

Step 5: the two doors

Both doors share this tree shape. Build `door_tween.tscn` first:

```
DoorTween (Node2D)              <- script door_tween.gd
├── Frame (Sprite2D)            <- res://icon.svg, Scale (0.6, 1.7), Modulate #222222
├── Slab (Sprite2D)             <- res://icon.svg, Scale (0.5, 1.5), Modulate #8b5a2b
└── OpenSound (AudioStreamPlayer2D)    <- Bus: SFX
```

`door_tween.gd`:

```
class_name TweenDoor
extends Node2D

signal opened
signal closed

@export var open_offset: Vector2 = Vector2(0.0, -190.0)
@export var duration: float = 0.6

var is_open: bool = false

var _tween: Tween

@onready var slab: Sprite2D = $Slab
@onready var sound: AudioStreamPlayer2D = $OpenSound
@onready var _closed_position: Vector2 = slab.position

func open() -> void:
    if is_open:
        return
    is_open = true
    _play_sound()
    var tween: Tween = _slide_to(_closed_position + open_offset)
    tween.tween_callback(func() -> void: opened.emit())

func close() -> void:
    if not is_open:
        return
    is_open = false
    _play_sound()
    var tween: Tween = _slide_to(_closed_position)
    tween.tween_callback(func() -> void: closed.emit())

func _slide_to(target: Vector2) -> Tween:
    if _tween != null and _tween.is_valid():
```

```
        _tween.kill()
    _tween = create_tween()
    _tween.tween_property(slab, "position", target,
duration).set_trans(Tween.TRANS_CUBIC).set_ease(Tween.EASE_OUT)
    return _tween

func _play_sound() -> void:
    if sound.stream:
        sound.play()
```

This door can reverse in the middle of a move: the old tween is killed and a new one starts from where the slab is. The "opened" signal only fires if the move completes.

Now `door_anim.tscn`:

```
DoorAnim (Node2D)               <- script door_anim.gd
├── Frame (Sprite2D)            <- same settings as before
├── Slab (Sprite2D)             <- same settings as before
├── OpenSound (AudioStreamPlayer2D)    <- Bus: SFX
└── AnimationPlayer
```

Create the animation in the editor:

1. Select `AnimationPlayer` > **Animation > New** > name `open`. Length `0.6`. Loop **off**.
2. Select `Slab`. At time `0`, click the key icon beside **Position** (the value is `(0, 0)`).
3. Move the time cursor to `0.6`. Set Position to `(0, -190)` and click the key icon again.
4. Click the interpolation icon on the track and choose **Cubic**.
5. In the animation panel click **Add Track > Call Method Track**, choose the root node `DoorAnim`. Right-click the new track at time `0.05` > **Insert Key**, and choose the method `play_sound` (it appears once the script from the next step is attached; if you do not see it, attach the script first).
6. Open the **Edit** menu in the animation panel and choose **Create RESET Track(s) From Animation**. Check that `Slab` is back at `(0, 0)` in the viewport.

`door_anim.gd`:

```
class_name AnimDoor
extends Node2D

signal opened
signal closed

var is_open: bool = false

@onready var anim: AnimationPlayer = $AnimationPlayer
@onready var sound: AudioStreamPlayer2D = $OpenSound

func _ready() -> void:
    anim.animation_finished.connect(_on_animation_finished)

func open() -> void:
    if is_open or anim.is_playing():
        return
    is_open = true
    anim.play("open")

func close() -> void:
    if not is_open or anim.is_playing():
        return
    is_open = false
    play_sound()
    anim.play_backwards("open")
```

```
# Called by the Call Method track at the start of the "open" animation
# (close() also calls it directly). The "not sound.playing" check stops two
# near-simultaneous calls from restarting the sound.
func play_sound() -> void:
    if sound.stream and not sound.playing:
        sound.play()

func _on_animation_finished(anim_name: StringName) -> void:
    if anim_name != &"open":
        return
    if is_open:
        opened.emit()
    else:
        closed.emit()
```

Notice how the two doors have the **same interface** (`open()`, `close()`, `opened`, `closed`). The room does not care how a door is animated. That is a good habit: hide how, expose what.

The `AnimDoor` ignores requests while an animation runs, which avoids the `play_backwards` jump explained in Section 9.1.

Step 6: the room

Scene tree for `door_room.tscn`:

```
DoorRoom (Node2D)                   <- script door_room.gd
├── Camera2D                        <- script shake_camera.gd, Position (576, 324)
├── Player (CharacterBody2D)        <- script player.gd, Position (200, 450), Group: player
│   ├── Sprite2D                    <- res://icon.svg, Scale (0.25, 0.25)
│   └── CollisionShape2D            <- RectangleShape2D, Size (32, 32)
├── FloorButton                     <- the button from Step 4, Position (576, 450)
├── TweenDoor                       <- instance of door_tween.tscn, Position (350, 250)
├── AnimDoor                        <- instance of door_anim.tscn, Position (800, 250)
└── Music (AudioStreamPlayer)       <- Bus: Music (optional stream: a looping .ogg)
```

Name the instanced nodes exactly `TweenDoor` and `AnimDoor` (rename them after instancing). Add `Player` to the group `player`.

`door_room.gd`:

```
extends Node2D

@onready var camera: ShakeCamera = $Camera2D
@onready var floor_button: FloorButton = $FloorButton
@onready var tween_door: TweenDoor = $TweenDoor
@onready var anim_door: AnimDoor = $AnimDoor
@onready var music: AudioStreamPlayer = $Music

func _ready() -> void:
    floor_button.stepped_on.connect(_on_button_stepped_on)
    floor_button.stepped_off.connect(_on_button_stepped_off)
    tween_door.opened.connect(_on_door_opened.bind(tween_door))
    anim_door.opened.connect(_on_door_opened.bind(anim_door))
    if music.stream:
        music.play()

func _on_button_stepped_on() -> void:
    tween_door.open()
    anim_door.open()

func _on_button_stepped_off() -> void:
    tween_door.close()
```

```
    anim_door.close()

func _on_door_opened(door: Node2D) -> void:
    camera.shake(6.0, 0.25)
    _spawn_dust(door.global_position + Vector2(0.0, 90.0))

func _spawn_dust(at: Vector2) -> void:
    var dust: CPUParticles2D = CPUParticles2D.new()
    dust.position = at
    dust.one_shot = true
    dust.amount = 20
    dust.lifetime = 0.7
    dust.explosiveness = 0.9
    dust.direction = Vector2.UP
    dust.spread = 60.0
    dust.initial_velocity_min = 40.0
    dust.initial_velocity_max = 120.0
    dust.gravity = Vector2(0.0, 120.0)
    dust.scale_amount_min = 3.0
    dust.scale_amount_max = 6.0
    dust.color = Color(0.8, 0.75, 0.6, 0.8)
    add_child(dust)
    dust.emitting = true
    await get_tree().create_timer(dust.lifetime + 0.3).timeout
    dust.queue_free()
```

`bind(tween_door)` adds the door as the argument for `_on_door_opened`, because the `opened` signal itself has no arguments.

Step 7: run and test

Press **F6** with `door_room.tscn` open.

1. Move the player with the arrow keys onto the red pad.
2. The pad squashes and turns green within 0.1 s. Both doors slide up. When each finishes opening, the screen shakes a little and dust puffs.
3. Walk off the pad. The doors slide shut. (The `AnimDoor` ignores a request if it is still moving; the `TweenDoor` reverses at once. Try stepping on and off quickly to see the difference.)
4. With sounds assigned, you hear the click and the door sounds. Without sounds, no errors are printed.
5. Open **Audio** panel and mute the `SFX` bus. Sounds disappear. Unmute.

Expected result: two doors with the same behavior but different implementations, a reactive button, a short screen shake and dust, and sound that is optional.

Extension: In `AnimDoor`, change the Call Method track so it calls a second method `shake_request` that emits a new signal; connect it in `door_room.gd`.

Chapter summary

- `AnimationPlayer` animates any property of any node using tracks and keyframes; use Call Method tracks for events and a RESET track to keep the rest pose safe.
- `AnimatedSprite2D` plays frame lists from `SpriteFrames`; `AnimationTree` blends many animations and is a later step.
- A `Tween` animates by code: `create_tween()`, `tween_property`, chaining, `set_parallel`, easing with `set_trans` and `set_ease`, and a `finished` signal. A tween runs once.

- Use a `Timer` node for repeating or inspectable timers, and `await get_tree().create_timer(t).timeout` for a quick delay.
- `CPUParticles2D` gives easy bursts; a screen shake is a short, fading random camera offset.
- Audio: `AudioStreamPlayer` for non-positional sound, `AudioStreamPlayer2D` for positional; `.wav` for effects, `.ogg` for music.
- Route sounds into buses (Master, Music, SFX); change volume with `AudioServer.set_bus_volume_db` and `linear_to_db`; fade by tweening `volume_db`.
- Guard `play()` with `if stream:` so missing sounds never crash the game.

Review questions

1. Name three kinds of things an `AnimationPlayer` track can control.
2. What problem does the RESET animation solve?
3. True or false: `animation_finished` is emitted every time a looping animation reaches its end.
4. Which node would you choose for a character walk cycle made from a sprite sheet, `AnimatedSprite2D` or `AnimationPlayer`? Why?
5. After a Tween finishes, can you call `tween_property` on it again to play it once more? What should you do instead?
6. What is the difference between `set_trans` and `set_ease`?
7. When is a `Timer` node a better choice than `await get_tree().create_timer(1.0).timeout`?
8. Which audio file format is the best choice for a long, looping piece of music, and which for a very short sound effect?
9. What does `AudioServer.set_bus_volume_db(index, linear_to_db(0.5))` do, and why do we convert with `linear_to_db`?
10. Why should you write `if stream:` before calling `play()` in the lab scripts?

Exercises

1. **Practice (easy):** Create an `AnimationPlayer` animation named `flash` that turns a sprite red and back to white in 0.3 seconds, and play it when you press the Space key (`ui_accept`).
2. **Practice (easy):** Write a `Tween` that makes a `Label` slide in from 100 pixels to the left of its position while fading in, in parallel, over 0.4 seconds.
3. **Apply (medium):** Add `MusicSlider` and `SfxSlider` (two `HSlider` nodes) to a small scene and wire them to the `Music` and `SFX` buses with the `set_bus_volume` function from Section 9.5.
4. **Apply (medium):** Make the `FloorButton` play a sound with `pitch_scale` random between 0.95 and 1.05, and make its pad squash with `TRANS_BACK` easing.
5. **Challenge (hard):** Create a `Music` autoload with `play_track(stream, fade_seconds)` that fades the old track out and the new track in at the same time, using two `AudioStreamPlayer` children and tweens.

Common mistakes

- **The sprite is in the wrong place when the scene starts.** -> The scene was saved while the animation was scrubbed to a later frame. -> Create a RESET track and re-save the scene.
- `animation_finished` **never fires.** -> The animation has looping turned on. -> Turn looping off. (`AnimatedSprite2D` has an `animation_looped` signal for loops; for an `AnimationPlayer` loop, add a Call Method track key at the end instead.)

- **A tween does nothing the second time.** -> The same finished tween was reused. -> Call `create_tween()` again every time.
- **Object jitters while tweening.** -> Two tweens (or a tween and `_physics_process`) change the same property. -> Kill the old tween first and let only one thing control the property.
- **No sound plays.** -> No stream assigned, wrong bus muted, `volume_db` too low, or the file has not imported. -> Check `stream`, the bus in the Audio panel, and the Import dock.
- **Music clicks or stops abruptly.** -> It was stopped at full volume. -> Fade `volume_db` down with a tween before `stop()`.
- **A Call Method track key does nothing.** -> The function name was misspelled, or the track points at a node that does not have it. -> Pick the function from the list the editor offers after you attach the script.

Further reading

- Introduction to animation features: https://docs.godotengine.org/en/stable/tutorials/animation/introduction.html
- Using the AnimationPlayer class reference: https://docs.godotengine.org/en/stable/classes/class_animationplayer.html
- Tween class reference: https://docs.godotengine.org/en/stable/classes/class_tween.html
- AnimatedSprite2D class reference: https://docs.godotengine.org/en/stable/classes/class_animatedsprite2d.html
- Timer class reference: https://docs.godotengine.org/en/stable/classes/class_timer.html
- CPUParticles2D class reference: https://docs.godotengine.org/en/stable/classes/class_cpuparticles2d.html
- Audio buses: https://docs.godotengine.org/en/stable/tutorials/audio/audio_buses.html
- AudioStreamPlayer class reference: https://docs.godotengine.org/en/stable/classes/class_audiostreamplayer.html
- Importing audio samples: https://docs.godotengine.org/en/stable/tutorials/assets_pipeline/importing_audio_samples.html

Chapter 10: Resources and Saving

Learning objectives

By the end of this chapter you will be able to:

- Explain the difference between a `Node` and a `Resource`, and say when each is the right tool.
- Create a custom `Resource` class with `class_name` and `@export`, save instances as `.tres` files and use them from scenes and scripts.
- Load assets with `load` and `preload`, and create copies of scenes with `PackedScene.instantiate()`.
- Read and write text files in `user://` with `FileAccess`, and encode data as JSON with `JSON.stringify` and `JSON.parse_string`.
- Store player settings with `ConfigFile`.
- Design a save system: choose what to save, add a version number, collect state with groups, and keep game data in a `GameState` autoload.
- Find your way around the most important Project Settings.

Key terms

- **Resource**: a data object (not part of the scene tree) that can be saved to a file, shared and reused: textures, sounds, themes, your own data classes.
- `.tres` / `.res`: file formats for saved resources; `.tres` is readable text, `.res` is binary.
- `preload`: loads a resource when the script is compiled (before the game runs); the path must be a constant string.
- `load`: loads a resource at run time; the path may be computed.
- **PackedScene**: a resource that stores a whole scene, used to create copies with `instantiate()`.
- `res://`: the project folder; read-only in an exported game.
- `user://`: a folder for data created while playing (saves, settings); always writable.
- **JSON**: a text format for dictionaries, arrays, numbers, strings, booleans and `null`.
- **ConfigFile**: Godot's class for INI-style settings files with sections and keys.
- **Serialization**: turning game data into text or bytes so it can be stored.
- **Save version**: a number written into every save file so that newer versions of the game can read older saves.
- **Autoload**: a script or scene that Godot loads automatically at start and keeps alive in every scene (Chapter 5).

10.1 Nodes and Resources

You know Nodes well by now: they live in the scene tree, run `_process`, receive input and draw things. Godot has a second family of objects: **Resources**.

	Node	Resource
In the scene tree?	Yes	No
Has `_process`, `_ready`?	Yes	No

	Node	Resource
Saved in	`.tscn` scenes	`.tres` or `.res` files, or inside a scene
Shared between users?	One place in the tree only	Many nodes can point at the same one
Typical examples	`Sprite2D`, `Timer`, `Button`	`Texture2D`, `AudioStream`, `Theme`, `SpriteFrames`, `PackedScene`

You already use resources every day. When you drag `icon.svg` into the `Texture` slot of a `Sprite2D`, the sprite holds a reference to a `Texture2D` resource. The `SpriteFrames` of Chapter 9 and the `Theme` of Chapter 8 are resources too.

> **Note:** A resource is "data with a type". A node is "behavior in the tree". If you only need to store information, such as the stats of an enemy, a Resource is lighter and cleaner than a Node.

Resources are shared

When two nodes use the same texture file, there is only **one** `Texture2D` object in memory and both nodes point to it. Loading the same path again returns the same object (it is cached).

That is usually what you want. It also leads to the most common Resource surprise: **if you change a shared resource, it changes for everyone who uses it**.

```gdscript
# Fragment: two nodes share a resource
var a: Resource = load("res://coin_gold.tres")
var b: Resource = load("res://coin_gold.tres")
print(a == b)   # true: it is the same object
```

To get a private copy, call `duplicate()`:

```gdscript
# Fragment
var my_copy: Resource = load("res://coin_gold.tres").duplicate()
```

In the Inspector, a resource property that you do not want to share can be made unique with the dropdown arrow > **Make Unique**. Resources inside a scene can also be set to **Local to Scene** so that each instance of the scene gets its own copy.

> **Why it works:** A `Resource` is a reference type, like an object in Java. Variables hold a reference, not a copy, so changing it through one variable changes it for all.

10.2 Custom Resources

You can define your own resource type with a script. This is one of the best features of Godot for organizing game data: you design a type, then create many instances in the editor and edit them with the Inspector, no code needed.

Step 1: write the class

Create `coin_type.gd` (a plain script, no node needed):

```gdscript
class_name CoinType
extends Resource

@export var display_name: String = "Copper"
@export var value: int = 1
@export var color: Color = Color.WHITE
```

Two things make this work:

1. `extends Resource`: the script is a data class.
2. `class_name CoinType`: registers the name so you can use `CoinType` as a type everywhere, and so that it shows up in the **Create New Resource** dialog.

`@export` makes each variable editable in the Inspector and stored in the `.tres` file.

> **Warning:** Never choose a `class_name` that is already a Godot class (`Resource`, `Node`, `Timer`, `Tween`, `Input`). Also never use the same name for a `class_name` and an autoload; Godot will refuse.

Step 2: create instances in the editor

1. In the FileSystem dock, right-click > **New Resource...**
2. Type `CoinType` in the search box, select it, click **Create**.
3. Save it as `coin_copper.tres`. In the Inspector set `display_name` = `Copper`, `value` = `1`, `color` = brown.
4. Repeat for `coin_gold.tres` with `Gold`, `5`, yellow.

Open `coin_gold.tres` in a text editor and you will see something like this:

```
[gd_resource type="Resource" script_class="CoinType" load_steps=2 format=3]

[ext_resource type="Script" path="res://coin_type.gd" id="1"]

[resource]
script = ExtResource("1")
display_name = "Gold"
value = 5
color = Color(1, 0.84, 0, 1)
```

The first line (the header) differs a little between Godot versions; newer ones add a `uid=` value and drop `load_steps`. The rest is plain readable text, which makes it friendly for version control (Chapter 11).

Step 3: use them

Any script can export a variable of your type. The Inspector will then show a slot where you drag a `.tres` file:

```gdscript
# Attach to: Area2D "Coin" (scene tree: Coin > Sprite2D, CollisionShape2D)
class_name Coin
extends Area2D

signal collected(value: int)

@export var coin_type: CoinType

@onready var sprite: Sprite2D = $Sprite2D

func _ready() -> void:
    if coin_type != null:
```

```
        sprite.self_modulate = coin_type.color
    body_entered.connect(_on_body_entered)

func get_value() -> int:
    if coin_type == null:
        return 1
    return coin_type.value

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        collected.emit(get_value())
        queue_free()
```

Note the null check. A good habit when using exported resources: the designer (you, later) might forget to assign one. Give a sensible fallback.

You can also export **lists** of resources. `Array[CoinType]` shows a typed list in the Inspector:

```
# Fragment: a spawner that can spawn several kinds of coin
@export var coin_types: Array[CoinType] = []

func pick_type() -> CoinType:
    if coin_types.is_empty():
        return null
    return coin_types.pick_random()
```

> **Try it:** Create `coin_type.gd`, make the two `.tres` files, and open the Inspector for each. Change `value` of the gold coin to 10 and notice that nothing else in the project needs editing.

Resources can hold methods too

A resource is a normal script, so it may have functions:

```
# coin_type.gd (extended)
class_name CoinType
extends Resource

@export var display_name: String = "Copper"
@export var value: int = 1
@export var color: Color = Color.WHITE

func describe() -> String:
    return "%s coin worth %d" % [display_name, value]
```

Saving resources from code

`ResourceSaver.save(resource, path)` writes a resource to a file and returns an `Error` code (`OK` when it worked). It is great in tools and editor plugins:

```
# Fragment: run once from a script to create a resource file by code
var silver: CoinType = CoinType.new()
silver.display_name = "Silver"
silver.value = 2
silver.color = Color.SILVER
var err: Error = ResourceSaver.save(silver, "res://coin_silver.tres")
if err != OK:
    push_error("Could not save: " + error_string(err))
```

This works in the editor, because `res://` is writable there. In an exported game it is not (Section 10.4).

> **Warning:** Do **not** use `.tres` files for player save games. Loading a resource file can run scripts stored in it, so a modified save file from the internet could run code on your player's computer. Use JSON for saves (Section 10.5). `.tres` files are for data that *you* ship with the game.

10.3 preload, load and PackedScene

preload vs load

	`preload("res://x.tscn")`	`load("res://x.tscn")`
When	While the script compiles, before the game starts	When the line runs
Path	Must be a constant string	Any string, computed at run time
Failing path	Error in the editor right away	Error at run time, returns `null`
Speed in game	No waiting	Can cause a small pause for big files

Rule of thumb: use `preload` when you know the path while writing the script, and `load` when the path is chosen while playing (for example "level_" + str(number)).

```gdscript
# Fragment: preload at the top of a script, as a constant
const COIN_SCENE: PackedScene = preload("res://coin.tscn")

# Fragment: load with a computed path
func load_level(number: int) -> PackedScene:
    var path: String = "res://levels/level_%d.tscn" % number
    if not ResourceLoader.exists(path):
        push_warning("Missing level: " + path)
        return null
    return load(path)
```

`ResourceLoader.exists(path)` tells you whether a resource exists before you try to load it, which avoids an error message.

PackedScene and instantiate

When you save a scene, you create a `PackedScene`: the blueprint of a node tree. `instantiate()` builds a fresh copy of that tree in memory. Each call makes a new, independent copy; nothing appears in the game until you `add_child` it.

```gdscript
# Attach to: Node2D "Spawner" with a child Node2D named "Coins"
extends Node2D

const COIN_SCENE: PackedScene = preload("res://coin.tscn")

@onready var coins: Node2D = $Coins

func spawn_coin_at(where: Vector2) -> void:
    var coin: Coin = COIN_SCENE.instantiate()
    coin.position = where
    coins.add_child(coin)
```

Order matters: set properties **before** `add_child` when you can, because `_ready` runs when the node enters the tree and may need those values. (The lab's `game.gd` does this: it assigns `coin_type` before calling `add_child`.)

> **Warning:** Do not call `add_child` for a physics body or area from inside a physics callback such as `body_entered`. Godot reports "Can't change this state while flushing queries". Use `call_deferred`: `spawn_coin.call_deferred()`. Chapter 8's lab does this.
>
> **Try it:** In an empty scene, add a `Node2D` with the script above. Add a line to `_ready` that calls `spawn_coin_at(Vector2(100, 100))` and `spawn_coin_at(Vector2(200, 100))`. Run: two independent coins appear.

10.4 Files: res://, user:// and FileAccess

Two folders, two rules

Path	Meaning	Writable while developing?	Writable in an exported game?
`res://`	Your project folder	Yes (editor only)	**No**, it is packed in a read-only file
`user://`	A per-game folder in the user's profile	Yes	**Yes**

Always write saves and settings to `user://`. Where is it on your computer? Use **Project > Open User Data Folder** in the editor. On Windows it is usually `%APPDATA%\Godot\app_userdata\<your project name>`. In code, `ProjectSettings.globalize_path("user://")` returns the real path:

```
# Fragment
print("Save folder: ", ProjectSettings.globalize_path("user://"))
```

Writing a text file

```
# Attach to: any Node. Call write_note() once, for example from _ready.
extends Node

const NOTE_PATH: String = "user://note.txt"

func write_note(text: String) -> void:
    var file: FileAccess = FileAccess.open(NOTE_PATH, FileAccess.WRITE)
    if file == null:
        push_error("Cannot open file: " + error_string(FileAccess.get_open_error()))
        return
    file.store_string(text)
    file.close()
```

`FileAccess.open` returns an object, or `null` if it failed (for example, a bad path or a full disk). **Always check for `null`.** `FileAccess.get_open_error()` tells you why. The modes are:

Mode	Meaning
`FileAccess.READ`	Read an existing file
`FileAccess.WRITE`	Create the file, or erase and overwrite it
`FileAccess.READ_WRITE`	Read and write an existing file

Mode	Meaning
`FileAccess.WRITE_READ`	Create or erase; read and write

Reading a text file

```gdscript
# Attach to: any Node
extends Node

const NOTE_PATH: String = "user://note.txt"

func read_note() -> String:
    if not FileAccess.file_exists(NOTE_PATH):
        return ""
    var file: FileAccess = FileAccess.open(NOTE_PATH, FileAccess.READ)
    if file == null:
        return ""
    var text: String = file.get_as_text()
    file.close()
    return text
```

Useful members:

Member	Use
`FileAccess.file_exists(path)`	Does the file exist? (static)
`file.store_string(text)`	Write text
`file.store_line(text)`	Write text plus a newline
`file.get_as_text()`	Read the whole file
`file.get_line()` / `file.eof_reached()`	Read line by line
`file.close()`	Finish (also happens automatically when the variable goes out of scope)

To delete a file, use `DirAccess.remove_absolute("user://note.txt")`. Create folders with `DirAccess.make_dir_recursive_absolute("user://saves")`.

> **Try it:** Call `write_note("hello")` from `_ready`, then `print(read_note())`. Open the user data folder and find `note.txt`. Open it with Notepad.

10.5 JSON

A game state is a bundle of numbers, strings, lists and sub-bundles. **JSON** is a text format for exactly that. It looks like a GDScript Dictionary:

```
{
    "version": 1,
    "score": 12,
    "player": { "x": 320.5, "y": 200.0 }
}
```

Godot gives you two main functions:

Function	Does
`JSON.stringify(data, indent)`	Converts a Dictionary/Array (or number, string…) to a JSON string. A non-empty `indent`, like `"\t"`, makes it pretty
`JSON.parse_string(text)`	Converts a JSON string back to a Variant. Returns `null` if the text is not valid JSON

```
# Attach to: any Node
extends Node

func _ready() -> void:
    var data: Dictionary = {
        "name": "Ada",
        "score": 12,
        "items": ["key", "bone"],
        "position": {"x": 320.5, "y": 200.0},
    }
    var text: String = JSON.stringify(data, "\t")
    print(text)

    var back: Variant = JSON.parse_string(text)
    if back is Dictionary:
        print("Score is ", back["score"])
        print("Second item is ", back["items"][1])
```

Expected output: the pretty JSON, then `Score is 12.0` and `Second item is bone`. Notice the `12.0`: the score went in as an integer and came back as a float (first gotcha below).

Gotchas (read these twice)

1. **Numbers come back as floats.** JSON has one number type. `12` returns as `12.0`. Convert with `int(value)` when you need an integer.
2. **Only basic types are allowed.** JSON has no `Vector2`, `Color` or `Node`. Store a Vector2 as `{"x":..,"y":..}` or `[x, y]` and rebuild it when loading.
3. **Keys are strings.** A dictionary with integer keys comes back with string keys.
4. **Parsing can fail.** Always check the result: `if parsed is Dictionary`.
5. **Missing keys.** Use `data.get("key", default)` instead of `data["key"]`, so old or damaged saves do not crash the game.

For error details, use the `JSON` object:

```
# Fragment: parse with an error message
var json: JSON = JSON.new()
var err: Error = json.parse(text)
if err != OK:
    push_error("JSON error at line %d: %s" % [json.get_error_line(), json.get_error_message()])
else:
    var data: Variant = json.data
```

A helper for vectors:

```
# Fragment: converting a Vector2 to and from a dictionary
func vector_to_dict(v: Vector2) -> Dictionary:
    return {"x": v.x, "y": v.y}

func dict_to_vector(d: Dictionary) -> Vector2:
    return Vector2(float(d.get("x", 0.0)), float(d.get("y", 0.0)))
```

10.6 ConfigFile for settings

Settings (volume, fullscreen, key bindings) fit a simpler format: sections and keys, like the `.ini` files of Windows. `ConfigFile` reads and writes them.

```
[audio]

master_volume=0.8

[display]

fullscreen=false
```

```gdscript
# Attach to: any Node
extends Node

const PATH: String = "user://settings.cfg"

func save_demo() -> void:
    var config: ConfigFile = ConfigFile.new()
    config.set_value("audio", "master_volume", 0.8)
    config.set_value("display", "fullscreen", false)
    var err: Error = config.save(PATH)
    if err != OK:
        push_error("Could not save settings: " + error_string(err))

func load_demo() -> void:
    var config: ConfigFile = ConfigFile.new()
    var err: Error = config.load(PATH)
    if err != OK:
        print("No settings file yet; using defaults")
        return
    var volume: float = config.get_value("audio", "master_volume", 1.0)
    var fullscreen: bool = config.get_value("display", "fullscreen", false)
    print(volume, " ", fullscreen)
```

The third argument of `get_value` is the default, used when the section or key is missing. `ConfigFile` can store Godot types like `Vector2` and `Color` directly (it writes them in Godot's own text format), which JSON cannot.

When to use which?

Data	Use
Settings the player changes in Options	`ConfigFile` in `user://settings.cfg`
Game progress: score, position, inventory	JSON in `user://savegame.json`
Data you ship: enemy stats, item definitions	Custom `Resource` `.tres` files

Settings and progress are kept in separate files on purpose: loading a game must not change the player's volume, and deleting a save must not reset their options.

10.7 Designing a save system

Before writing code, answer four design questions.

1. What to save

Save **the minimum needed to rebuild the situation**. Not the whole scene tree. Ask: "If the game were restarted and I applied this data, would the player feel it is the same moment?" For Coin Dash:

- Score and best score
- Player position
- (settings go into the separate file)

Do **not** save things that can be recomputed or reloaded from the project: textures, the scene layout, node paths, your constants.

2. A dictionary of state

Gather state in a dictionary of simple types, grouped by owner:

```
{
    "version": 1,
    "score": 12,
    "best_score": 40,
    "nodes": {
        "Player": { "x": 320.5, "y": 200.0 }
    }
}
```

3. Versioning

Your game will change. A save from version 1 may be loaded by version 3 of the game. Put a `"version"` number in every save from day one. On load:

```gdscript
# Fragment: inside a load function that has "data" (Dictionary) and a SAVE_VERSION constant
var version: int = int(data.get("version", 0))
if version > SAVE_VERSION:
    push_warning("Save file is from a newer game version")
    return false
if version < 2:
    # Example migration: version 1 had no "best_score"
    data["best_score"] = data.get("score", 0)
```

Migration code is a short chain of `if version < N` blocks that upgrade old data step by step. Because you use `data.get(key, default)`, a missing field never crashes.

4. Who knows how to save themselves

Two bad extremes: one huge function that reaches into every node (it breaks whenever you rename something), or every node writing its own file. A tidy middle path uses **groups** (Chapter 5):

- Any node that has something to save joins the group `saveable`.
- It provides two methods: `save_data() -> Dictionary` and `load_data(data: Dictionary)`.
- The save system asks the tree: `get_tree().get_nodes_in_group("saveable")`, collects each answer into the dictionary, keyed by node name.

The save system does not need to know what a player or enemy is. It only knows the contract. New things become saveable by joining the group and writing two small methods.

```gdscript
# Fragment: the collecting loop
var nodes_data: Dictionary = {}
for node in get_tree().get_nodes_in_group("saveable"):
    if node.has_method("save_data"):
        nodes_data[str(node.name)] = node.call("save_data")
```

> **Note:** Using the node name as a key works when each saveable node has a unique name in the scene. For many identical objects (like 100 coins), give each a unique id or save them as a list instead.

The GameState autoload

Data that must survive scene changes (score, best score, unlocked levels) belongs in an **autoload** (Chapter 5), a script Godot creates once and keeps for the whole run. Name it `GameState`. It is also the natural home of the save and load functions.

Autoload checklist:

1. Write the script, for example `game_state.gd`, with `extends Node`.
2. **Project > Project Settings > Globals** (the tab may be called **Autoload** in some 4.x versions).
3. Click the folder icon, choose the script, set **Node Name** to `GameState`, click **Add**.
4. From now on `GameState.score` works from any script.

> **Warning:** Do not give the script a `class_name GameState` as well. The autoload name is already a global name; a class with the same name causes a conflict error.

Project Settings, the tour

Open **Project > Project Settings**. Turn on **Advanced Settings** (top right) to see everything. The ones you will use in this course:

Path	What for
Application > Config > Name / Icon	The game name (also names the `user://` folder) and its icon
Application > Run > Main Scene	The scene that F5 starts
Display > Window	Window size, stretch mode (Chapter 8), fullscreen
Input Map	Define your actions and keys
Globals > Autoload	Autoload scripts and scenes
Layer Names > 2D Physics	Name your collision layers
Audio > Buses	Default bus layout file

Scripts can read settings too:

```gdscript
# Fragment
var game_name: String = ProjectSettings.get_setting("application/config/name")
print("Game: ", game_name)
```

The settings are stored in `project.godot`, a plain text file you can read and commit.

Lab 10: Save and load for Coin Dash

Time: 60 to 90 minutes.

Goal: a small Coin Dash game with a data-driven coin (a custom resource), a `GameState` autoload that saves and loads the score and player position as JSON in `user://`, and a `GameSettings` autoload that stores volume and fullscreen in a `ConfigFile`.

This lab is self-contained: it rebuilds a compact Coin Dash in a **new project** (call it `CoinDashSaves`) with its own scene tree and every script printed below. It is not the Chapter 7 or Chapter 8 version. Those projects have their own `Coin` class whose `collected` signal carries no value; here the signal carries the coin's value, so do not copy files over from them. Chapter 11 continues from this project. Create the Input Map actions only if you want to; the lab uses the built-in `ui_left`, `ui_right`, `ui_up` and `ui_down` (the arrow keys).

Files:

```
res://
├── coin_type.gd
├── coin_copper.tres
├── coin_gold.tres
├── coin.tscn / coin.gd
├── player.gd
├── game_state.gd        (autoload name: GameState)
├── game_settings.gd     (autoload name: GameSettings)
└── game.tscn / game.gd  (main scene)
```

Step 1: the resource

Create `coin_type.gd`:

```gdscript
class_name CoinType
extends Resource

@export var display_name: String = "Copper"
@export var value: int = 1
@export var color: Color = Color.WHITE

func describe() -> String:
    return "%s coin worth %d" % [display_name, value]
```

Wait until Godot has parsed the script (a moment), then create `coin_copper.tres` (`Copper`, value `1`, color `#b87333`) and `coin_gold.tres` (`Gold`, value `5`, color `#ffd700`) with **New Resource...** (Section 10.2).

Step 2: the coin

Scene `coin.tscn`:

```
Coin (Area2D)              <- script coin.gd
├── Sprite2D               <- Texture res://icon.svg, Scale (0.1, 0.1)
└── CollisionShape2D       <- CircleShape2D, Radius 13
```

`coin.gd`:

```gdscript
class_name Coin
extends Area2D
```

```gdscript
signal collected(value: int)

@export var coin_type: CoinType

@onready var sprite: Sprite2D = $Sprite2D

func _ready() -> void:
    if coin_type != null:
        sprite.self_modulate = coin_type.color
    body_entered.connect(_on_body_entered)

func get_value() -> int:
    if coin_type == null:
        return 1
    return coin_type.value

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        collected.emit(get_value())
        queue_free()
```

Step 3: the player

Script `player.gd` (attach to the `Player` node of Step 6):

```gdscript
extends CharacterBody2D

@export var speed: float = 320.0

func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")
    velocity = direction * speed
    move_and_slide()
    position = position.clamp(Vector2(16.0, 16.0), Vector2(1136.0, 632.0))

func save_data() -> Dictionary:
    return {"x": position.x, "y": position.y}

func load_data(data: Dictionary) -> void:
    position = Vector2(float(data.get("x", position.x)), float(data.get("y", position.y)))
```

The two last functions are the "saveable" contract from Section 10.7.

Step 4: GameState (autoload)

`game_state.gd`:

```gdscript
extends Node

signal score_changed(new_score: int)

const SAVE_PATH: String = "user://savegame.json"
const SAVE_VERSION: int = 1

var score: int = 0:
    set(value):
        score = value
        best_score = maxi(best_score, score)
        score_changed.emit(score)

var best_score: int = 0

func new_game() -> void:
```

```
    score = 0

func add_score(amount: int) -> void:
    score += amount

func has_save() -> bool:
    return FileAccess.file_exists(SAVE_PATH)

func save_game() -> bool:
    var nodes_data: Dictionary = {}
    for node in get_tree().get_nodes_in_group("saveable"):
        if node.has_method("save_data"):
            nodes_data[str(node.name)] = node.call("save_data")

    var data: Dictionary = {
        "version": SAVE_VERSION,
        "score": score,
        "best_score": best_score,
        "nodes": nodes_data,
    }

    var file: FileAccess = FileAccess.open(SAVE_PATH, FileAccess.WRITE)
    if file == null:
        push_error("Cannot write save: " + error_string(FileAccess.get_open_error()))
        return false
    file.store_string(JSON.stringify(data, "\t"))
    file.close()
    return true

func load_game() -> bool:
    if not has_save():
        return false
    var file: FileAccess = FileAccess.open(SAVE_PATH, FileAccess.READ)
    if file == null:
        push_error("Cannot read save: " + error_string(FileAccess.get_open_error()))
        return false
    var text: String = file.get_as_text()
    file.close()

    var parsed: Variant = JSON.parse_string(text)
    if not (parsed is Dictionary):
        push_warning("Save file is damaged; ignoring it")
        return false
    var data: Dictionary = parsed

    var version: int = int(data.get("version", 0))
    if version > SAVE_VERSION:
        push_warning("Save file is from a newer version of the game")
        return false

    score = int(data.get("score", 0))
    best_score = maxi(best_score, int(data.get("best_score", 0)))

    var nodes_data: Dictionary = data.get("nodes", {})
    for node in get_tree().get_nodes_in_group("saveable"):
        var key: String = str(node.name)
        if nodes_data.has(key) and node.has_method("load_data"):
            node.call("load_data", nodes_data[key])

    score_changed.emit(score)
    return true

func delete_save() -> void:
    if has_save():
        DirAccess.remove_absolute(SAVE_PATH)
```

Observe three design choices:

- Every file operation is checked for failure and reports it with `push_error`.
- Every value read from the file uses `data.get(key, default)` so a damaged save never crashes the game.
- The score setter keeps `best_score` in sync, so there is one place for that rule.

Step 5: GameSettings (autoload)

`game_settings.gd`:

```
extends Node

const SETTINGS_PATH: String = "user://settings.cfg"

var master_volume: float = 1.0
var fullscreen: bool = false

func _ready() -> void:
    load_settings()
    apply()

func apply() -> void:
    var bus: int = AudioServer.get_bus_index("Master")
    AudioServer.set_bus_volume_db(bus, linear_to_db(master_volume))
    var wanted_mode: DisplayServer.WindowMode = DisplayServer.WINDOW_MODE_WINDOWED
    if fullscreen:
        wanted_mode = DisplayServer.WINDOW_MODE_FULLSCREEN
    if DisplayServer.window_get_mode() != wanted_mode:
        DisplayServer.window_set_mode(wanted_mode)

func save_settings() -> void:
    var config: ConfigFile = ConfigFile.new()
    config.set_value("audio", "master_volume", master_volume)
    config.set_value("display", "fullscreen", fullscreen)
    var err: Error = config.save(SETTINGS_PATH)
    if err != OK:
        push_error("Could not save settings: " + error_string(err))

func load_settings() -> void:
    var config: ConfigFile = ConfigFile.new()
    var err: Error = config.load(SETTINGS_PATH)
    if err != OK:
        return
    master_volume = config.get_value("audio", "master_volume", 1.0)
    fullscreen = config.get_value("display", "fullscreen", false)
```

Register both autoloads: **Project Settings > Globals > Autoload**, add `res://game_state.gd` as `GameState` and `res://game_settings.gd` as `GameSettings`. Both must show **Enable** ticked.

Step 6: the game scene

Scene tree for `game.tscn`:

```
Game (Node2D)                       <- script game.gd
├── Player (CharacterBody2D)        <- script player.gd, Position (576, 324), Groups: player, saveable
│   ├── Sprite2D                    <- res://icon.svg, Scale (0.25, 0.25)
│   └── CollisionShape2D            <- RectangleShape2D, Size (32, 32)
├── Coins (Node2D)
└── Ui (CanvasLayer)
    ├── ScoreLabel (Label)          <- Position (16, 12)
    └── StatusLabel (Label)         <- Position (16, 44)
```

Add both groups to `Player` in the Node dock > Groups tab. Set `game.tscn` as the main scene.

`game.gd`:

```
extends Node2D

const COIN_SCENE: PackedScene = preload("res://coin.tscn")
const AREA_SIZE: Vector2 = Vector2(1152.0, 648.0)
const START_COINS: int = 5

@export var coin_types: Array[CoinType] = []

@onready var coins: Node2D = $Coins
@onready var score_label: Label = $Ui/ScoreLabel
@onready var status_label: Label = $Ui/StatusLabel

func _ready() -> void:
    print("Save folder: ", ProjectSettings.globalize_path("user://"))
    GameState.score_changed.connect(_refresh_score)
    GameState.new_game()
    _refresh_score(GameState.score)
    status_label.text = "Arrows: move    K: save    L: load    F: fullscreen    - / = : volume"
    for i in START_COINS:
        spawn_coin()

func spawn_coin() -> void:
    var coin: Coin = COIN_SCENE.instantiate()
    if not coin_types.is_empty():
        coin.coin_type = coin_types.pick_random()
    coin.position = Vector2(
        randf_range(40.0, AREA_SIZE.x - 40.0),
        randf_range(80.0, AREA_SIZE.y - 40.0)
    )
    coin.collected.connect(_on_coin_collected)
    coins.add_child(coin)

func _on_coin_collected(value: int) -> void:
    GameState.add_score(value)
    spawn_coin.call_deferred()

func _refresh_score(_value: int) -> void:
    score_label.text = "Score: %d    Best: %d" % [GameState.score, GameState.best_score]

func _unhandled_input(event: InputEvent) -> void:
    var key_event: InputEventKey = event as InputEventKey
    if key_event == null or not key_event.pressed or key_event.echo:
        return
    match key_event.keycode:
        KEY_K:
            var saved: bool = GameState.save_game()
            status_label.text = "Game saved." if saved else "Save failed (see Output)."
        KEY_L:
            var loaded: bool = GameState.load_game()
            status_label.text = "Game loaded." if loaded else "No usable save found."
        KEY_F:
            GameSettings.fullscreen = not GameSettings.fullscreen
            GameSettings.apply()
            GameSettings.save_settings()
        KEY_MINUS:
            _change_volume(-0.1)
        KEY_EQUAL:
            _change_volume(0.1)

func _change_volume(step: float) -> void:
    GameSettings.master_volume = clampf(GameSettings.master_volume + step, 0.0, 1.0)
    GameSettings.apply()
    GameSettings.save_settings()
    status_label.text = "Volume: %d%%" % roundi(GameSettings.master_volume * 100.0)
```

Select the `Game` node and in the Inspector fill **Coin Types** with `coin_copper.tres` and `coin_gold.tres` (set the array size to 2, then drag each file into a slot). Add `coin_copper.tres` twice to make gold rarer if you like.

Step 7: test

Run with **F5**. Check each item:

1. The Output shows the save folder path. Open it with **Project > Open User Data Folder**.
2. Collect a few coins. Copper coins are worth 1, gold worth 5. The score label updates.
3. Move the player to a recognizable spot and press **K**. The status says "Game saved." In the user folder, open `savegame.json` in Notepad and read it. You should see `version`, `score`, `best_score` and `nodes > Player`.
4. Walk away, collect more coins, press **L**. The player jumps back to the saved spot and the score returns to the saved value. The best score stays at its highest value.
5. Press **F** (fullscreen toggles; if nothing happens, your editor may be showing the game embedded in its own window, so switch off the game-embedding option of the **Game** workspace (Godot 4.4 and newer) and run again, or test fullscreen in the exported game of Chapter 11) and **-** / **=** (volume changes). Close the game completely and start it again: the fullscreen state and volume are remembered, because they were stored in `settings.cfg`.
6. Close the game. Open `savegame.json`, break it on purpose by deleting a closing brace, start the game and press **L**. The status says "No usable save found." and the Output shows a warning (Godot may also print its own JSON parse message). The game does not crash.
7. Delete `savegame.json`, press **L**: "No usable save found." again.

Expected result: progress saves with K and loads with L; settings persist between runs automatically; a damaged or missing file is handled gracefully.

Stretch goals (park them in your idea list if you are out of time): save the positions of the coins; add a `delete_save` key; show the version number on screen.

Chapter summary

- Nodes live in the tree and run code; Resources are shareable data objects that can be saved to files. Resources are shared by reference, so use `duplicate()` for a private copy.
- A custom Resource is a script with `extends Resource`, a `class_name` and `@export` variables; create instances as `.tres` files and edit them in the Inspector.
- `preload` loads at compile time with a constant path; `load` loads at run time. `PackedScene.instantiate()` makes a new copy of a scene.
- `user://` is for files made while playing; `res://` is read-only in an exported game. Always check that `FileAccess.open` did not return `null`.
- JSON via `JSON.stringify` and `JSON.parse_string` is the safe format for save games; numbers come back as floats and there are no Vector types.
- `ConfigFile` is the right tool for settings; keep settings and save games in separate files.
- Design saves around a version number, a dictionary of simple types, and nodes in a `saveable` group that provide `save_data` and `load_data`.
- A `GameState` autoload keeps data alive across scenes and owns the save code.

Review questions

1. Give two differences between a `Node` and a `Resource`.
2. Two `Sprite2D` nodes use the same `.tres` resource. You change a property of that resource at run time from one of them. What happens to the other, and how can you avoid it?
3. Which two lines are essential at the top of a script to make a custom resource class that appears in the New Resource dialog?
4. When would you use `load` instead of `preload`?
5. Why must a script write saves to `user://` and not `res://`?
6. What does `FileAccess.open(path, FileAccess.WRITE)` return if it fails, and what should your code do?
7. After saving `{"score": 12}` as JSON and parsing it back, what is the type of `score`, and how do you convert it to an integer?
8. True or false: JSON can store a `Vector2` directly.
9. Why is it risky to use `.tres` files as player save games?
10. What is the purpose of a `version` number in a save file?

Exercises

1. **Practice (easy):** Create a `SwordData` resource class with `display_name`, `damage` and `icon` (a `Texture2D`), make two `.tres` files, and print `describe()` for both from a test script.
2. **Practice (easy):** Write a function `read_json_file(path: String) -> Dictionary` that returns an empty dictionary if the file is missing, unreadable, or not valid JSON.
3. **Apply (medium):** Extend the lab so that the positions and types of all coins are saved and restored. Hint: putting every coin in the group `saveable` does not work well because the coins share similar names. Instead, add the `Coins` node to the group and give it its own `save_data` (return an array of coin positions) and `load_data` (delete the old coins, then spawn new ones).
4. **Apply (medium):** Connect the Chapter 8 options menu to `GameSettings` so the slider and the checkbox read their initial state from it and write changes through it.
5. **Challenge (hard):** Add three save slots. Write `save_game(slot: int)` and `load_game(slot: int)` that use `user://save_1.json`, `user://save_2.json`, etc., and a function that returns each slot's score and save time for a "Load game" screen. Include a `"saved_at"` field using `Time.get_datetime_string_from_system()`.

Common mistakes

- **"Invalid get index" or crash when loading an old save.** -> The code reads `data["key"]` for a key that does not exist. -> Use `data.get("key", default)` and add a version check.
- **Integers come back as `12.0`.** -> JSON numbers are floats. -> Wrap with `int(...)`.
- **The save works in the editor but not after export.** -> The code wrote to `res://`. -> Write to `user://`.
- `FileAccess.open` **returns `null` and the next line crashes.** -> The error was not checked. -> `if file == null:` print `FileAccess.get_open_error()` and return.
- **Changing one coin type's value changes all coins.** -> They share one Resource. -> That is by design; use `duplicate()` if one coin needs different values.
- **"Identifier GameState not declared".** -> The autoload is not registered, is disabled, or the name differs. -> Check Project Settings > Globals > Autoload for the exact name and the Enable box.

- **Loading changes nothing in the scene.** -> The node was not in the `saveable` group, or the node name in the save does not match the name in the scene. -> Check Groups and the names.

Further reading

- Resources: https://docs.godotengine.org/en/stable/tutorials/scripting/resources.html
- Saving games: https://docs.godotengine.org/en/stable/tutorials/io/saving_games.html
- File paths in Godot projects (`res://` and `user://`): https://docs.godotengine.org/en/stable/tutorials/io/data_paths.html
- FileAccess class reference: https://docs.godotengine.org/en/stable/classes/class_fileaccess.html
- JSON class reference: https://docs.godotengine.org/en/stable/classes/class_json.html
- ConfigFile class reference: https://docs.godotengine.org/en/stable/classes/class_configfile.html
- Singletons (Autoload): https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html
- Project Settings: https://docs.godotengine.org/en/stable/tutorials/editor/project_settings.html

Chapter 11: Professional Habits

Learning objectives

By the end of this chapter you will be able to:

- Read a Godot error message and a stack trace and find the line that caused the problem.
- Debug a running game with `print`, `assert`, breakpoints, the Debugger panel, the Remote scene tree, the Monitors and the Profiler.
- Put a Godot project under version control with git, using a correct `.gitignore`, and use commits and branches as a safety net.
- Organize a project with consistent folders, names and small, self-contained scenes.
- Plan a small game with a clear scope and a vertical slice, and run a playtest.
- Export a game as a Windows executable and test it on a clean setup.
- Credit third-party assets correctly and use an AI assistant as a tutor without giving away your own learning.

Key terms

- **Bug**: behavior that differs from what you intended.
- **Stack trace**: the list of function calls that led to an error, newest first.
- **Breakpoint**: a marker on a line of code where the running game stops so you can look around.
- **Assertion**: a check that stops the game in a debug build if a condition you rely on is false.
- **Remote scene tree**: a live view of the scene tree of the running game, shown in the editor.
- **Profiler**: a tool that measures how much time each function takes.
- **Version control**: a system that records the history of your files so you can go back or compare.
- **Commit**: one saved snapshot in git, with a message.
- **Branch**: a separate line of commits where you can try things without touching the main line.
- **Vertical slice**: one small part of the game finished to final quality, from start to end.
- **Export template**: the prebuilt game runtime that Godot packs together with your project to make an executable.
- **PCK file**: the package file holding your game's resources; it can sit next to the `.exe` or be embedded in it.

11.1 Debugging

Every programmer writes bugs. What separates beginners from experienced developers is not that the experienced ones write fewer, but that they find them faster. Finding bugs is a skill with tools and a method.

The method

1. **Reproduce** the bug. Find the exact steps that make it happen every time.
2. **Read** the message. If there is an error, read all of it, slowly.
3. **Guess** one cause. Say it out loud: "I think `score_label` is null because the path is wrong."
4. **Test** the guess with a `print`, a breakpoint or a look at the Remote tree.
5. **Fix one thing**, run again, and confirm.

Do not change five things at once. Then you will not know which one worked.

The Output panel

The **Output** panel (bottom of the editor) shows everything your game prints. Your own messages appear in plain text. Problems appear in color:

- **Red** lines are errors. The script usually stops running the failing function.
- **Yellow** lines are warnings. The game continues but something is suspicious.

Useful ways to write to it:

```
# Attach to: any Node
extends Node

func _ready() -> void:
    print("Plain message")
    print("Score: ", 12, " lives: ", 3)
    print("Position is %s" % Vector2(10, 20))
    push_warning("This looks odd, but I can go on")
    push_error("Something is really wrong")
```

`push_warning` and `push_error` also appear in the Debugger's **Errors** tab with the script line. Use them for real problems; use `print` for temporary investigation.

Print debugging

Adding temporary `print` lines is the oldest debugging tool and still very effective. Good habits:

- Print a label with the value: `print("speed = ", speed)`, not just `print(speed)`.
- Print where you are: `print("entering _on_coin_collected")`.
- Remove the prints when you are done, or they bury real messages.

```
# Fragment: a coin handler with debugging prints
func _on_coin_collected(value: int) -> void:
    print("coin collected, value = ", value, ", score before = ", GameState.score)
    GameState.add_score(value)
    print("score after = ", GameState.score)
```

Assertions

`assert(condition, message)` checks something that **must** be true. In a debug run (from the editor), if the condition is false the game stops at that line and shows the message. It is a way of writing down your assumptions.

```
# Fragment
func add_score(amount: int) -> void:
    assert(amount > 0, "add_score needs a positive amount, got %d" % amount)
    score += amount
```

> **Warning:** Assertions are removed in release exports. Never put real work inside the assert condition (like `assert(do_something())`), and never use `assert` to check user input or files. Use `if` and `push_error` for those.

Reading an error message

Take this error. It is **illustrative**: the file names, line numbers and times will be different on your computer, and the exact layout changes between 4.x versions.

```
E 0:00:02:0412   coin.gd:16 @ _ready(): Invalid access to property or key 'color' on a base object of
type 'Nil'.
  <GDScript Error>Invalid_Access
  <GDScript Source>coin.gd:16 @ coin.gd::_ready()
  <Stack Trace>  coin.gd:16 @ _ready()
                 game.gd:35 @ spawn_coin()
                 game.gd:24 @ _ready()
```

Read it in this order:

1. **Where:** `coin.gd:16` is the file and line (click it in the editor to jump there).
2. **What:** "Invalid access to property or key 'color' on a base object of type 'Nil'." You tried to read `.color` on something that is `null`. ("Nil" or "null instance" means "no object".)
3. **How we got here:** the **stack trace**, newest first. `_ready()` in `coin.gd` was called while `spawn_coin()` in `game.gd` was running, which was called by the `_ready()` of the game. The bug appears at line 16, but the cause may be higher up: who gave `coin` a null `coin_type`?

The exact wording changes a little between 4.x versions, but the pattern is the same: **file:line, what went wrong, the chain of calls**.

A small dictionary of common messages:

Message (shortened)	Usual cause	What to check
`Invalid access to property or key 'x' on a base object of type 'Nil'` (or `'null instance'`)	A variable is null	Was the node found? Was the resource assigned?
`Node not found: "A/B" (relative to "/root/Game")`	`$A/B` path wrong	Compare with the Scene dock; use unique names `%`
`Invalid call. Nonexistent function 'foo' in base 'X'`	Typo, or wrong type of object	Spelling; does the object have that method?
`Identifier "x" not declared in the current scope`	Typo or variable defined elsewhere	Spelling, where it is declared
`Trying to assign value of type 'String' to a variable of type 'int'`	Type mismatch with static typing	Convert with `int()` / `str()`
`Out of bounds get index '5' (on base: 'Array')`	Index larger than the list	`array.size()`, loops
mentions `previously freed`	Using a node after `queue_free()`	Check with `is_instance_valid(node)`
`Mixed use of tabs and spaces for indentation`	Pasted code with spaces	Use tabs only (Edit > Convert Indent to Tabs in the script editor)
`Cannot infer the type of "x" variable`	`:=` used where the type is unknown	Write the type: `var x: float = ...`

> **Note:** Static typing is a debugging tool. With `var speed: float`, many mistakes become red underlines in the script editor *before* you run the game.

The Debugger panel and breakpoints

When you run from the editor, the **Debugger** panel at the bottom has several tabs: **Stack Trace**, **Errors**, **Evaluator**, **Profiler**, **Visual Profiler**, **Network Profiler**, **Monitors**, **Video RAM** and **Misc**. You will mostly use Errors, Stack Trace (which also shows the variables), Profiler and Monitors. The exact tab list can differ a little between 4.x versions.

A **breakpoint** pauses the game on a line. To set one:

1. Open the script. Click in the narrow margin left of the line numbers on the line you want. A red dot appears. (Or put the cursor on the line and press **F9**.)
2. Run the game with **F5** (or F6).
3. When the line is about to run, the game freezes and the editor comes to the front, highlighting the line.

Now look around:

- **Stack Trace** list: the chain of function calls. Click an entry to jump to that line.
- **Variables** list (inside the Stack Trace tab, next to the call chain): locals, members and globals with their current values. Expand objects to see their properties. Compare to what you expected.
- Use the toolbar buttons (hover them to see names and shortcuts): **Step Over** (run this line, stay in this function), **Step Into** (go inside the function being called), **Continue** (run until the next breakpoint), and **Break** (pause now).

You can also write the word `breakpoint` on its own line in a script; it pauses at that line when running from the editor.

```gdscript
# Fragment: pause here to look at 'value' and the player
func _on_coin_collected(value: int) -> void:
    breakpoint
    GameState.add_score(value)
```

> **Try it:** Open the Chapter 10 `game.gd`, click the margin next to `GameState.add_score(value)` to set a breakpoint, run, collect a coin, and read `value` in the **Variables** list of the Stack Trace tab. Is it 1 or 5? Press Continue.

Remote scene tree and live editing

While the game runs, the **Scene** dock shows two buttons at the top: **Local** and **Remote**. Click **Remote** to see the real tree of the running game, including nodes you create from code (coins, particles). Click a node and its properties appear in the Inspector; you can change them while the game runs (for example move the player, or change `speed`). Changes are not saved, which makes this ideal for experiments.

This answers questions like "Did my bullet really get added?" and "Why are there 300 coins?" Look in the Remote tree.

Monitors and the Profiler

The **Monitors** tab draws graphs: frames per second (FPS), process time, physics time, memory, **Objects**, **Nodes** and **Orphan Nodes**. Two patterns to recognize:

- **Nodes keeps rising** while nothing should be added: something creates nodes and never frees them.
- **Orphan Nodes > 0 and rising**: nodes were created but never added to the tree and never freed. Call `queue_free()` or add them.

The **Profiler** tab tells you where time goes:

1. Run the game from the editor.
2. Open Debugger > Profiler and click **Start**.
3. Do the slow thing in the game for a few seconds.
4. Click **Stop**. A table lists functions with the **time** they used (total, and self-time without the functions they call) and the number of **calls**.
5. Sort by self time. The first rows are your suspects.

> **Note:** Measure first, optimize second. Beginners often guess the wrong cause. Typical real causes of slowness: thousands of nodes, a heavy loop in `_process`, searching the tree (`get_node`, `find_child`) every frame, or `print` calls every frame.

Two more editor helps

- **Debug > Visible Collision Shapes** draws all collision shapes in the running game. If the player passes through a wall or a coin does not trigger, turn this on first.
- **Project Settings > Debug > GDScript** has warnings you can raise to errors, for example "unused variable" and "untyped declaration".

11.2 Version control with git

Why

Imagine every time you finish something that works you could take a photo of the whole project. Later you can compare, go back, or branch off to try something risky. That is what **git** does. It also protects you from a bad AI edit, a mistaken delete, and a dead hard drive (when you push to a remote copy).

Install and first use

Install **Git for Windows** from git-scm.com (the default options are fine). Open a terminal in your project folder and run:

```
git init
git status
```

`git init` creates a hidden `.git` folder that holds the history. `git status` shows what changed.

Tell git who you are, once per computer:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

What to commit, what to ignore

Godot makes some files that should not be in version control, because they are generated and big.

Commit (keep)	Ignore (do not commit)
`project.godot`	`.godot/` (the cache; Godot rebuilds it)
Your `.gd`, `.tscn`, `.tres` files	`/android/` (generated Android build files)
Assets: images, sounds, fonts	Export outputs (for example `builds/`)
The `.import` files next to assets (`icon.svg.import`)	Large files you do not need to share
`.uid` files next to scripts (Godot 4.4 and newer)	Passwords, keys, tokens
`export_presets.cfg` (see below)	Editor temp files

> **Warning:** The `.import` files and (in Godot 4.4+) the `.uid` files **must** be committed. They record how each asset was imported and the permanent id of each script. It is only the `.godot/` folder that is ignored.

Create a file named `.gitignore` in the project root:

```
# Godot 4 generated files
.godot/
/android/
*.translation

# Our exported games
builds/

# Operating system junk
Thumbs.db
.DS_Store
```

The official Godot documentation ("Version control systems") lists two things to ignore: `.godot/`, the project cache, and `*.translation`, the binary translation files that Godot generates from CSV files. This course has no CSV translations, so the second line changes nothing for you, but it is harmless and it is in the official list. The `/android/` line comes from the `.gitignore` that Godot writes itself for new projects; it matters only if you install the Android build template. If you use C#, add the usual .NET build folders (`bin/`, `obj/`) too; this course uses only GDScript.

If you enabled **Version Control Metadata** when you created the project (an option in the new-project dialog), Godot has already written a `.gitignore` and a `.gitattributes` for you. Check them against the above.

A `.gitattributes` file keeps line endings consistent between Windows and other systems:

```
# Normalize line endings
* text=auto eol=lf
```

About `export_presets.cfg`: this file stores your export settings (platform, options, which files to include). It is useful to keep so that every machine exports the same way. In Godot 4.1 and newer, passwords and keys (for example an Android signing key) are stored separately in `.godot/export_credentials.cfg`, which is inside the

ignored `.godot/` folder. (Godot 3 and 4.0 kept them in `export_presets.cfg`.) Still, open `export_presets.cfg` once and check that it contains no secrets before you push to a public place. Never commit keystores or passwords.

The daily loop

```
git add -A
git commit -m "Add coin types as resources"
```

- `git add -A` stages every change (new, edited, deleted files).
- `git commit -m "message"` records a snapshot with your message.

Look at history and changes:

```
git log --oneline
git diff
```

`git log --oneline` lists commits with short ids. `git diff` shows exactly what changed since the last commit, line by line. Read it before every commit; it takes ten seconds and catches mistakes.

Undo:

```
git restore player.gd
```

throws away uncommitted changes to `player.gd` and returns it to the last commit. (This cannot be undone, so be sure.)

Commit messages: short, say what changed and why, in the present tense. "Fix coin spawn crash when no types are assigned" is a good message. "stuff" is not.

> **Why it works:** git stores your files' history, not just the latest version. Because a commit is a complete snapshot with a message, you can always find the last time something worked.

Branches

A branch is a safe sandbox:

```
git switch -c try-new-player
```

creates a branch and moves to it. Work and commit freely. If you like the result:

```
git switch main
git merge try-new-player
```

If you do not:

```
git switch main
git branch -D try-new-player
```

Your main line never saw the experiment. (If your first branch is called `master`, either name works; use whichever your `git status` shows, or rename with `git branch -M main`.)

Remote copy

A remote repository (GitHub, GitLab, and others) is an off-site backup and a way to share. Create an empty **private** repository on the site, then:

```
git remote add origin https://github.com/YOUR-NAME/coin-dash.git
git branch -M main
git push -u origin main
```

After the first time, `git push` is enough. Keep the repository private unless you have checked the license of every asset in it.

When Godot is open while you switch branches

Godot watches the files. If a branch switch changes a scene that is open, the editor asks whether to reload. Choose **Reload** (or close and reopen the project). Commit or stash your own changes first.

11.3 Project structure and naming

Folders

Pick a layout early and keep it. Two common styles:

By type:

```
res://
├── scenes/
├── scripts/
├── art/
├── audio/
└── data/
```

By feature (recommended for games that grow):

```
res://
├── player/
│   ├── player.tscn
│   └── player.gd
├── coins/
│   ├── coin.tscn
│   ├── coin.gd
│   ├── coin_type.gd
│   ├── coin_copper.tres
│   └── coin_gold.tres
├── ui/
│   ├── main_menu.tscn
│   └── options_menu.tscn
├── autoload/
│   ├── game_state.gd
│   └── game_settings.gd
├── audio/
└── levels/
```

With the feature layout, everything about the coin is in one place. A scene and its script live next to each other. Move files **inside the Godot editor** (drag in the FileSystem dock), never in Windows Explorer, so that Godot can update all references.

Naming rules

Godot's official style guide uses these:

Thing	Style	Example
Files and folders	`snake_case`	`main_menu.tscn`, `coin_type.gd`
Node names in the scene	`PascalCase`	`ScoreLabel`, `Player`
Classes (`class_name`)	`PascalCase`	`CoinType`
Variables and functions	`snake_case`	`time_left`, `spawn_coin()`
Private members	leading underscore	`_tween`, `_on_timer_timeout()`
Constants and enum values	`CONSTANT_CASE`	`MAX_SPEED`, `SAVE_PATH`
Signals	past tense verbs	`died`, `collected`, `door_opened`

Why this matters: Windows ignores upper/lower case in file names, but exported games on other systems do not, so `Player.tscn` vs `player.tscn` can work on your machine and fail elsewhere. Using lowercase file names everywhere avoids that class of bug.

Scene composition: small scenes that stand alone

A good scene can be opened and run (F6) **alone**. Build the game from many small scenes (`player.tscn`, `coin.tscn`, `hud.tscn`) and instance them into bigger ones.

The rule that keeps scenes independent is **call down, signal up**:

- A parent may call methods and set properties of its children (down), because it knows them.
- A child should not reach up to its parent with `get_parent()`. Instead it emits a **signal**, and whoever owns it decides what to do (up).

Example: a reusable health component.

```
# health.gd. Attach to a Node named "Health" that is a child of Player (or Enemy).
class_name Health
extends Node

signal changed(current: int, maximum: int)
signal died

@export var maximum: int = 3

var current: int = 0

func _ready() -> void:
    current = maximum

func take_damage(amount: int) -> void:
    if current <= 0:
        return
    current = maxi(current - amount, 0)
    changed.emit(current, maximum)
```

```
    if current == 0:
        died.emit()
```

The player scene listens to it:

```gdscript
# Attach to: CharacterBody2D "Player" that has a child Node "Health" (health.gd)
extends CharacterBody2D

@onready var health: Health = $Health

func _ready() -> void:
    health.died.connect(_on_died)

func _on_died() -> void:
    print("Player died")
    queue_free()
```

`Health` knows nothing about the player. You can reuse it on enemies, crates, doors. The player connects to its signals. This is called **composition**: building objects from small parts instead of one giant script.

Other composition habits:

- **Exports for dependencies.** If a node needs another node, give it `@export var target: Node2D` and assign it in the Inspector, instead of a long `get_node("../../Other")` path that breaks when you move things.
- **Groups for broadcasts.** `get_tree().call_group("enemies", "freeze")` tells all enemies to freeze, with no list to maintain.
- **Few autoloads.** Use them for truly global things (`GameState`, `GameSettings`, an audio manager). A dozen autoloads is a sign the design is leaking.
- **Data in resources.** Numbers you want to tweak (enemy speed, coin value) belong in `.tres` files or exports, not hard-coded in functions.

11.4 Designing a small game

Start with one sentence

"You are a ____ who must ____ before ____." For Coin Dash: "You are a square who must grab coins before time runs out." If you cannot say it in a sentence, the game is not clear yet.

The core loop

Write the repeating cycle in three or four steps. Coin Dash: *move -> reach coin -> score and new coin appears -> repeat*. If the core loop is not fun with gray boxes and no sound, art and polish will not rescue it. Build and test that first.

Scope: the beginner's biggest enemy

Your ideas are bigger than your time. That is normal, and healthy. The skill is choosing. Make three lists:

List	Contains	Example for Coin Dash
Must	The game does not exist without it	Move, coins, score, timer, restart
Should	Makes it good	Menu, sounds, best score, save

List	Contains	Example for Coin Dash
Could	Nice, if there is time	Power-ups, enemies, levels, music

Build all of Must first. Then Should. Park everything else in an ideas list instead of starting it. A common rule: take your estimate of the time and double it.

The vertical slice

A **vertical slice** is one small part of the game finished all the way: art, sound, menu, gameplay, and polish. Not "all the levels in gray boxes", but "one level that looks and plays like the final game". It proves that the whole process works and tells you honestly how long the rest will take.

For *Weird Hotel* in Part III, the slice is: wake up in the bedroom, open the drawer, take the key, unlock the door, walk into the hallway. If those five things feel good, the other rooms are more of the same.

Playtesting

Your own opinion of your game is the least reliable one, because you know all the secrets.

How to run a playtest:

1. Find 3 to 5 people who have not seen the game. Friends are fine.
2. Give them the build, say only "play this", and **do not help**.
3. Watch their screen and face. Take notes: where they pause, hesitate, laugh, or get stuck.
4. Do not defend the game. Do not explain. If they are confused, that is the result.
5. Afterwards ask: "What did you think you were supposed to do?" and "What was the most annoying part?"
6. Make a list of problems, sorted by how many testers hit each one. Fix the top ones, then test again with new people.

Five testers find most of the major problems. After that you mostly see repeats.

11.5 Exporting to Windows

Running from the editor is not a game your friends can play. **Exporting** packs your project and the Godot runtime into files that run without the editor.

Step 1: export templates

The runtime parts are called **export templates**. They are a separate download that must match your exact Godot version.

1. In the editor, open **Editor > Manage Export Templates**.
2. Click **Download and Install** and wait for it to finish. (If you are offline, download the template package from godotengine.org and use **Install From File**.)

Step 2: add a preset

1. Open **Project > Export...**
2. Click **Add...** and choose **Windows Desktop**.

3. If the dialog warns about missing templates, go back to Step 1.

Step 3: set options

- **Export With Debug:** leave this off for a release build to get a smaller and faster game. Keep it on while you are still searching for problems (error messages and the debug console still work).
- **Binary Format > Embed PCK:** turn this **on** to get a single `.exe` file. When off, you get an `.exe` plus a `.pck` file that must stay together.
- **Application > Icon, Product Name, Company Name:** metadata for the file. Godot converts your project icon for the `.exe`. Setting the other file details (and in some versions the icon) may need a helper tool called *rcedit*; see the "Exporting for Windows" page in Further reading and **Editor > Editor Settings > Export > Windows**. This is optional, and the game works without it.
- **Resources tab:** by default all resources are exported. If your game reads extra files that are not Godot resources (for example `.txt` or `.json` files in `res://`), add them in **Filters to export non-resource files**, such as `*.json`.

Step 4: export

1. Click **Export Project...**
2. Choose a folder for the build, for example a `builds` folder in the project (git ignores it) or any folder outside the project, and the file name `CoinDash.exe`.
3. Make sure **Export With Debug** matches your choice, then **Save**.

Step 5: test like a stranger

The game must work on a computer that does not have Godot.

1. Copy the `.exe` (and `.pck` if you did not embed it) to a new empty folder, for example on your Desktop.
2. Double-click the `.exe`. The game starts.
3. If possible, try it on another PC.
4. Check the saves: after saving, look in `%APPDATA%\Godot\app_userdata\<your project name>`. The exported game uses the same `user://` folder as the editor.
5. If the game closes immediately, export again with **Export With Debug** on and start the `.exe` from a terminal (`.\CoinDash.exe`) to read the messages. (If the preset has the **Debug > Export Console Wrapper** option, the exporter also writes a `CoinDash.console.exe` that shows a console window.) Missing files are the usual cause: something was not included.

> **Note:** Windows may show a blue **SmartScreen** warning ("Windows protected your PC") for programs from unknown publishers. That is normal for an unsigned game from a new developer. The player can choose "More info > Run anyway". Signing a program with a certificate is possible but not part of this course.

Other platforms, briefly

The same Export dialog covers other targets; each needs extra setup that we do not cover here.

- **Web:** exports an HTML page you host on a web server (it does not run correctly when opened as a plain file, and some features need specific server headers).
- **Android:** needs the Android SDK, a Java development kit, and a signing key.

- **Linux and macOS:** presets exist; macOS builds have extra signing requirements.

Always read the official export documentation for the platform before you start.

11.6 Licenses and credits

Every file in your game comes from somewhere. Your responsibility is to know where, and to follow its license. (This is general information, not legal advice.)

License	In plain words
CC0 / public domain	Free to use, no credit needed. Credit anyway as a courtesy
CC BY	Free to use, but you must credit the author (and often link the license)
CC BY-SA	Like CC BY, and changes you share must use the same license
MIT / Apache (common for code)	Free to use; keep the copyright notice
"Free for personal use" or no license	You may not put it in a game you share. Ask the author

Habits:

1. Keep a `CREDITS.md` file in the project. Add a line **when you add the asset**, not at the end: file, author, source link, license.
2. Copy the license text if the license asks for it.
3. Show credits in the game (a Credits screen using a `RichTextLabel`).
4. Godot itself is under the MIT license, and the engine includes third-party components. Godot asks that you include its license notice somewhere accessible. You can read the text from code with `Engine.get_license_text()`, and show it on a credits page:

```
# Attach to: a RichTextLabel named "LicenseText" on a Credits screen
extends RichTextLabel

func _ready() -> void:
    bbcode_enabled = false
    text = Engine.get_license_text()
```

Expected result: a long text with the MIT license and the licenses of components used by Godot.

An example of `CREDITS.md`:

```
# Credits

- Pickup sound: "coin.wav" by Some Author, CC0, https://opengameart.org/...
- Font: "Example Sans" by Foundry Name, SIL Open Font License, https://...
- Everything else: made by YOUR NAME with Godot Engine (MIT license).
```

If you use art or text generated by an AI tool, read that tool's terms about commercial use and credit, and note it in your credits too.

11.7 Using an AI assistant as a tutor, responsibly

An AI assistant can explain, check and encourage. It can also quietly do all the work and leave you unable to make the next game alone. The difference is in how you ask.

The explain-then-try loop

1. **Try first.** Write your own attempt, even if it is ugly or half finished.
2. **Ask for an explanation, not a solution.** "Explain what `move_and_slide` does with `velocity`." or "Why does this line give a null error? Do not fix it yet."
3. **Predict.** Before running, say what you expect to happen.
4. **Run and compare.** If the result differs from your prediction, that gap is where you learn.
5. **Write the fix yourself,** then ask the assistant to review it.
6. **Explain it back** in one or two sentences. If you cannot, ask again.

Good questions

- "Here is my error and my script. What does the error mean? Give me a hint, not the answer."
- "Explain this script line by line."
- "I think this bug is because of X. Am I right?"
- "What would break if I removed this line?"
- "Give me three ways to do this and the trade-offs."
- "Check this change for mistakes. Do not rewrite it."

Make it your tutor with standing instructions

Tools like Claude Code can read a project instructions file (`CLAUDE.md`) every time. Write your learning rules once:

```
# How to help me on this project
- I am a beginner learning Godot 4.
- Act like a patient teacher. Use short, plain sentences.
- Explain why something works or fails, not just the fix.
- Before you edit a file, tell me what you plan to change and why.
- Give hints first. Give the full answer only when I ask for it.
- Keep changes small. One thing at a time.
- Check the official Godot 4 documentation instead of guessing.
```

Trust, but verify

AI assistants make confident mistakes. Specific risks for Godot learners:

- **Old Godot 3 code.** If you see `onready`, `yield`, `export var`, `KinematicBody2D` or `connect("signal", self, "method")`, it is Godot 3, not Godot 4.
- **Invented functions.** The name may look right and not exist. Check the class reference at docs.godotengine.org.
- **Code that does not fit your scene.** Node names and paths are specific to your project.

Defend yourself with the habits from this chapter:

1. **Commit before you let anyone, human or AI, change many files.** Then `git diff` shows exactly what changed, and `git restore` undoes it.
2. **Read the diff.** If you cannot explain a change, ask.
3. **Run it.** The Output panel is the judge, not the assistant's confidence.
4. **Do not paste secrets** (passwords, keys, tokens, private personal data) into any chat.
5. **Follow your course's rules** about when AI help is allowed and how to disclose it. Learning is the goal of this book; the assistant is the study partner, not the student.

Lab 11: Break it, fix it, ship it

Time: 60 to 90 minutes.

Goal: start from your finished Coin Dash of Chapter 10, put it under version control, break it on purpose in four ways, find each bug with the debugging tools, fix it, and export a Windows `.exe` that you test in a clean folder.

If you did not do Lab 10, build it first (the `CoinDashSaves` project). This lab quotes its files: `game.gd`, `coin.gd`, `game_state.gd`, `game_settings.gd`, `player.gd`, `coin_type.gd`.

Part A: version control

1. In **Project > Project Settings > Application > Config**, set **Name** to `Coin Dash`.
2. Create `.gitignore` and `.gitattributes` in the project folder with the contents from Section 11.2 (you can create the files with any text editor; make sure the names start with a dot and have no `.txt` ending).
3. In a terminal opened in the project folder:

```
git init
git add -A
git status
```

4. Read the `git status` list. You should see `project.godot`, your scripts, scenes, `.tres` files and `icon.svg` plus `icon.svg.import`. You should **not** see `.godot/`.
5. Commit:

```
git commit -m "Coin Dash: coins as resources, save and settings"
git branch -M main
```

(`git branch -M main` makes sure the main line is called `main`, whatever your git version chose by default.)

6. Create a branch for the experiments:

```
git switch -c break-things
```

Part B: four breakages

For each one: make the change, run the game, and use the tools to find the cause yourself. Write what you see in a text file: the symptom, the tool that showed the cause, and the fix.

Break 1: a wrong node path. In `game.gd` change `$Ui/ScoreLabel` to `$Ui/ScoreLable` (a typo).

- Run. Read the **Output** and the **Debugger > Errors** tab. You will see a message that the node was not found, followed by an error about accessing a property on null when the score is shown.
- Which line does the first error point to? Which line does the second point to? The first is the cause; the second is the consequence.
- Look at the **Stack Trace**. Click each entry.
- Fix the typo, run, and confirm the error is gone.

Break 2: a null coin type. In `coin.gd`, change `_ready` so it no longer checks for null:

```
func _ready() -> void:
    sprite.self_modulate = coin_type.color
    body_entered.connect(_on_body_entered)
```

Then select the `Game` node and clear the **Coin Types** array in the Inspector (set its size to 0).

- Run. Read the error: it names `coin.gd`, a line, and a base object of type `Nil` (or `null instance`).
- Open the **Stack Trace**: the next entry up is `spawn_coin()` in `game.gd`, and above that `_ready()`. The coin crashed during `add_child`, because that is when `_ready` runs.
- Why is `coin_type` null? (Because nobody assigned one.) Where should the guard be? (In the coin, which cannot trust its creator.)
- Restore the check (`if coin_type != null:`), run again, and the game works with plain coins.

Break 3: an assertion. In `game_state.gd`, add this as the first line of `add_score`:

```
    assert(amount > 0, "add_score needs a positive amount, got %d" % amount)
```

Then open `coin_gold.tres` and set `value` to `0`.

- Run and collect a gold coin (coins pick their type at random, so collect until a gold one appears). The game stops at the assert with your message. Look at the **Variables** list in the Stack Trace tab to confirm `amount` is `0`.
- Decide: is the data wrong, or the assumption? Here we decide that zero-value coins are a data mistake. Set the value back to `5`.
- Think: what would happen in an exported release build? (The assertion would be removed and nothing would stop the bad data.)

Break 4: a leak you can see. In `coin.gd`, comment out `queue_free()` in `_on_body_entered`:

```
func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        collected.emit(get_value())
        #queue_free()
```

- Run. Collect a coin, step off it, step on it again. The score rises again and another coin appears each time.
- Press **Remote** in the Scene dock, and open `Game > Coins`. Count the children. Collect more and watch the number go up.
- Open **Debugger > Monitors** and select **Nodes**. The graph rises as you play.
- Use **Debug > Visible Collision Shapes** to see the old coin areas that never left.
- Restore `queue_free()`.

Part C: review and clean up

1. In the terminal run `git diff`. You will see all your breakages as red and green lines.
2. Everything should be restored by now. If `git diff` shows any difference, decide if it is a wanted fix or leftover damage.
3. Throw away the experiment branch:

```
git restore .
git switch main
```

```
git branch -D break-things
```

(`git restore .` discards uncommitted changes on the branch. The `-D` deletes the branch and its commits that were never merged. Use it only because you want to throw the experiment away.)

4. Now make one real improvement on `main`: set the **Main Scene**, make sure the window **Stretch Mode** is `canvas_items` (Chapter 8), and commit it with a clear message.

Part D: export

1. **Editor > Manage Export Templates > Download and Install** (once).
2. **Project > Export... > Add... > Windows Desktop.**
3. Turn on **Binary Format > Embed PCK**.
4. Click **Export Project...**, choose `builds/CoinDash.exe`, leave **Export With Debug** off, and save. (If you create the `builds` folder inside the project, add an empty file named `.gdignore` to it so the editor does not scan your exported files.)
5. Create a new empty folder on your Desktop called `CoinDash-test`. Copy only `CoinDash.exe` into it.
6. Double-click it. Play, press **K**, close the game, start it again, press **L**.
7. Open `%APPDATA%\Godot\app_userdata\Coin Dash` in Explorer and find `savegame.json` and `settings.cfg`.
8. Commit: `git add -A` then `git commit -m "Add export preset"`. (`builds/` is ignored, so the `.exe` is not in the repository.)

Expected result

- A git history with at least two commits, a `.gitignore` that excludes `.godot/` and `builds/`, and no leftover breakages.
- A text file with your four debugging notes: the symptom, the tool that showed the cause, and the fix.
- A working `CoinDash.exe` that runs from a clean folder, saves, loads, and remembers settings.

Chapter summary

- Debug with a method: reproduce, read, guess one cause, test, fix one thing.
- An error message gives the file and line, the problem, and a stack trace of how the code got there; the cause is often further up the trace than the line shown.
- Use `print`, `assert`, breakpoints with the Stack Trace tab and its variables, the Remote scene tree, Monitors and the Profiler. Measure before optimizing.
- Git records history. Ignore `.godot/` (and generated build folders), but commit `.import` files, `.uid` files and `export_presets.cfg` after checking it for secrets. Use branches for experiments.
- Use lowercase `snake_case` file names, `PascalCase` nodes and classes, a folder layout by feature, small standalone scenes, "call down, signal up", and data in resources.
- Scope small: one-sentence pitch, core loop, Must/Should/Could lists, a vertical slice, and quiet playtests with new players.
- Export needs export templates, a Windows Desktop preset, and an honest test on a clean folder; Embed PCK gives one file.
- Keep a `CREDITS.md`, follow each license, and use an AI assistant to explain and check rather than to replace your own attempts.

Review questions

1. Why is it better to fix one thing at a time while debugging?
2. In a stack trace, which entry shows where the error happened, and which entries show how the program got there?
3. What is the difference between a red error and a yellow warning in the Output panel?
4. Name two tabs of the Debugger panel and say what each shows.
5. Why must you never put real work inside an `assert(...)` call?
6. Which folder must be in `.gitignore` for a Godot 4 project, and why?
7. True or false: `.import` files should be ignored by git.
8. What does the "call down, signal up" rule mean?
9. What is a vertical slice, and why build one early?
10. You exported your game but it closes at once when double-clicked. What is the first thing you do to see why?

Exercises

1. **Practice (easy):** Write a `.gitignore` for a Godot 4 project from memory, then compare it with the one in this chapter.
2. **Practice (easy):** Add a `Health` component (Section 11.3) to the Coin Dash player, with `maximum = 3`, and print a message whenever `changed` is emitted.
3. **Apply (medium):** Put a breakpoint in `GameState.load_game()`. Press L in the game, and step line by line until the player's position is restored. Write down the value of `nodes_data` as shown in the Variables list of the Stack Trace tab.
4. **Apply (medium):** Write a one-page game design for a new small game: one-sentence pitch, core loop, Must/Should/Could lists, and a vertical slice description. Show it to one person and note their first question.
5. **Challenge (hard):** Run a playtest of your Coin Dash `.exe` with three people. Record what each does in the first 30 seconds, make three changes based on what you saw, export again, and compare. Write a short changelog.

Common mistakes

- **Pushing the `.godot/` folder to git.** -> No `.gitignore`, or it was added after the first commit. -> Add the ignore rule, then run `git rm -r --cached .godot` and commit.
- **The game works in the editor but the `.exe` fails to start or misses files.** -> Non-resource files (such as `.json`, `.txt`) were not included, or case-sensitive paths differ. -> Add filters in the export Resources tab and use lowercase file names.
- **Breakpoint never stops.** -> The line is never reached, or the game was not started from the editor. -> Add a `print` before the line to check, and run with F5/F6.
- **"Fixed" a bug by hiding the error.** -> Added a null check without asking why it was null. -> Find out who should have set the value, and fix there too.
- **Export button says templates are missing.** -> Export templates for this exact Godot version are not installed. -> Editor > Manage Export Templates > Download and Install.
- **Files moved in Windows Explorer and scenes show missing resources.** -> Godot did not know about the move. -> Move files only inside the FileSystem dock, or fix paths and commit.

Further reading

- Debugger panel: https://docs.godotengine.org/en/stable/tutorials/scripting/debug/debugger_panel.html
- Overview of debugging tools: https://docs.godotengine.org/en/stable/tutorials/scripting/debug/overview_of_debugging_tools.html
- GDScript style guide: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_styleguide.html
- Version control systems: https://docs.godotengine.org/en/stable/tutorials/best_practices/version_control_systems.html
- Project organization: https://docs.godotengine.org/en/stable/tutorials/best_practices/project_organization.html
- Scene organization: https://docs.godotengine.org/en/stable/tutorials/best_practices/scene_organization.html
- Exporting projects: https://docs.godotengine.org/en/stable/tutorials/export/exporting_projects.html
- Exporting for Windows: https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_windows.html
- Pro Git book (free): https://git-scm.com/book
- Godot license information: https://docs.godotengine.org/en/stable/about/complying_with_licenses.html

Part III: Point-and-Click Adventures with Popochiu

Chapter 12: Adventure Game Design

Learning objectives

By the end of this chapter you will be able to:

- Describe the history of the point-and-click adventure genre and name the main design traditions that shaped it.
- Compare a verb-based interface with a one-click interface and choose one for a project.
- Draw a puzzle dependency graph and use it to find dead ends, bottlenecks and unfair puzzles.
- Apply design rules for fairness, hints, red herrings and low-cost failure.
- Control the scope of an adventure game so that it can actually be finished.
- Write a complete paper design for a small adventure, including a room map, item list, dialogue sketch and design document.

Key terms

- **Point-and-click adventure**: a game in which the player explores places, collects objects and talks to characters mostly by clicking on the screen.
- **Parser**: an early interface in which the player typed sentences such as "open door" and the game tried to understand them.
- **Verb interface**: an interface with a list of actions (Open, Pick up, Use, and so on) that the player combines with an object.
- **One-click interface**: an interface in which a single click on an object triggers the most sensible action for it.
- **Puzzle**: an obstacle that the player must remove by finding, combining or using something.
- **Lock and key**: the basic puzzle pattern; the player is blocked (the lock) until they find the thing that opens it (the key).
- **Dependency graph**: a diagram of which puzzle or item must be completed before another one can be.
- **Red herring**: an object or clue that looks useful but is not part of any solution.
- **Dead end**: a state of the game from which it can no longer be finished.
- **Pixel hunt**: the frustrating chore of sweeping the mouse over the screen to find the one object that can be clicked.
- **Scope**: the total amount of content (rooms, items, puzzles, dialogue) a project contains.
- **Design document**: a written plan that describes what the game contains and how it fits together.

12.1 A short history of the genre

You are about to build a point-and-click adventure, so it helps to know where the form comes from. Every design rule in this chapter was learned the hard way by someone, usually by a player getting stuck.

Text adventures and the first graphics

The first adventure games had no pictures. In the 1970s players typed commands such as `go north` or `take lamp` and the computer printed a description. The program that read these sentences is called a **parser**. Players loved the freedom, but they also fought the parser: the game might understand `take lamp` and not

`pick up the lamp`, and guessing the right words became part of the puzzle.

In the early 1980s the company Sierra On-Line added pictures to the formula. Its *King's Quest* series (the first game appeared in 1984) let you see your character walk around a drawn screen, but you still typed commands. Later Sierra games switched to a mouse-driven interface with icons.

Sierra and Lucasfilm Games

Two studios set the two main traditions of the genre.

Sierra made games in which the world was dangerous. You could fall off a cliff, be eaten, or make a mistake that quietly ruined the game hours later. Dying was normal and the player was expected to save often. The games were rich in jokes and in death messages.

Lucasfilm Games (later called LucasArts) took the opposite approach. Its adventure *Maniac Mansion* (1987) was built with a system whose name, SCUMM, stands for Script Creation Utility for Maniac Mansion. The player picked a verb from a panel at the bottom of the screen and then clicked an object. The later LucasArts adventures, above all *The Secret of Monkey Island* (1990), became famous for comedy and for a design philosophy associated with Ron Gilbert, one of the creators of both games: do not let the player die by accident and do not leave the player in a game that can no longer be won. Gilbert wrote about adventure design in a 1989 essay, "Why Adventure Games Suck"; the two rules above are the way his approach is usually summarized, so read the essay itself if you want his exact words.

You will see both traditions in this book. "Weird Hotel" follows the LucasArts side: it is safe to experiment, and failing a challenge costs the player almost nothing.

First-person worlds and the long quiet

In 1993 *Myst* (made by Cyan) showed a different style. You moved through a series of still, beautifully rendered first-person scenes, and the puzzles were mechanical machines rather than inventory riddles. Many "Myst-likes" followed. Meanwhile, by the end of the 1990s, the big studios were turning to other genres, and the classic adventure went quiet in the mainstream.

The return

Small teams kept the form alive, and then brought it back:

- Independent designers made adventures with free tools. The most important was **Adventure Game Studio (AGS)**, a free Windows program created by Chris Jones in 1997, which let amateurs make complete adventures.
- *Machinarium* (2009) told a story with no spoken words, using pictures and wordless bubbles.
- *Broken Age* (2014) was financed by crowdfunding, which showed that there was an audience.
- *Thimbleweed Park* (2017), by Ron Gilbert and Gary Winnick, deliberately revived the verb interface of the old days.
- *Return to Monkey Island* (2022) continued a series that began in 1990.

Today an adventure can be made by one person with a general-purpose engine. In Unity there is an asset called **Adventure Creator**. In Godot there is **Popochiu**, the plugin you will use from the next chapter. Popochiu's own README says it is inspired by AGS and by PowerQuest (a Unity tool). Like them, it organizes a game into rooms with characters, props and hotspots, and it comes with an inventory and a dialogue system, all inside Godot, so you also keep access to everything in Chapters 1 to 11.

> **Note:** Dates and names in a history like this one are easy to get slightly wrong. If you quote them in a report, check each one against a second source.
>
> **Try it:** Pick one game named in this section and watch ten minutes of someone playing it online. Write down (1) how the player chooses an action, (2) what happens when the player gets stuck, and (3) whether the player can die. Keep these notes; you will use them in Section 12.2.

12.2 Verbs or one click?

The **interface** decides how many different things the player can try. It is the first big design choice, because it controls how many ways each object has to react.

The verb interface

In a verb interface (SCUMM style) the screen shows a list of words such as *Open, Close, Pick up, Look at, Talk to, Use, Push, Pull, Give*. The player clicks a verb and then an object. A sentence line shows what is being built: "Open drawer".

Good points:

- Many different actions on the same object, so many jokes ("Pull the drawer", "Talk to the drawer").
- The player always knows what the verbs are.

Weak points:

- Each object needs a response for every verb, or a default. Nine verbs on twenty objects is 180 possible responses.
- Most of the verbs are pointless on most objects, so players click twice for every real action.
- It plays badly on touch screens.

The one-click interface

In a one-click interface the player clicks on an object and the game picks the action. A common arrangement is **left click: do the natural thing**, and **right click: look at it**. Selecting an inventory item and then clicking an object means "use this on that".

Good points:

- Fast and easy to learn.
- Much less writing and scripting: two responses per object (click and look), plus item combinations.
- Works on a touch screen.

Weak points:

- Fewer joke possibilities per object.
- One click must serve many purposes, so the designer must decide what "the natural thing" is.

Which one for us?

"Weird Hotel" uses the **Simple Click** style that Popochiu ships (you will choose it in Chapter 13): left click interacts, right click looks. Popochiu also ships a Sierra-inspired template (named "Action Bar" in the Setup popup) and a 9-verb template, so the choice can be revisited later. We choose one click for three reasons: it halves the writing, it makes labs shorter, and it keeps the focus on puzzle design.

> **Why it works:** A good design decision is one that you can defend by what it saves. The one-click interface does not make a better game than the verb interface in general. It makes a smaller game that a beginner can finish.
>
> **Try it:** Take your notes from Section 12.1. Which interface did the game use? List two objects you could click on and say what each interface would let you do to them.

12.3 The puzzle: a lock and a key

In the original course for this project, one sentence stood out: **every adventure puzzle is just a lock and a key, often in disguise.** The player is blocked by something (the lock). The way through is to find something (the key) and use it. The world changes.

A lock can be a closed door, but it can also be a guard who wants a bribe, a machine with a missing part, a character who needs convincing, a broken bridge, or a number the player must find. A key can be an object, a piece of information, a sequence of actions, or a person's trust.

Chains

One lock and key is a puzzle. A game is a chain. The key to the first lock sits behind a second lock, and so on:

```
find key  ->  open door  ->  enter hallway  ->  find bone  ->  ...
```

When designing, ask three questions of every puzzle:

1. **What blocks the player?** (the lock)
2. **What opens it?** (the key)
3. **Where does the player learn about it?** (the hint)

If you cannot answer the third question, the puzzle is unfair.

Four kinds of lock

Kind	The player has to	Example in Weird Hotel
Object lock	Use an object on a hotspot	Use the Key on the Door
Combination	Merge two items	Bone + Twine = BoneFlute
Social lock	Talk to a character and say the right thing	Persuade the Frog
Skill lock	Win a small action game	Catch the Bug in the Jar; dodge the boss

A balanced game mixes kinds. If every puzzle is an object lock, the game feels repetitive. The skill locks in this book also give the player a rest from reading.

12.4 Dependency graphs

A **dependency graph** shows what must happen before what else. You draw a box for each important event (get an item, open a place, finish a talk) and an arrow from an event to anything that needs it.

Here is a small example, drawn in text:

```
[Open drawer] --> [Key] --> [Open door] --> [Hallway]
```

Reading the graph tells you facts that the prose version hides.

What to look for

- **Parallel paths.** Two boxes with no arrow between them can be done in either order. This gives the player a choice, which feels like freedom. A game that is only one long chain feels like a corridor.
- **Bottlenecks.** A box that everything else depends on. The first door in "Weird Hotel" is a bottleneck: nothing beyond it can happen until it opens. Make bottlenecks easy.
- **Dead ends.** Follow every arrow. Is there any way for the player to lose something that is needed later? A key that is used up and also needed again is a dead end.
- **Orphans.** A box with no arrow leading anywhere. That is either a red herring (fine, if on purpose) or a mistake.
- **Long chains.** Every arrow is something the player must work out. Ten boxes in a row is a long hunt for one hint.

The complete graph of Weird Hotel

Here is the whole game in one picture. Read from left to right.

```
BEDROOM                        HALLWAY
-------                        -------

[Search Drawer]
      |
      v
   (Key) --use on--> [Door] --> enter Hallway
                                    |
        +---------------+-----------+-----------+------------------+
        |               |                       |                  |
        v               v                       v                  v
  [click BonePile] [click TwineBall]      [click Jar]      [click Ceiling]
        |               |                       |                  |
     (Bone)          (Twine)                    v                  v
        |               |            [mini-game: catch bug]  [boss fight]
        +--> combine <--+                       |              |       |
                |                               v             win     lose
                v                             (Bug)            |       |
          (BoneFlute)                           |              v       v
                |                               v          [ending]  [wake up in the
                v                     [use Bug on Frog]                Bedroom at the Bed
   [talk to Frog: "Flute" option]     (optional: the Frog              marker, items kept,
                |                      comments = a hint)              try again]
                v
   (spare Key: the landlord's thanks;
    a keepsake, nothing needs it)
```

Observe:

- One bottleneck (the Door) at the start.
- Four branches in the Hallway that can be done in any order: the BonePile and TwineBall (which lead to the flute), the Jar, and the Ceiling.
- The Bug is optional. It is a branch that leads only to a hint (the Frog comments on it), which means it is a reward for curious players, not a requirement.
- The boss fight is reachable from the moment the player enters the Hallway. The game never locks it away.
- The Frog's reward is a spare Key. Nothing in the graph needs it: it is a deliberate loose end, explained in Section 12.10.

Making a graph for your own game

1. List every item and every place the player can get stuck. Write each as a box.
2. For each box, write what it needs. Draw arrows from those needs.
3. Look for the five things above.
4. Walk through the graph as the player and as a mischievous player who does things in the worst order.

> **Try it:** Draw the dependency graph of a simple everyday task, such as "make toast for a friend". Include at least one parallel path and one possible dead end (for example, the bread was eaten before it was toasted). Mark the dead end with an X.

12.5 Fair play: how to design puzzles that are not insulting

A good puzzle is hard to solve and easy to understand once solved. The player should think "of course", not "how was I supposed to know that?" The following rules help.

Rule 1: Show the lock before the key

If the player finds a key before ever seeing a lock, they carry it around confused. If they see the lock first, the player already has a *goal*, and the key becomes a delightful solution. Weird Hotel does this: the Door is visible and locked from the first minute, so when the player finds a Key in the drawer they know where to use it.

Rule 2: A hint at every step

Every puzzle needs at least one hint in the world. A hint can be a line of text (right-click on the object), a character's comment, or something visible: a frayed bit of twine on a lamp, a bone on the floor. Design the hint at the same time as the puzzle.

Rule 3: Logic first, jokes second

Adventure puzzles often have absurd solutions, and that is part of the charm. But the absurdity should have a *rule behind it*. In Weird Hotel, a frog landlord who loves music may well trade a key for a bone flute. That makes sense inside the world. A frog who gives a key if you click the lamp three times does not.

Rule 4: Few objects, many uses

A hotel with 40 items is a nightmare for the player: every combination of two items is a possible puzzle, 40 x 39 / 2 = 780 pairs. With 5 items there are only 10 pairs. A small number of items keeps the guessing sensible.

Rule 5: Give a response for everything

When the player tries something silly, such as using the Key on the Jar, the game should answer with a friendly, funny line, not with silence. A default "That won't work" is acceptable. A joke is better. Silence makes the player think the game is broken.

Rule 6: Red herrings are seasoning, not the meal

A **red herring** is an object that has no use. A few are fun, because they make the world feel real. But if half of the objects are red herrings, the player stops trusting any of them. A useful guideline: no more than one red herring in five interactive objects. Never make a red herring *look* important.

Rule 7: No pixel hunts

A pixel hunt happens when a tiny object is hard to see and the player must wave the mouse over the whole screen to find it. Make clickable things big and visible, and avoid "hidden" ones unless they are the entire point of a puzzle. Popochiu helps a little: the Simple Click interface shows the name of the object under the pointer, so the player can sweep the mouse and learn what is clickable. That is still a chore, so do not rely on it.

Rule 8: Never make the game unwinnable

This is the most important rule. A game is unwinnable when the player has done something that can never be undone and the game cannot be finished. Examples:

- A key is used up, but is also needed to open a second door.
- A character leaves and cannot be talked to again.
- An item is thrown away.

Tell each such problem apart from a **setback**. A setback loses a bit of time. A dead end loses the game. Check the graph for dead ends, and fix them in one of three ways: do not allow the action, make the item reappear, or make the item not consumable.

12.6 Failure should cost little

Many players enjoy challenge, but few enjoy repeating ten minutes of play. Sierra's tradition of death at every corner produced many save-and-restore cycles. LucasArts' principle was that the player explores and experiments without fear, because nothing is irreversible.

In Weird Hotel, the **boss fight** is the only place where the player can fail. The rules for failure are:

- You lose by being hit three times.
- Losing is not the end of the game. You wake up in the Bedroom.
- You **keep everything** you collected. Nothing is lost except time.
- You can try again immediately.

This keeps the sense of danger while removing the punishment. A friendly game is not necessarily an easy game. It is one in which learning is cheap.

A useful test: after any failure, ask "how many seconds of the player's life did that cost?" If the answer is more than about thirty, shorten it.

12.7 Game flow: pacing a small adventure

Game flow is the shape of the player's experience over time. Even a half-hour game should have a shape.

A simple, reliable shape for a small adventure:

1. **Opening (2 to 3 minutes).** A safe starting room. The player learns the controls by doing something easy. In Weird Hotel: click the drawer, get the key.
2. **First gate.** A lock that is visible from the start and easy to open. The player feels progress. In Weird Hotel: the Door.
3. **Open middle.** A new area with several tasks that can be done in any order (see the parallel branches in the graph). This is where the player explores.
4. **Change of pace.** A skill challenge. In Weird Hotel: catching the Bug.
5. **Climax.** The hardest challenge, using what the player has gathered. In Weird Hotel: the boss.
6. **Ending.** A short, rewarding payoff.

Vary the rhythm: a reading-heavy dialogue should be followed by something to *do*, and an action section by something calm. Players who have been staring at dialogue boxes for three minutes become impatient.

Where does the player's attention go?

Plan where the player's eye goes in each room. A room with ten equally loud objects overwhelms. Put the important object where the player will look first, and let it be visibly different (bigger, brighter, or doing something).

12.8 Scope: the thing that kills most projects

Scope is how much stuff your game contains. Beginner projects fail because the scope is too big, not because the code is hard. A "short" adventure of ten rooms with twenty characters is a year of work.

Rules of thumb for a first project

Item	Limit for a first game
Rooms	2 to 4
Characters with dialogue	1 to 2
Inventory items	4 to 8
Puzzles	3 to 6
Mini-games	1 or 2
Playing time	10 to 20 minutes

Weird Hotel has **2 rooms, 1 talking character (the Frog), 5 items (Key, Bone, Twine, BoneFlute, Bug), 5 puzzles, 2 mini-games** (the jar, the boss) and roughly fifteen minutes of play. That is deliberate.

The cost of a thing

Before you add any new item, ask what it costs:

- Art for the item (or a placeholder).
- A script for its click and look responses.
- A reaction to every other item (or one default answer).
- A hint in the world.
- Testing.

A new room adds a background, walkable area, entrance marker, return path, and everything inside it. A new character adds a sprite, animations, dialogue, and writing.

The first rule of scope

Make it with boxes first. Draw a rectangle for the background, a coloured square for the player, text for items. If the game is fun with boxes, it will be fun with art. If it is not, art will not rescue it. This is the single rule of the original course, and this book keeps it.

Cutting

Keep a list called "Later". Whenever you have a great new idea, it goes on that list. You are not forgetting it; you are placing it after the first release. A game released with half the ideas is worth more than a perfect game never finished.

12.9 Writing for adventure games

Writing is part of the game design. Most of the player's experience is text.

The voice

Pick a voice for the protagonist and keep to it. The original course gives Weird Hotel a deadpan voice. Examples from its scripts: "A drawer that hums faintly." "Locked. The keyhole is breathing." "I made a flute out of a bone. Normal." The voice is calm and surprised by nothing, and that is the joke.

Short lines

Players read quickly. Keep each spoken line under about 15 words, and break long speeches into several lines. Never put the solution to a puzzle in a long paragraph.

Every object gets a look line

The right-click response is the cheapest and most valuable writing. It lets you describe the world, deliver a hint and make a joke, and it needs no animation. Write one line for every clickable object, even red herrings.

Hints in the voice

Prefer hints that are in the character's voice and not a game tip. Instead of "You need the bone and the twine", write "A bone. Something stringy would help." The first is a manual. The second is a clue.

Repetition

Players click on the same thing many times. If an object has an action that happens once (taking the key), the second click needs a different response ("Just old socks now."). Writing the second response is easy to forget and is a common bug source. We will see exactly why in Chapter 14.

Dialogue trees

A **dialogue tree** is a conversation in which the player picks from a menu of lines. A good dialogue menu:

- Has 3 to 5 options on screen at once.
- Has an exit option that is always available ("Bye").
- Uses menu text that is a short version of what the character will say, so the player knows what to expect.
- Turns options off or on as the story advances, so the menu reflects what the player has learned.

Here is the sketch for the Frog:

```
FROG DIALOGUE: ChatWithFrog

Start options visible:
  [Who]    "Who are you?"                (shown at the start)
  [Flute]  "Do you need a flute?"        (hidden until Who has been asked)
  [Bye]    "Goodbye."                    (always shown)

Who   -> Frog: "I am the landlord."
         Frog: "I used to have a band. Now I have rent."
         (Who switches off; Flute switches on; menu returns)

Flute -> If the player has no BoneFlute:
            Frog: "No flute. Sad."
            (menu returns)
         If the player has the BoneFlute:
            Frog: "A bone flute. Hm. Take my spare key."
            (BoneFlute is used up; the player receives a Key)
            (Flute switches off for good)
            (conversation ends)

Bye   -> Frog: "Rent is due. Always."
         (conversation ends)

Using the Bug on the Frog (not part of the menu; it is an item reaction):
         Frog: "A bug. Nice. But I want music, not snacks."
         (the Bug is kept; this is only a hint that points to the flute)
```

Notice what the sketch does: every branch ends with a clear statement (menu returns, or conversation ends). Writers often forget to say what happens *after* a line, and the programmer then has to guess. The sketch also shows which options start hidden. Chapter 16 turns this sketch into a working dialog with exactly these option names (`Who`, `Flute`, `Bye`).

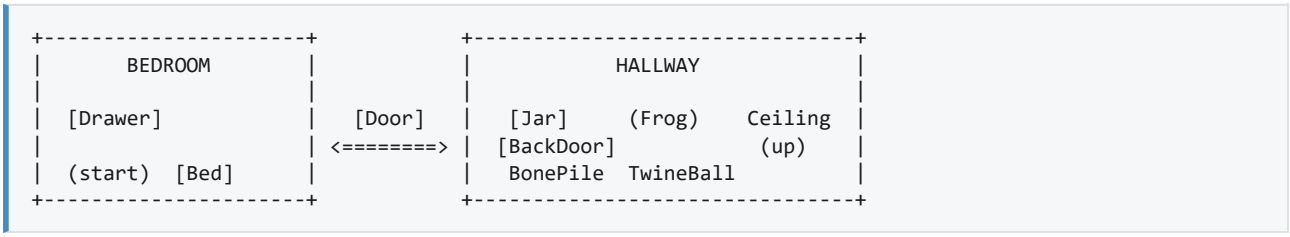
12.10 Paper design of "Weird Hotel"

You now have what you need to design the whole game on paper. This section presents the complete design of the book's game. In the lab you will write it out yourself in your own words, and you may change anything.

Concept

> **Weird Hotel.** You wake up in a bedroom in a hotel where everything is a little wrong. The landlord is a frog. You need to leave, but the door is locked. Along the way you will build a flute out of a bone, catch a bug in a jar, and dodge whatever lives in the ceiling.

Room map



Two rooms joined by one doorway. The Door is a hotspot in the Bedroom; it starts locked. Once it has been opened, the player can move between the rooms at will: the Hallway has its own hotspot, the BackDoor, which leads back to the Bedroom. `Bed` is a marker (a named position, Chapter 14) in the Bedroom where the player wakes up after losing the boss fight.

Item list

Item	Found where	Used for	Consumed?
Key	In the Drawer (Bedroom)	Opens the Door	Yes (the Door takes it)
Bone	Hallway, from the BonePile prop	Combine with Twine	Yes (becomes BoneFlute)
Twine	Hallway, from the TwineBall prop	Combine with Bone	Yes (becomes BoneFlute)
BoneFlute	Made from Bone + Twine	Gift for the Frog	Yes (trade)
Bug	Reward for winning the Jar mini-game	Show it to the Frog; he comments (a hint)	No

The prop is where a thing lies on the floor (`BonePile`, `TwineBall`); the item is what the player carries (`Bone`, `Twine`). Using different names for the two avoids confusing them in code.

Notice that the Key appears twice in the story. The first is consumed by the Door. The second is the Frog's reward, and it is a **spare**: the same inventory item `Key`, handed over after the first has long been used up. Section 12.10 explains why the spare is allowed to open nothing.

Characters

Character	Where	Role
Player	Everywhere	The protagonist. Deadpan.
Frog	Hallway	The landlord. Trades a spare Key for the BoneFlute; comments on the Bug.

Hotspots and props

Object	Type	Room	Left click	Right click
Drawer	Prop	Bedroom	Search it; first time gives the Key	"A drawer that hums faintly."
Door	Hotspot	Bedroom	Locked: a line. Unlocked: go to Hallway	"A door. It looks shy." (placeholder text)
BackDoor	Hotspot	Hallway	Go back to the Bedroom	"The door I came from. It waved."
BonePile	Prop	Hallway	Take the Bone (once)	A look line that hints at the Twine
TwineBall	Prop	Hallway	Take the Twine (once)	A look line that hints at the Bone
Jar	Prop	Hallway	Start the catch-the-bug mini-game	"Something in the jar wants out."
Ceiling	Hotspot	Hallway	Start the boss fight	"The ceiling is looking at me."

The puzzles, listed

#	Puzzle	Type	Needs	Result
1	Get through the Door	Object lock	Key from Drawer	Hallway opens
2	Make the BoneFlute	Combination	Bone, Twine	BoneFlute
3	Catch the bug	Skill lock	Click the Jar, win	Bug (optional; a hint from the Frog)
4	Persuade the Frog	Social lock	BoneFlute, dialogue	A spare Key
5	Beat the ceiling	Skill lock	Dodge 20 seconds	Ending

What the game must remember

Every "once only" event in the table above needs a stored fact, or the game forgets it when a room is rebuilt (you will see this bug happen in Chapter 14). These are the names the later chapters use. They live in the `Globals` script (Chapter 15 onward); you do not need them yet, but the design should list them.

Name	Meaning	Chapter
`key_taken`	The Key has been taken from the Drawer	15
`door_unlocked`	The Door has been opened with the Key	15
`bone_taken`, `twine_taken`	The Bone and the Twine have been picked up	15
`hallway_intro_seen`	The arrival scene in the Hallway has played	16
`frog_met`	The player has spoken to the Frog before	16
`frog_who_asked`	The "Who are you?" option has been used	16
`flute_given`	The Frog has traded the spare Key for the flute	16
`boss_defeated`	The ceiling boss has been beaten	17
`just_died`	The player has just lost and is about to wake up in the Bedroom	17
`deaths`	How many times the player has lost	17 and 18
`click_counts`	How often each object was clicked (for varied lines)	18

Fairness check against Section 12.5

- *Lock before key:* the locked Door is the first thing visible.
- *Hints:* the Drawer hums (invitation); the BonePile and TwineBall have look lines that hint at each other; the Frog says "No flute. Sad." which tells you what to bring; the Frog's comment on the Bug points the same way.
- *Dead ends:* the consumable things are the Key (used on the Door), the Bone and Twine (used up making the flute) and the flute (traded). Each is needed exactly once, and the game never asks for any of them a second time. The spare Key is never required, so the player cannot be stuck lacking it, and the Door keeps a Key that is used on it after it is already open, so holding one does no harm. No dead end.
- *Failure:* only in the boss fight, cheap, items kept.
- *Red herrings:* the Bug and the spare Key are not needed to finish the game. Neither is a true red herring, because each has a purpose: the Bug earns a hint, and the Key is a reward. Still, they are optional things, so keep Rule 6 in mind if you add more.

What does the spare Key open?

A design gap, and a real one, so we state the decision rather than leave it vague. The Frog gives the player a spare Key, and in the version of the game you build **nothing opens with it**. We choose this on purpose:

- It is a reward and a backup: the landlord's thanks, proof that the trade worked. It does not block anything and nothing depends on it.
- It cannot cause a dead end, because the first Key was used up long before the Frog is reached, and the Door keeps any extra Key used on an already open door.
- A full game would use it for a front door and the story's ending. For this book the story ends when the boss is beaten, so a front door would be extra scope (Section 12.8).

This is an honest example of scope control: we noticed an unfinished thread, wrote down why it is safe, and put the finishing touch on the "Later" list. Exercise 3 below asks you to give it a real lock on paper, a Lobby with a front door, and Chapter 16 suggests the Jar as another possible lock.

Sketch of the first minute

```
Player wakes up in the Bedroom.
Player clicks the Door -> "Locked. The keyhole is breathing."
Player clicks the Drawer -> walks over -> "A key! It is warm. Why is it warm?"
                          -> Key appears in the inventory.
Player selects the Key in the inventory bar, then clicks the Door -> "It fits!"
                          -> Key is used up.
Player clicks the Door -> fade -> Hallway.
```

The design-document template

Copy the following to a file called `weird-hotel-design.md` and fill it in for the lab. Be brief. One line per field is enough.

```
# Design document: <GAME TITLE>

## 1. Concept
One sentence: who is the player, what do they want, what is in the way?

## 2. Interface
Verb or one click? Which buttons do what?
```

```
## 3. Rooms (map)
(ASCII map of the rooms and how they connect)

## 4. Characters
| Name | Room | Role | Voice (3 adjectives) |

## 5. Items
| Name | Found | Used on | Consumed? |

## 6. Objects (props and hotspots)
| Name | Room | Left click | Right click | Item use |

## 7. Puzzles
| # | Lock | Key | Hint (where does the player learn this?) |

## 8. Dependency graph
(ASCII diagram)

## 9. Dialogue
(Sketch for each conversation: options, results, what turns on/off)

## 10. Failure and recovery
Where can the player fail? What does it cost?

## 11. Scope check
Rooms: __ Characters: __ Items: __ Puzzles: __ Estimated play time: __

## 12. Later (ideas not in the first version)
-
```

Lab 12: Write the Weird Hotel design document

Time: 45 to 75 minutes. **Tools:** a text editor (Notepad, VS Code, or any Markdown editor) or a sheet of paper. No Godot.

Steps

1. Create a folder for the project somewhere you can find it, such as `Documents/weird-hotel-notes`. Create a file `weird-hotel-design.md` and paste the template from Section 12.10.
2. Fill in sections 1 and 2. Write the concept in your own words. State that the interface is one-click (left click acts, right click looks).
3. Draw the room map (section 3) with the two rooms and the Door. Mark where the player starts.
4. Fill in the characters table (section 4). Write three adjectives for the Player's voice and for the Frog's.
5. Fill in the items table (section 5) with the five items.
6. Fill in the objects table (section 6) with the seven objects (Drawer, Door, BackDoor, BonePile, TwineBall, Jar, Ceiling), giving both a left-click and a right-click response in your own voice. Write the *second-click* response for the Drawer.
7. Write the puzzles table (section 7). For each puzzle, say where the player learns what to do.
8. Draw the dependency graph (section 8). Do not copy the picture in Section 12.4; draw it from your own tables, and then compare.
9. Write the Frog's dialogue sketch (section 9), including what happens after each line.
10. Fill in sections 10 and 11. Under section 8 add a short list of the facts the game must remember (see "What the game must remember" in Section 12.10), in your own words.
11. Perform a dead-end test: walk through your graph and try to do things in the worst order. Write down in section 12 anything you find, and one change you would make to the design.
12. Add at least two ideas of your own to the "Later" list.

Expected result

- One Markdown file with all twelve sections completed.
- A dependency graph that matches your tables.
- A written answer to "Can the player lose the game by doing things in the wrong order?" (The correct answer for the design above is no.)
- A clear list of all the exact names you will use in Godot: rooms `Bedroom` and `Hallway`; characters `Player` and `Frog`; items `Key`, `Bone`, `Twine`, `BoneFlute`, `Bug`; props `Drawer`, `BonePile`, `TwineBall` and `Jar`; hotspots `Door`, `BackDoor` and `Ceiling`; markers `FromHallway`, `FromBedroom` and `Bed`; dialog `ChatWithFrog`. Put this list at the top of your file. You will use the names exactly as written.

You can change the design. If you do, be aware that later chapters in the book assume the version in this chapter. Changes are fine, but write the differences down.

Chapter summary

- Adventure games began as text parsers, moved to Sierra's dangerous worlds and LucasArts' safe, funny ones, and returned through free tools and crowdfunding. Popochiu continues the line inside Godot.
- A verb interface offers many actions per object; a one-click interface offers few, and is much cheaper to write. We use one click.
- A puzzle is a lock and a key, and a game is a chain of them. Always ask what blocks, what opens, and where the player learns it.
- A dependency graph reveals bottlenecks, dead ends, orphans and unfair chains before a single line of code is written.
- Fair design means: lock before key, a hint at every step, a rule behind every absurd solution, few items, a response for everything, few red herrings, no pixel hunts, and never an unwinnable game.
- Failure should cost seconds, not minutes. In Weird Hotel, death is a setback with no loss of items.
- Scope is the main risk. Keep to two to four rooms, make everything with boxes first, and keep a "Later" list.
- Write short lines, give every object a look line, give a different response on the second click, and say what happens after every dialogue line.

Review questions

1. What is a parser, and why did players find it frustrating?
2. Name the two main design traditions of 1980s adventure games and give one difference between them.
3. In a one-click interface, what normally happens on left click and on right click?
4. Give two advantages of a one-click interface over a verb interface for a beginner project.
5. In "lock and key" terms, what are the lock and the key in the first puzzle of Weird Hotel?
6. True or false: two boxes in a dependency graph with no arrow between them can be done in either order.
7. Define a "dead end" and give an example from an adventure game.
8. What is the difference between a setback and a dead end?
9. Why is "show the lock before the key" a good rule?
10. In Weird Hotel, what does the player lose when they are defeated by the boss?

Exercises

1. **Practice.** Draw a dependency graph for this mini-adventure: a locked shed; the shed key is under a flowerpot; inside the shed is a ladder; the ladder reaches a cat stuck in a tree; rescuing the cat earns a reward. Mark the bottleneck.
2. **Practice.** For each of these puzzle descriptions, say whether it is fair, and if not, add the missing hint: (a) the player must click a plain wall to find a secret door; (b) the player must give a sandwich to a hungry guard who says "I'm starving".
3. **Apply.** Add one new room and one new item to the Weird Hotel design (for example a Lobby with a front door that the Frog's spare Key opens). Update the room map, items table, puzzle table and dependency graph. Check for dead ends. Write what the new content costs (Section 12.8).
4. **Apply.** The sketch in Section 12.9 has one line for the Frog when the player uses the Bug on him. Write a fuller version as a new menu option, "Look what I caught", that appears only after the player has the Bug. Decide what the Frog says, whether the option appears only once, and what it turns on or off. Use the sketch format from Section 12.9.
5. **Challenge.** Design a complete, original three-room adventure on paper using the twelve-section template. It must contain at least one example of each of the four lock kinds from Section 12.3 and a dependency graph with two parallel paths. Trade with another reader and try to find a dead end in their design.

Common mistakes

- **Symptom:** The player wanders for ten minutes with nothing to do. **Cause:** a puzzle has no hint, or the hint is in a place the player has not reached. **Fix:** check each puzzle for where the player learns what to do; place the hint before the lock.
- **Symptom:** A tester gets stuck after using an item the wrong way. **Cause:** a dead end in the design. **Fix:** find it in the dependency graph; make the item non-consumable, replaceable, or the action impossible.
- **Symptom:** Every object has a funny reaction except the ones the tester tries. **Cause:** no default response. **Fix:** write a default line for item-on-object attempts.
- **Symptom:** The design document grows and the game is never started. **Cause:** scope creep. **Fix:** move ideas to "Later" and set a fixed number of rooms.
- **Symptom:** The player clicks the same object again and gets the same line, even though the object has changed. **Cause:** no second-click response was written. **Fix:** write what the object says after the event, and keep the dependency in the design.
- **Symptom:** Players dislike the boss fight. **Cause:** failing costs too much. **Fix:** respawn immediately, keep items, shorten the retry.

Further reading

- The Popochiu documentation home page, for the vocabulary of rooms, props and hotspots that Chapters 13 to 18 use: https://carenalgas.github.io/popochiu/
- The Popochiu README on GitHub, which names the tools Popochiu is inspired by: https://github.com/carenalgas/popochiu
- The Adventure Game Studio website, where many free fan-made adventures can be played for inspiration: https://www.adventuregamestudio.co.uk/

- Ron Gilbert's essay "Why Adventure Games Suck" (1989) was first published in a print journal and is not linked here because we could not confirm a stable web address. If you search for it, check the author and the date on the page you find.

Chapter 13: Installing Popochiu and How It Is Built

Learning objectives

By the end of this chapter you will be able to:

- Explain what a Godot plugin (addon) is and how it differs from a project you write yourself.
- Choose the Popochiu version that matches your Godot version, using the version table.
- Install Popochiu from a GitHub release zip and enable it in Project Settings.
- Complete the Setup popup and pick the Simple Click GUI template.
- Name the dock's tabs and the folders Popochiu generates under `res://game/`.
- Describe the ten autoload singletons `R`, `C`, `I`, `D`, `E`, `G`, `A`, `T`, `Cursor` and `Globals`, and say what each is for.
- Explain how every Popochiu object is made of a scene, a script and a state script, and why you create objects through the dock.

Key terms

- **Plugin (addon)**: a folder of scenes, scripts and editor tools that adds features to Godot; it lives in `res://addons/`.
- **Popochiu**: an open-source Godot plugin for making point-and-click adventure games.
- **Release zip**: a packaged download of a specific version of a program, found on a GitHub "Releases" page.
- **Dock**: a panel in the Godot editor. Popochiu adds its own dock.
- **Setup popup**: the window Popochiu shows the first time it is enabled, which configures the project.
- **GUI template**: a ready-made on-screen interface (verb list, inventory bar, cursor) that you choose for the game.
- **Autoload**: a script or scene that Godot loads automatically when the game starts, available by name from everywhere (Chapter 5).
- **Singleton**: an object of which only one exists; autoloads act as singletons.
- **Script name**: the identifier Popochiu uses for an object, such as `Bedroom`. It is used in code, for example `R.goto_room("Bedroom")`.
- **State script**: a small script that stores the variables of one room, character or item so that they can be saved.
- **Commit**: a saved snapshot of your project in git (Chapter 11).

13.1 What a plugin is

In Chapter 2 you learned that a Godot project is a folder of scenes and scripts. A **plugin**, also called an **addon**, is simply another folder placed inside your project, in a folder named `res://addons/`. It can contain anything a project contains (scenes, scripts, images), and it may also contain *editor code*: scripts that run inside the Godot editor itself and add panels, buttons and new node types.

Popochiu does both. At edit time, it adds a dock to the editor and a set of menus and popups, which help you create rooms and characters. At play time, its engine code runs inside your game. It is not a separate program. When you export your game, the Popochiu engine code goes in with it.

This is a good moment to notice what you do *not* have to do:

- You do not write the clicking system. Popochiu knows how to detect that a character was clicked.
- You do not write walking. Popochiu knows how to walk a character along a path and around obstacles.
- You do not write an inventory. Popochiu has items, a bar to show them, and rules for dragging them onto objects.

What you write is the *specific content* of your game: what happens when this drawer is clicked.

Why use a plugin rather than build it ourselves?

You could build all of this yourself with Chapters 6 to 10. It would be a very instructive project, and also a very long one. A plugin lets you reach the interesting part (the puzzles) sooner. It also gives you something to read: the Popochiu code is open source, and reading how professionals structure a system is a good way to learn.

The risk of plugins

Plugins have a cost. They are written by other people and they change. The version you install must match your Godot version. If you update Godot or the plugin, parts of your game may stop working, and the plugin's authors might rename functions. That is why this chapter starts with the version table.

> **Note:** Everything in this book was checked against Godot 4.7.2 and Popochiu 2.2.1. If you use different versions, menu names and a few function names may differ. When something does not match, check the Popochiu documentation (https://carenalgas.github.io/popochiu/) before assuming that you made a mistake.

13.2 Choosing the right versions

Godot and Popochiu release new versions independently. Popochiu's README on GitHub (https://github.com/carenalgas/popochiu) contains a table showing which Popochiu version belongs to which Godot version. At the time of writing it reads:

Godot version	Popochiu version
4.7	2.2.1
4.6	2.1.0
4.3	2.0.3

Godot 3 is not supported by this plugin.

Always check the README for the current pair. By the time you read this, the table may have new rows. The table above was correct when the book was written, but it is only a snapshot.

What to install

- Install the latest **stable** Godot 4 release that appears in the Popochiu table. At the time of writing this is Godot 4.7.2 with Popochiu 2.2.1.
- Do not use a beta of Godot (for example, a 4.8 beta) with Popochiu. The table lists stable versions, and beta versions can break things.
- Use the standard Godot, not the ".NET" or "Mono" version. This book uses GDScript.

Upgrading later

If you ever upgrade Popochiu inside an existing project, follow the release notes. The 2.2.1 release notes say that from 2.1.x you should first go to 2.2.0 (which contains migration steps), and that methods deprecated in 2.1 were removed in 2.2.1. The safe habit is: commit to git, upgrade one step, run the game, commit again.

> **Warning:** Do not upgrade the plugin in the middle of a project unless you have a reason. For this course, install once and keep that version until Chapter 18.
>
> **Try it:** Open the Popochiu GitHub page in a browser. Find the version table. Write down (1) the newest Popochiu version, (2) the Godot version it needs. Compare with the table above and note any difference. Then open Godot, and read the version in the Project Manager's title bar or the editor's Help > About.

13.3 Installing step by step

The process is the same for most Godot plugins, so you are learning something that you can reuse. The official pages are https://carenalgas.github.io/popochiu/getting-started/installing-popochiu/ and Godot's own page on installing plugins (https://docs.godotengine.org/en/stable/tutorials/plugins/editor/installing_plugins.html).

Step 1: Create a project

In the Godot Project Manager click **New Project**. Name it `weird-hotel`. Choose a folder (using **Create Folder** makes a new folder with that name). If the dialog has a "Version Control Metadata" option, choose **Git**; this creates a `.gitignore` file for you. Click **Create & Edit**. The renderer can stay at its default.

Step 2: Download the release

1. Go to https://github.com/carenalgas/popochiu/releases.
2. Find the release for the version you chose, for example `v2.2.1`.
3. Under **Assets**, download the zip named like `popochiu-v2.2.1.zip`. Do not download "Source code (zip)", which is a different file and includes the whole repository.
4. Unzip it into a temporary folder, not inside your project. Windows: right-click the zip > **Extract All**.

Inside, you should find a folder named `addons`. That is what we want.

> **Note:** Popochiu is also distributed on itch.io. This book uses the GitHub release zip because its version number is clear. Whether Popochiu is available in Godot's built-in AssetLib depends on when you read this; check the documentation.

Step 3: Copy the addons folder into the project

1. In Godot's **FileSystem** dock, right-click `res://` and choose **Open in File Manager**. This opens your project folder.
2. Copy the unzipped `addons` folder into it, so that you have `weird-hotel/addons/popochiu/...`.
3. Go back to the Godot window. The FileSystem dock refreshes and shows `addons`.

If your project already had an `addons` folder, merge the two; do not overwrite other plugins.

Step 4: Enable the plugin

1. Open **Project > Project Settings**.
2. Click the **Plugins** tab.
3. Find **Popochiu** and tick **Enable**.

Godot's plugin documentation says that no restart is needed. Popochiu's docs say that if you are asked to reload, use **Project > Reload Current Project**. Do that if prompted.

On success, the **Output** panel at the bottom prints a short greeting from Popochiu ("You're using Popochiu, a plugin for making point n' click games", with a little ASCII drawing), and a new dock appears in the editor. The greeting does not show the version; the version is shown in small text at the bottom of the dock (for example `v2.2.1`). Popochiu's code puts the dock in the **upper-left dock area**, as a tab named **Popochiu** next to the Scene tab (the Import tab shares that area). If you cannot see it, click the tabs at the top of the left-hand panel. You can drag a dock to another place if you prefer. At the bottom of the dock there are two buttons, **Setup** and **Documentation**; remember them.

Step 5: The Setup popup

The first time Popochiu is enabled in a project, it opens a **Setup** popup automatically (it opens whenever setup has not been done yet; the **Setup** button at the bottom of the dock opens it again later). This window configures your project for an adventure game. Take your time here. The wizard has three steps, shown as "Step 1 / 3" and so on, with **NEXT** and **BACK** buttons:

1. **"What kind of game do you want to make?"** Two buttons: **Retro / Pixel Art** and **Modern / Hi-Res**.
2. **"Choose your game resolution"** (a drop-down list; for Modern it offers 4K (2160p), Quad-HD (1440p), Full-HD (1080p), HR-Ready (720p) and Retro HD (1024x768)) and **"Choose preview window size"** (Half (0.5x), Native (1x), Double (2x), Quadruple (4x)). The preview size is the size of the window when you press F5 while developing; it does not change the game's own resolution.
3. **"Choose the UI for your game"**: one button per GUI template, with a description underneath. On this last step **NEXT** disappears, and the button that finishes the job is **Create** at the bottom of the popup (it reads **Update** if you run Setup again later).

There is also a small link at the bottom, **"No thanks, I need custom settings"**, which switches to a single-page *custom* setup (a Game type of Custom, Pixel Art or High Resolution, a Game UI list, and boxes for width and height). We use the wizard.

For Weird Hotel:

Step	Pick	Why
1. Game type	**Modern / Hi-Res**	Plain shapes look fine at higher resolution; placeholder art is easier
2. Resolution	**Full-HD (1080p)** (1920 x 1080)	A common size; the book's placeholder pictures use it
2. Preview window size	**Half (0.5x)** if your screen is small, otherwise **Native (1x)**	A 1920 x 1080 window may be larger than a laptop screen; half size is 960 x 540
3. UI	**Simple Click**	Left-click acts, right-click looks (Chapter 12)

Write down the resolution you chose. Chapter 14 asks you to make pictures of that size. Whatever you pick, read the real numbers afterwards in **Project > Project Settings > Display > Window** (Viewport Width and Viewport Height); Chapter 14 relies on them.

The GUI templates

Popochiu 2.2.1 ships three templates, each in a low-resolution and a high-resolution version. The wizard shows the set that matches your choice in step 1, so with **Modern / Hi-Res** you automatically get the high-resolution versions and need not choose a variant yourself. The names on the buttons are:

Button name	Idea
9 Verb	The classic LucasArts verb panel: Open, Pick Up, Push, Close, Look At, Pull, Give, Talk To, Use
Action Bar	Sierra-inspired: a top bar with action icons that you cycle, an overlay inventory and portrait dialogues
Simple Click	Left click interacts, right click looks; a foldable inventory bar at the top

Choose **Simple Click**.

> **Warning:** The template is copied into `res://game/gui/`. If you re-run Setup and pick another template, Popochiu will overwrite changes you made in that folder, and it warns you first. Choose carefully, and do not customize the GUI until Chapter 18.

When Setup finishes, Popochiu creates the `res://game/` folder, copies the template, sets the project's resolution and stretch settings, and registers the autoloads (Section 13.6).

Step 6: Commit

Now is a very good time to save a snapshot. In a terminal opened in the project folder:

```
git init
git add -A
git commit -m "Install Popochiu 2.2.1 and run Setup"
```

If your project already has a repository, skip `git init`. Commit the `addons` folder: it is part of the project, and anyone who clones the project needs it. The `.godot/` folder (created by Godot to cache imported files) should be in `.gitignore`; check that it is.

13.4 Tour of the dock

The Popochiu dock is your control panel. It has four tabs. (Source: https://carenalgas.github.io/popochiu/the-editor-handbook/popochiu-dock/.)

The Main tab

The Main tab lists everything that exists in the game, in four groups:

Group	What it holds	Create with
Rooms	Places	+ at the right of the group header
Characters	The player and NPCs	+
Inventory items	Things you carry	+
Dialog trees	Conversations	+

Each row has small buttons (hover over one to see its tooltip):

- **Open in Editor** (opens the scene).
- **Open in Script**.
- **Open state** (shows the state resource in the Inspector).
- **Open state Script** (opens the state script).
- Room rows also have **Play scene**, which runs just that room.
- **Options** (the last button on the row): a small menu with actions such as **Remove**.

Special markers:

- A **heart** on a room: that room is the main scene, the one the game starts in. Popochiu stores this in Godot's own Project Setting `application/run/main_scene`. The create-room popup only offers a "make this the main scene" tick box while the project has no main scene yet, so tick it for the very first room. Later you can use the row's **Options** menu and choose **Set as Main scene**.
- A **player icon** on a character: that is the player-controlled character (PC). The create-character popup offers a tick box for it while no PC exists yet. Later, use the row's **Options** menu and choose **Set as Player-controlled Character (PC)**. Popochiu stores the choice in `popochiu_data.cfg`, and at start-up it assigns that character to `C.player`. If no PC is set, it falls back to the first character in its list.
- A **star** on an inventory item: it starts in the player's inventory.

Some rows may appear *dimmed*. A dimmed row means the object exists on disk but is not yet registered in the file `res://game/popochiu_data.cfg`. This can happen after you copy a project from somewhere else. The **Options** menu of such a row offers **Add to Popochiu**.

The Room tab

The Room tab only works when a room scene is open in the editor. It lists the room's contents in groups:

Group	Meaning
Props	Objects you can look at and click, or decoration
Hotspots	Invisible clickable zones
Regions	Zones that react when a character walks into them

Group	Meaning
Markers	Named positions (a plain `Marker2D` node)
Walkable areas	Where characters may walk
Characters in room	NPCs and the player placed in this room

Each group has its own + button to create one (for characters it is "Add character to room"). The header shows the room's name and has buttons to open the room's script, its state, and its state script. When no room is open, the tab shows only a hint to open a room in the Main tab.

The Audio tab

Audio files are organised by the **prefix of their file name**: `mx_` for music, `sfx_` for sound effects, `vo_` for voices and `ui_` for interface sounds. Each becomes an *audio cue* that you play with `A.<cue_name>.play()`. You will use this in Chapter 18.

The Tools tab

A collection of helpers for the project. Its contents change between versions; we do not use it in this book.

> **Try it:** Click through the four tabs. In the Main tab, confirm that there are no rooms yet. Click the Room tab; notice that it is empty or greyed out because no room is open.

13.5 The generated folders

After Setup, the FileSystem dock shows a folder `res://game/`. This is *your* game, as opposed to `res://addons/popochiu/`, which is the plugin. The layout, as it will look when we are done in Chapter 15:

```
res://
├── addons/
│   └── popochiu/                 <- the plugin; never edit
├── game/
│   ├── autoloads/                <- r.gd, c.gd, i.gd, d.gd, a.gd (generated)
│   ├── characters/
│   │   ├── player/
│   │   │   ├── character_player.tscn
│   │   │   ├── character_player.gd
│   │   │   ├── character_player.tres
│   │   │   └── character_player_state.gd
│   │   └── frog/ ...
│   ├── dialogs/
│   │   └── chat_with_frog/
│   │       ├── dialog_chat_with_frog.gd
│   │       └── dialog_chat_with_frog.tres
│   ├── gui/                      <- the GUI template you picked
│   ├── inventory_items/
│   │   └── key/
│   │       ├── inventory_item_key.tscn
│   │       ├── inventory_item_key.gd
│   │       ├── inventory_item_key.tres
│   │       └── inventory_item_key_state.gd
│   ├── rooms/
│   │   └── bedroom/
│   │       ├── room_bedroom.tscn
│   │       ├── room_bedroom.gd
│   │       ├── room_bedroom.tres
```

```
│   │       ├── room_bedroom_state.gd
│   │       ├── props/
│   │       │   └── drawer/ ...
│   │       └── hotspots/
│   │           └── door/ ...
│   ├── transition_layer/
│   ├── popochiu_data.cfg          <- the registry of your objects
│   └── popochiu_globals.gd        <- the Globals script (Chapter 15)
└── project.godot
```

Some of these folders appear as soon as Setup is done, and others only when you create the first object of that kind.

Reading the naming rule

Popochiu converts an object's name to *snake_case* (lower-case words with underscores) for file names, and puts the type in front:

You type	Folder	Files
Bedroom (room)	`rooms/bedroom/`	`room_bedroom.tscn`, `.gd`, `.tres`, `room_bedroom_state.gd`
Player (character)	`characters/player/`	`character_player.tscn` ...
BoneFlute (item)	`inventory_items/bone_flute/`	`inventory_item_bone_flute.tscn` ...
ChatWithFrog (dialog)	`dialogs/chat_with_frog/`	`dialog_chat_with_frog.gd`, `.tres` (no scene)

Rooms, characters and items have a scene (`.tscn`), a script (`.gd`), a resource (`.tres`) and a state script. Dialogs are different: a dialog tree is a Godot **Resource** and has no scene.

Inside a room folder, there are sub-folders for objects that belong to that room: `props/`, `hotspots/`, `regions/`, `walkable_areas/` and `markers/`. Each object has its own folder and scene, named after the type: for example `props/drawer/prop_drawer.tscn` with `prop_drawer.gd`, `hotspots/door/hotspot_door.gd`, `walkable_areas/floor/walkable_area_floor.gd`. A marker is a plain `Marker2D` node with no script, so its folder holds only a scene (`markers/from_hallway/marker_from_hallway.tscn`). A prop made with "Will this prop have interaction?" unticked (a background, say) still gets a script from the template (`props/background/prop_background.gd`); unticking the box only makes the prop not clickable, and you never need to open that script.

> **Warning:** Do not rename, move or delete these files in the FileSystem dock. The state scripts, the generated autoload scripts and `popochiu_data.cfg` all contain their paths. Rename and delete through the Popochiu dock, which updates everything. Deleting through the dock is safe.

13.6 The ten autoloads

When Popochiu is enabled, it registers a set of autoloads (Chapter 5). They are global names, available from any script in the project. Eight of them are single capital letters (the ones you will type most), and two have longer names. There are ten in all.

Name	Full role	What you use it for	Example
`R`	Rooms	Go to rooms, find objects in the current room	`await R.goto_room("Hallway")`
`C`	Characters	Reach any character, especially the player	`await C.player.say("Hi")`
`I`	Inventory	Add, remove, check items	`I.Key.add()`
`D`	Dialogs	Start and control conversations	`await D.ChatWithFrog.start()`
`E`	Engine	Waiting, cutscenes, save and load, camera	`await E.wait(0.5)`
`G`	Graphic interface	Block or hide the GUI, show system text	`G.block()`
`A`	Audio	Play sound cues	`A.mx_theme.play()`
`T`	Transition layer	Fades and screen transitions	`T.show_curtain()`
`Cursor`	The custom cursor	Show or hide the mouse cursor	(rarely used directly)
`Globals`	Your own variables	Remember things across the whole game	`Globals.key_taken`

R, C, I, D and A are generated

The scripts behind `R`, `C`, `I`, `D` and `A` live in `res://game/autoloads/` as `r.gd`, `c.gd`, `i.gd`, `d.gd` and `a.gd`. Popochiu **regenerates them every time you create an object through the dock**. That is how `I.Key` comes to exist: when you create an item named `Key`, Popochiu adds a variable to `i.gd` so that `I.Key` autocompletes. The same holds for `C.Frog`, `R.Bedroom` and `D.ChatWithFrog`.

This is a good reason not to create objects by copying files by hand: the autoload script would not know about them, and you would get "Identifier not found" errors.

E, G and T come from the engine

`E` is the engine: its scene (`popochiu.tscn`) is the root of the running game, with a layer for the GUI (layer 10), a transition layer (15) and the cursor (100). Layer numbers matter in Chapter 17, when you draw a mini-game above the room.

Globals is yours

`Globals` is a script at `res://game/popochiu_globals.gd`. It starts as a nearly empty `extends Node`. You add variables to it. In Chapter 15 you will declare the two variables the game needs:

```gdscript
# res://game/popochiu_globals.gd  (preview; you will write this in Chapter 15)
extends Node

var key_taken: bool = false
var door_unlocked: bool = false
```

When the game is saved, Popochiu automatically saves the `Globals` variables that have a static type of `bool`, `int`, `float`, `String` or `Array` (its source checks exactly these types). A variable without a type (`var x = false`) or of another type, such as a `Dictionary` or a `Vector2`, is *not* saved automatically; for those you write `on_save()` and `on_load()` functions in `Globals` (Chapter 18).

Per-object state

Each room, character and item also has its own **state**, kept in its `*_state.gd` file. You will meet it in Chapter 15 as `R.Bedroom.state`, `C.Frog.state` and `I.Key.state`.

Using autocomplete

Open any script and type `I.`: Godot lists the items. Type `R.` for the rooms and `C.` for the characters. If a name you just made does not appear, you probably did not create it through the dock.

> **Try it:** Open **Project > Project Settings** and look for the autoload list (the tab is named "Globals" or "Autoload" depending on the Godot version). Find all ten names in it and note which file each one points to. You will meet each of them in code in the next chapters.

13.7 How Popochiu scripts relate to Godot scenes

Everything you made in Chapters 2 to 10 follows one rule: **a scene is a tree of nodes, and a script extends one node type**. Popochiu uses the same rule. Nothing in Popochiu is magic. It is built from Godot's own parts.

The class hierarchy

Popochiu defines several classes that extend Godot's built-in ones:

Popochiu class	Extends	Meaning
`PopochiuRoom`	`Node2D`	A room
`PopochiuClickable`	`Area2D`	Something with a clickable shape
`PopochiuProp`	`PopochiuClickable`	A visible object in a room
`PopochiuHotspot`	`PopochiuClickable`	An invisible clickable zone
`PopochiuCharacter`	`PopochiuClickable`	A character
`PopochiuRegion`	`Area2D`	A zone triggered by walking into it
`PopochiuWalkableArea`	`Node2D`	Where characters can walk
`PopochiuInventoryItem`	`TextureRect`	An item in the inventory bar
`PopochiuDialog`	`Resource`	A dialog tree

If you remember Chapter 7, an `Area2D` detects the mouse (and bodies) in a shape. A prop *is* an `Area2D` with extra abilities. When you click it, Godot's normal input system sends the event to the area, and Popochiu turns it into a call to your function.

What a generated script looks like

Every generated script starts with a `@tool` line (items excepted) and an `extends` line that names the Popochiu class. Then Popochiu fills the file with commented templates for the functions you may want. For a prop, the heart of it is:

```
@tool
extends PopochiuProp

func _on_click() -> void:
    E.command_fallback()
```

(The real generated file has more in it: comments, a `#region Virtual` fold, and the other optional functions `_on_right_click()`, `_on_item_used()`, `_on_double_click()` and so on, each calling `E.command_fallback()` until you replace it. (Popochiu's template file says `PopochiuUtils.e.command_fallback()`; the dock shortens it to `E.command_fallback()` when it writes your script.) That call asks the current GUI to show its default reply, such as "Can't INTERACT with it" in Simple Click. This is a *fragment*, showing only what we need now. When you write your own handler, you replace the `command_fallback()` line.)

The `@tool` annotation tells Godot to run the script inside the editor too, which Popochiu needs to draw its selection outlines. For you, it makes no difference.

Rooms, characters and items additionally get these two lines, which connect the script to its state:

```
const Data := preload('room_bedroom_state.gd')

var state: Data = load("res://game/rooms/bedroom/room_bedroom.tres")
```

You do not edit those lines.

The three-file idea

For each room, character and item you get three things that you will touch:

File	Contains	You edit it to
`*.tscn` (scene)	The node tree, shapes and pictures	Draw polygons, set pictures, place things
`*.gd` (script)	Behavior	Write click handlers
`*_state.gd`	Variables to save	Add per-object variables (Chapter 15)

The `.tres` file is a resource that registers the object with Popochiu, and you normally leave it alone.

Why the dock

You could, in theory, build a room by hand: create a `Node2D`, attach `PopochiuRoom`, add nodes. You should not. A room made through the dock has dozens of small links that the dock sets up: the registry entry, the autoload variable, the state script, the right child nodes. A room built by hand will be missing some, and the failures are confusing. **Create through the dock; edit in the editor.**

> **Why it works:** The dock is a code generator. Writing the same boilerplate by hand for each room would be tedious and error-prone, so Popochiu does it for you, in the same way that a project template saves you from writing the same `project.godot` for every game.

13.8 Pitfalls

Symptom	Cause	Fix
No Popochiu entry in the Plugins tab	The folder is in the wrong place, e.g. `res://addons/addons/popochiu` or `res://popochiu-v2.2.1/addons/...`	The path to `plugin.cfg` must be `res://addons/popochiu/plugin.cfg`. Move the folder
Red errors flood the Output after enabling	The Godot and Popochiu versions do not match	Check the table in Section 13.2 and use the correct pair
`Identifier "R" not declared`	The plugin is disabled, or the project was not reloaded	Enable it in Project Settings > Plugins; use Project > Reload Current Project
The Setup popup never appears	It was shown and dismissed earlier, or the plugin was enabled before the files were complete	Disable and re-enable the plugin; read the Popochiu docs on setup
The dock is missing	It was moved or closed	Look in the other dock areas, or disable and re-enable the plugin
Godot says the project has no main scene when you press F5	There is no room yet, so no main scene is set	Normal for an empty project; Chapter 14 creates the first room and makes it the main scene
Strange errors after you renamed a file	You renamed it in the FileSystem dock	Undo with git (`git restore .`) and rename through the Popochiu dock

> **Warning:** If a project stops working after you copy it to another computer, check that the `addons` folder came with it and that the same versions are installed. The `.godot` folder does not need to be copied; Godot rebuilds it.

Lab 13: Create the weird-hotel project and install Popochiu

Time: 30 to 60 minutes. **Requirements:** Godot 4.7.2 installed and working; git installed (Chapter 11); an internet connection.

Steps

1. Open the Godot Project Manager. Click **New Project**. Name: `weird-hotel`. Pick a folder you can find again. Choose **Create Folder**. If there is a Version Control Metadata option, choose **Git**. Click **Create & Edit**.
2. In a browser, open https://github.com/carenalgas/popochiu. Find the version table in the README. Write down the pair for your Godot version. (For Godot 4.7: Popochiu 2.2.1.)
3. Open the Releases page. Download the zip named `popochiu-v2.2.1.zip` (or your correct version). Save it somewhere that is *not* inside the project.
4. Extract the zip into a temporary folder. Confirm that it contains a folder named `addons`, and that inside it is `popochiu`.
5. In Godot's FileSystem dock, right-click `res://` and choose **Open in File Manager**. Copy the `addons` folder into the project folder.
6. Return to Godot. Expand `res://addons/popochiu/` and check that `plugin.cfg` is directly inside it.

7. Choose **Project > Project Settings > Plugins**. Tick **Enable** for Popochiu. If asked, choose **Project > Reload Current Project**.
8. Look in the Output panel for the Popochiu greeting. Find the Popochiu dock (upper-left dock area) and write down the version shown at its bottom edge.
9. In the Setup popup, choose: step 1 **Modern / Hi-Res**; step 2 **Full-HD (1080p)** and a preview window size that fits your screen; step 3 **Simple Click**. Write down the choices. Press **Create**.
10. Open the FileSystem dock and find `res://game/`. Write down which sub-folders exist. Open `res://game/gui/` and confirm that the template was copied.
11. Find the Popochiu dock. Click through its four tabs. Confirm that there are no rooms yet.
12. Open **Project > Project Settings > Display > Window**. Write down the viewport width and height. You will need them in Chapter 14.
13. Open **Project > Project Settings > Globals > Autoload** (the tab is named Globals in Godot 4; check your version). Find `R`, `C`, `I`, `D`, `E`, `G`, `A`, `T`, `Cursor`, `Globals` in the list. Count them.
14. Press **F5** to run the empty game. Popochiu sets the project's main scene only when you create a room, and you have not made one yet, so **Godot itself** (not Popochiu) tells you that the project has no main scene and offers to let you select one. Choose to cancel or close that dialog; do not select a scene. This is expected. Write down the exact words of the message you see.
15. Make a first commit. In a terminal in the project folder:

```
git status
git add -A
git commit -m "Install Popochiu 2.2.1 and run Setup"
git log --oneline
```

Before you commit, look at `git status`. Confirm that `.godot/` does not appear in the list (it should be ignored) and that nothing larger than a few megabytes is included.

Expected result

- A project named `weird-hotel` containing `addons/popochiu/` and `game/`.
- The Popochiu dock, with four tabs and no rooms.
- Ten autoloads listed in Project Settings.
- A run with F5 that does not crash the editor (Godot's "no main scene" dialog is the normal result until Chapter 14 creates the first room).
- One git commit; `git log --oneline` shows it.
- A note listing: Godot version, Popochiu version, resolution, GUI template.

Chapter summary

- A plugin is a folder in `res://addons/` that adds editor tools and engine code to a Godot project. Popochiu is a plugin for adventure games.
- Plugin versions must match Godot versions. The pair used here is Godot 4.7 with Popochiu 2.2.1, but always read the README table for the current pairs.
- Install from the GitHub release zip: copy `addons` into the project, enable under Project Settings > Plugins, and complete the Setup popup.
- Setup creates `res://game/`, copies the chosen GUI template, sets the resolution and registers the autoloads. We choose Simple Click.
- The dock has four tabs (Main, Room, Audio, Tools). Rooms, characters, items and dialogs are created with + in the Main tab; room contents with + in the Room tab.

- The ten autoloads are `R`, `C`, `I`, `D`, `E`, `G`, `A`, `T`, `Cursor` and `Globals`. The scripts behind `R`, `C`, `I`, `D` and `A` are regenerated when you add objects.
- Each room, character and item is a scene, a script and a state script; dialogs are resources. They extend Godot node types such as `Area2D`.
- Create, rename and delete through the dock, never in the FileSystem dock. Commit after every working step.

Review questions

1. What is a Godot plugin, and in which folder does it live?
2. Which Popochiu version belongs to Godot 4.7, according to the table in this chapter, and where should you check for the current pair?
3. True or false: you should download "Source code (zip)" from the Releases page to install Popochiu.
4. Where is Popochiu enabled?
5. Name the GUI template this book uses and say what left-click and right-click do in it.
6. List the four tabs of the Popochiu dock.
7. What is the difference between the `Rooms` group in the Main tab and the `Walkable areas` group in the Room tab?
8. Which of the ten autoloads would you use to (a) go to another room, (b) make the player speak, (c) put an item in the inventory, (d) wait half a second?
9. Why must you create a new inventory item through the dock rather than copy an existing item's folder?
10. A prop's script begins with `extends PopochiuProp`. What Godot node type does `PopochiuProp` ultimately extend, and what does that tell you about how clicks are detected?

Exercises

1. **Practice.** Open `res://game/gui/` and list the files you find. Do not edit them. Write one sentence on what you think each is for.
2. **Practice.** In the Project Settings autoload list, write down the order of the ten autoloads and the file each points to.
3. **Apply.** Create a throwaway second project, install Popochiu again into it, but this time choose the 9 Verb GUI template. Run it and compare the interface with Simple Click. Delete the project when finished.
4. **Apply.** Read the Popochiu documentation page on the dock (https://carenalgas.github.io/popochiu/the-editor-handbook/popochiu-dock/). Write down one feature it describes that this chapter did not cover.
5. **Challenge.** Intentionally break the install: move `addons/popochiu` into a sub-folder, then try to enable the plugin. Record the exact error and then fix it. Write a three-line "troubleshooting card" you could give to a friend.

Common mistakes

- **Symptom:** The plugin does not appear in the Plugins tab. **Cause:** The zip was extracted into the project with an extra folder level. **Fix:** Make sure the path is `res://addons/popochiu/plugin.cfg`.
- **Symptom:** Countless errors in the Output after enabling. **Cause:** Wrong Godot or Popochiu version. **Fix:** Check the README table and reinstall the matching version.

- **Symptom:** `I.Key` is flagged as unknown. **Cause:** The item was not created through the dock, or the project was not reloaded. **Fix:** Create it through the dock; if needed, Project > Reload Current Project.
- **Symptom:** The GUI changes you made vanished. **Cause:** You re-ran Setup, which overwrites `res://game/gui/`. **Fix:** Keep GUI changes in git, and avoid re-running Setup.
- **Symptom:** Broken references after you tidied the FileSystem dock. **Cause:** You renamed or moved generated files. **Fix:** Restore with git; do all renames through the Popochiu dock.
- **Symptom:** `git status` shows thousands of files. **Cause:** `.godot/` is not ignored. **Fix:** Add `.godot/` to `.gitignore` before committing.

Further reading

- Popochiu documentation home: https://carenalgas.github.io/popochiu/
- Installing Popochiu: https://carenalgas.github.io/popochiu/getting-started/installing-popochiu/
- The Popochiu dock: https://carenalgas.github.io/popochiu/the-editor-handbook/popochiu-dock/
- Popochiu on GitHub (README with the version table, and Releases): https://github.com/carenalgas/popochiu
- Godot documentation, Installing plugins: https://docs.godotengine.org/en/stable/tutorials/plugins/editor/installing_plugins.html

Chapter 14: Rooms, Characters, Props and Hotspots

Learning objectives

By the end of this chapter you will be able to:

- Create rooms through the Popochiu dock and give them a background and a walkable area.
- Place markers and use them to position characters.
- Create the player character and explain how `C.player` refers to it.
- Distinguish props, hotspots and regions, and choose the right one for an object.
- Write `_on_click()` and `_on_right_click()` handlers that use `walk_to_clicked()` and `say()`.
- Use room hooks such as `_on_room_entered()` and move between rooms with `await R.goto_room()`.
- Build the Bedroom and Hallway of Weird Hotel with a drawer, a key and a locked door.
- Explain what Popochiu remembers when the player leaves a room, and why the facts of the game belong in `Globals` (Chapter 15) rather than in prop scripts.

Key terms

- **Room**: one screen of the game with a background, objects and characters.
- **Background**: the picture that fills the room; here it is usually a prop with no interaction.
- **Walkable area**: a polygon that defines where characters may walk.
- **Marker**: a named position in a room, made with a `Marker2D` node.
- **Player character**: the character the player controls, reached in code as `C.player`.
- **Prop**: a visible object in a room that can be clicked.
- **Hotspot**: an invisible clickable zone laid over the background.
- **Region**: a zone that reacts when a character walks into it, not when it is clicked.
- **Walk-to point**: the position a character walks to before interacting with an object.
- **Baseline**: a vertical position used to decide which of two overlapping objects is drawn in front.
- **Room hook**: a function Popochiu calls at a moment in the room's life, such as `_on_room_entered()`.
- **Coroutine**: a function that contains `await` and therefore pauses and resumes (Chapter 5).

14.1 What a room is

A room is one screen of the adventure. In terms you already know, a room is a Godot scene whose root is a `PopochiuRoom` (a `Node2D`), and whose children are the things in it: a background, a walkable area, props, hotspots, regions, markers and characters.

```
Bedroom (Node2D, with the PopochiuRoom script)
├── WalkableAreas (Node2D)
│   └── Floor (PopochiuWalkableArea)      <- where characters may walk
├── Props (Node2D)
│   ├── Background (PopochiuProp)         <- a picture; no interaction
│   └── Drawer (PopochiuProp)             <- clickable, has a script
├── Hotspots (Node2D)
│   └── Door (PopochiuHotspot)            <- invisible clickable zone
├── Regions (Node2D)                      <- empty in this chapter
├── Markers (Node2D)
│   └── FromHallway (Marker2D)            <- a named position
```

```
└── Characters (Node2D)
    └── CharacterPlayer (PopochiuCharacter)  <- placed in this room
```

This tree comes from Popochiu's source: the room scene starts with exactly the six empty container nodes `WalkableAreas`, `Props`, `Hotspots`, `Regions`, `Markers` and `Characters`, and every object you add through the dock is put inside the matching container. (In your Scene dock the objects you added are shown as instanced scenes, with a small "scene" icon. The name of a character you place by hand starts with `Character`, as in `CharacterPlayer`.)

Inside one prop, the nodes are:

```
Drawer (Area2D with the PopochiuProp script)
├── Sprite2D                <- shows the picture
├── AnimationPlayer
├── InteractionPolygon      <- CollisionPolygon2D: the clickable shape
└── ObstaclePolygon         <- NavigationObstacle2D: only used if the prop is an obstacle
```

A hotspot has only an `InteractionPolygon`; a character has a `Sprite2D`, an `AnimationPlayer`, an `InteractionPolygon`, a `ScalingPolygon` and an `ObstaclePolygon`; a walkable area has one child, a `NavigationRegion2D` named `Perimeter`.

The dock does the building

You never create these nodes with Godot's "Add child node" button. You create them with the Popochiu dock:

1. In the **Main** tab, press + next to Rooms to create the room.
2. Open the room (the "Open in Editor" button on its row).
3. In the **Room** tab, press + next to Props, Hotspots, Regions, Markers or Walkable areas to add each kind of object.

Each of those creates the node, the script (where it has one) and the registry entries. Each new object appears at the **centre of the room** (half the project's width and height), with a small default shape, so after creating an object you almost always move it and reshape its polygon.

The room's three files

Create a room named `Bedroom`, and Popochiu makes these in `res://game/rooms/bedroom/`:

File	Purpose
`room_bedroom.tscn`	The scene: what is in the room
`room_bedroom.gd`	The room script: hooks such as `_on_room_entered()`
`room_bedroom_state.gd`	Saved variables of the room (Chapter 15)

(Popochiu also creates `room_bedroom.tres`, a small resource that registers the room; you leave it alone.)

Making the first room the start

Popochiu has no separate "start room" setting. The room the game starts in is simply Godot's own **main scene**, the project setting `application/run/main_scene`, which Popochiu sets to the room's `.tscn` file. When you create a room and the project has no main scene yet, the create popup shows a tick box that says you have no main scene and asks whether this room should become it. Tick it for `Bedroom`. If the box is not shown (a

main scene already exists), or you forget, use the **Options** button at the end of the room's row in the Main tab and choose **Set as Main scene**. A heart icon on the row shows which room is the main scene. Pressing F5 then starts the game in that room.

Each room row also has a **Play scene** button, which runs only that room. It is useful for testing a room on its own.

> **Try it:** (You will do this properly in the lab.) Create a room named `Scratch` (do not tick the main-scene box), open it, and look at the Scene dock. Count the nodes. Then delete the room through the dock (the **Options** button on its row, then **Remove**; not the FileSystem dock) and check that `res://game/rooms/scratch/` disappears.

14.2 Backgrounds and placeholder pictures

Chapter 12 gave the rule: **make it with boxes first**. This section shows how to make a background without drawing anything.

Use a plain rectangle

A background can be a single-colour image. You need two things: a file, and the right size.

Find the size. Open **Project > Project Settings > Display > Window** and read the viewport width and height. If you used a Full HD preset in Chapter 13, they are 1920 and 1080. Make the picture that size, so that it fills the room exactly.

Make the file. Any image editor will do:

- **Windows Paint.** Open Paint > **Resize** > untick "Maintain aspect ratio" > choose Pixels > 1920 by 1080. Fill with a colour (use the paint-bucket). **File > Save as > PNG picture**. Save to a folder *outside* the project first.
- **GIMP, Krita, Photoshop** or any other editor: new image of that size, fill, **Export as PNG**.
- **PowerShell, if you prefer commands.** This script (Windows only) makes a plain coloured PNG without opening any program. It is a normal PowerShell script, not GDScript.

```
# make-box.ps1  usage: .\make-box.ps1 out.png 1920 1080 70 80 110
param($Path, $Width, $Height, $Red, $Green, $Blue)
Add-Type -AssemblyName System.Drawing
$bmp = New-Object System.Drawing.Bitmap ([int]$Width), ([int]$Height)
$gfx = [System.Drawing.Graphics]::FromImage($bmp)
$gfx.Clear([System.Drawing.Color]::FromArgb([int]$Red, [int]$Green, [int]$Blue))
$gfx.Dispose()
$bmp.Save((Join-Path (Get-Location) $Path), [System.Drawing.Imaging.ImageFormat]::Png)
$bmp.Dispose()
```

> **Note:** PowerShell variable names ignore upper and lower case, so `$G` (green) and `$g` would be the *same* variable. That is why the script uses long names such as `$Green` and `$gfx`. The script was tested in Windows PowerShell and PowerShell 7 on Windows 11; it writes the PNG into the folder you are in.

Save the script as `make-box.ps1` in the folder outside the project where you keep your pictures, open PowerShell in that folder, and run it once per picture. If Windows refuses to run scripts ("running scripts is disabled on this system"), run it this way instead, which allows it for this one command only and changes no settings:

```
powershell -ExecutionPolicy Bypass -File .\make-box.ps1 bedroom_bg.png 1920 1080 70 80 110
```

Run it five times, once per picture:

```
.\make-box.ps1 bedroom_bg.png 1920 1080 70 80 110
.\make-box.ps1 hallway_bg.png 1920 1080 90 70 60
.\make-box.ps1 player.png 80 160 230 170 40
.\make-box.ps1 drawer.png 200 140 130 90 50
.\make-box.ps1 key.png 64 64 240 220 60
```

Rooms of different colours help you see at once where you are. (Rename the saved files if you like; use only lower-case letters, digits and underscores in file names.)

Put pictures in a folder of your own

Make a folder `res://art/` in the FileSystem dock (right-click `res://` > New Folder). Copy your PNG files into it. Godot imports them automatically; they appear with a small thumbnail.

> **Warning:** Do not put your art inside `res://game/` or `res://addons/`. Those folders are managed by Popochiu. `res://art/` is yours.

Why not a ColorRect?

Godot can draw a coloured rectangle without a file, using a `ColorRect` node. Popochiu's props do not use one: a prop draws its picture with a `Sprite2D` child, which needs a texture (an image). So for a background made the Popochiu way you need an image file, which is why this book uses PNG files. (You could add a `ColorRect` to the room by hand, but it would not be a prop: the dock would not know about it, and it would not take part in Popochiu's drawing order.)

Assigning a picture

Every prop has a **Texture** property of its own, at the top of its Inspector, under the prop's script section. Popochiu copies whatever you put there to the prop's `Sprite2D` child. So:

1. Select the prop in the Scene dock (the prop itself, for example `Background`, not one of its children).
2. In the Inspector, find **Texture**.
3. Drag the PNG from the FileSystem dock onto the **Texture** field.

The prop's `Sprite2D` is *centred* on the prop's position, as Godot's `Sprite2D` is by default. Keep this in mind when you place things: the prop's position is the middle of its picture.

Characters work slightly differently: a character has no Texture property of its own, so you open the character's scene and set the **Texture** of its `Sprite2D` child (Section 14.4). Inventory items are `TextureRect` nodes, so you set the **Texture** of the item's root node (Lab step 6).

The Background prop

The usual way to make a background is as a prop with no interaction. In the Room tab, press + next to Props. A popup asks for a **Prop name** and has a tick box, "Will this prop have interaction?". Name it `Background`, and leave the box **unticked**, because a background is decoration only. A prop without interaction is not clickable; Popochiu still writes a script for it from the template (`prop_background.gd`), which you can leave alone. Popochiu puts it at `res://game/rooms/bedroom/props/background/`. Then set its texture to `bedroom_bg.png`.

Two details from Popochiu's source make backgrounds easy:

- New objects are placed at the centre of the room, and the picture is centred on that point, so a picture of exactly the project's size **fills the screen without any moving**. Do not set the position to `(0, 0)`: that would put the *centre* of the picture in the top-left corner. If you moved it by accident, set **Position** back to half the resolution, `(960, 540)` for 1920 x 1080.
- A prop named `Background` (or `Bg`) is automatically drawn behind everything else: Popochiu gives it a baseline at the top of the screen and a Z Index of -1.

14.3 Walkable areas and markers

Walkable areas

A **walkable area** is a polygon that tells Popochiu where characters may stand and walk. When the player clicks a point, Popochiu finds a path inside the polygon that avoids the outside. If you click outside the polygon, the character walks to the nearest reachable point.

Without a walkable area, nothing can walk. Create one in the Room tab with + next to Walkable areas and give it a name (we use `Floor`). The walkable area is a `Node2D` whose single child, `Perimeter`, is a Godot `NavigationRegion2D`; its outline is the polygon. The new area starts as a tiny square (20 x 20 pixels) at the centre of the room. Popochiu draws its own editing handles for this polygon in the 2D view:

1. Select the walkable area (`Floor`) in the Scene dock. Its outline appears in the 2D view with a white handle on each corner.
2. **Drag a handle** to move a corner. Drag the four corners out to the four corners of the floor in your picture.
3. **Click and drag on an edge** (between two corners) to add a new corner there.
4. **Right-click a handle** (or hover over it and press Delete) to remove a corner. A polygon always keeps at least three corners.

Rules of thumb:

- Cover only the floor, not the walls.
- Keep the polygon simple: four to eight points are enough.
- To make an obstacle (a table in the middle of the floor), you have two choices. You can shape the walkable polygon with a notch around it. Or, if the table is a prop, tick the prop's **Obstacle** property in the Inspector: its `ObstaclePolygon` then appears (drawn in its own colour) when the prop is selected, you shape it over the table's feet with the same handles, and characters walk around it.

> **Try it:** Make the polygon too small on purpose, press F5 and click outside it. Watch where the character goes. Then enlarge the polygon.

Markers

A **marker** is a named position. It is a plain Godot `Marker2D` node with no script. Create one with + next to Markers in the Room tab. Give it a name that says what it is for, such as `Entrance` or `FromHallway`. The new marker appears at the centre of the room. The node itself is hidden, but Popochiu draws a small handle for each marker in the 2D view: drag that handle (or set the marker's **Position** in the Inspector) to put it in the right place.

In code you use a marker with:

- `await C.player.teleport_to_marker("FromHallway")`: put the character there instantly.
- `await C.player.walk_to_marker("FromHallway")`: make the character walk there.
- `R.get_marker_position("FromHallway")`: get the `Vector2` position, if you need it.

Markers are the answer to "where does the player appear when they come through a door?" You will make one for each entrance.

14.4 The player character

Characters are clickable, too

A character is a `PopochiuCharacter`, which extends `PopochiuClickable` (and so `Area2D`). That is why you can click an NPC like any prop. The *player character* is simply the one that the game moves when you click the floor.

Creating the player

1. In the Main tab, press + next to Characters.
2. Name it `Player`.
3. If this is the first character in the project, the create popup asks whether it should become the Player-controlled Character (PC). Tick it. (If you missed it, use the **Options** button at the end of the row and choose **Set as Player-controlled Character (PC)**.) A player icon appears on the row.
4. Give it a picture: open the character (**Open in Editor**), select its `Sprite2D` child and drag `player.png` onto the **Texture** field. The sprite is centred on the character's position, but the position is the point Popochiu moves along the walkable area, so it should be where the character's **feet** touch the floor. So also set the `Sprite2D`'s **Offset** to `(0, -80)`, half the picture's height, so that the bottom of the picture sits on that point. Save the scene.

A single picture is enough. The character scene also has an `AnimationPlayer` for idle, walk and talk animations; Popochiu's source simply skips animation when that `AnimationPlayer` is empty, so you can add animations later (see the docs page *Select the main character*, https://carenalgas.github.io/popochiu/how-to-develop-a-game/characters/select-the-main-character/).

The files are in `res://game/characters/player/`.

C.player

The autoload `C` reaches every character by name. For a character called `Frog` you write `C.Frog`. For the player there is a special property, `C.player`, which always points at whichever character is currently the player. When the game starts, Popochiu reads the PC you chose in the dock (it is stored in `res://game/popochiu_data.cfg`) and assigns that character to `C.player`; if you never chose one, it uses the first character in its list. Using `C.player` rather than `C.Player` is a habit worth forming: if a later chapter has the game switch to another playable character, your scripts keep working.

```
# C.player and C.Player refer to the same character as long as Player is the player
await C.player.say("Hello.")
```

Putting the player in a room

Every room has a property **Has Player** (select the room's root node and look in the Inspector), and it is on by default. When such a room loads and the player character is not already in it, Popochiu adds the player automatically and puts it **in the middle of the room**. So the player always arrives; the question is only *where*.

To choose the starting place yourself, place the character by hand: in the Room tab, press **Add character to room** (the + in the *Characters in room* group) and choose `Player`. A node named `CharacterPlayer` appears under `Characters`. Drag it to where you want the player to start. When a room is entered through a door, a marker and the `_on_room_entered()` hook decide the position instead (Section 14.7).

(Turn **Has Player** off only for rooms in which the player must not appear, such as a title screen.)

Walking

Characters walk when you tell them to, and the player walks when you click the floor. In code:

```
await C.player.walk_to(Vector2(400, 700))        # to a position
await C.player.walk_to_clicked()                  # to the clicked object's walk-to point
await C.player.walk_to_marker("Entrance")         # to a marker
```

Each of these takes time, so each is a **coroutine** and you put `await` in front. Without `await` the script continues immediately and the next action starts while the character is still walking. This is the most common beginner bug in Popochiu (see Common mistakes at the end of the chapter).

Facing

`face_clicked()` makes the character turn toward the clicked object. There are also `face_up()`, `face_down()`, `face_left()`, `face_right()` and diagonal versions, and `face_direction(position)`.

Depth: the baseline

When two objects overlap, Popochiu decides which is in front by a **baseline**, a vertical position on the object. Things with a larger y are drawn in front. A character standing behind a tree is hidden by it because the tree's baseline is lower on the screen than the character's.

Props, hotspots and characters (everything built on `PopochiuClickable`) have three related properties in the Inspector:

Inspector name	Script name	Meaning
Baseline	`baseline`	The y position of the baseline, measured from the *centre* of the object; positive is lower. Default 0.
Walk To Point	`walk_to_point`	Where a character stands to interact, measured from the object's position. Default `(0, 0)`, the object's own position.
Look At Point	`look_at_point`	The point a character turns to look at. Default `(0, 0)`.

When you select the object, Popochiu also shows each of these as a labelled handle in the 2D view ("Baseline", "Walk To Point", "Look At Point"), which you can drag. For a prop standing on the floor, drag the Baseline handle to the bottom of its picture and the Walk To Point handle onto the floor in front of it.

In the early chapters nothing overlaps, so you can ignore baselines. Remember that they exist when you place a character behind furniture.

14.5 Props, hotspots and regions

Popochiu has three kinds of "things in a room that react". Choose the right one.

Kind	Visible?	Reacts to	Use for	Weird Hotel
Prop	Yes, has a picture	Click	Objects the player sees and uses; also decoration	Drawer, Jar
Hotspot	No	Click	A spot on the background picture	Door, Ceiling
Region	No	A character walking in	Triggers and zones	(not used)

Think of it like this: if the object is drawn *as a separate picture* (and may be taken, hidden, moved or animated), it is a **prop**. If it is part of the background art (a window painted into the wall), it is a **hotspot**. If it should react when somebody walks *there*, it is a **region**.

Props

Create with + next to Props. The popup asks for a **Prop name** and has the tick box "Will this prop have interaction?". Tick it for anything the player can click. The prop's script, where you write its clicks, is `res://game/rooms/<room>/props/<name>/prop_<name>.gd` (the name in snake_case, so `Drawer` gives `props/drawer/prop_drawer.gd`).

The clickable shape of a prop is its `InteractionPolygon`, and it starts as a small 24 x 24 square around the prop's position. Setting a texture does **not** resize it. So after giving a prop its picture, select the prop and drag the corner handles of its interaction polygon (Popochiu draws them in the 2D view, as for walkable areas) until the shape covers the picture.

Hotspots

Create with + next to Hotspots. A hotspot is an `Area2D` whose only child is its `InteractionPolygon`; it gets the script `hotspots/<name>/hotspot_<name>.gd` (so `Door` gives `hotspot_door.gd`, and `BackDoor` gives `hotspot_back_door.gd`). Like a prop, a new hotspot appears at the centre of the room as a 24 x 24 square. Work in this order:

1. Move the hotspot itself: set its **Position** in the Inspector to the middle of the thing it represents (for example the middle of the door in the picture).
2. Drag the corners of its polygon so that it covers the part of the background that should be clickable. Make it a little larger than the object, so the player does not have to aim carefully (the pixel hunt of Chapter 12).
3. Set its **Walk To Point** to a spot on the floor in front of it (Section 14.4).

The order matters because the walk-to point is measured from the hotspot's *position*, not from its polygon. A hotspot left at the centre of the room with only its polygon dragged to the right edge would send the player to the centre of the room.

Regions

Regions are also polygons, but they detect characters entering and leaving. Their hooks are:

```
func _on_character_entered(chr: PopochiuCharacter) -> void:
func _on_character_exited(chr: PopochiuCharacter) -> void:
```

Those are two fragments of what a region script offers. A region script might look like this, to make the player comment when they walk onto a rug (the example is not part of the lab):

```
@tool
extends PopochiuRegion

func _on_character_entered(chr: PopochiuCharacter) -> void:
    super(chr)  # keep the default behavior (the region's tint colour on the character)
    if chr == C.player:
        await C.player.say("This rug is watching my feet.")
```

You do not need regions for Weird Hotel. Knowing they exist will be useful later: a trapdoor, a dark corner, or a spot that makes a character slow down.

Is it clickable? Layers and the GUI

Clicks only reach props and hotspots when the GUI is not blocking them. During a cutscene or a queue (Chapter 16), Popochiu blocks the GUI, and clicking does nothing. This is correct and not a bug.

14.6 Clicks, right-clicks and speech

Handlers

Every prop, hotspot and character has a script with *virtual functions*, functions that Popochiu calls for you when something happens (Chapter 5). The ones you need now:

Function	Called when
`_on_click()`	The player left-clicks the object (in Simple Click: the main action)
`_on_right_click()`	The player right-clicks the object (in Simple Click: look at it)
`_on_item_used(item: PopochiuInventoryItem)`	The player uses an inventory item on the object: in Simple Click, clicks the item in the inventory bar to select it, then clicks the object (Chapter 15)

There are also `_on_double_click()`, `_on_middle_click()`, `_on_movement_started()` and `_on_movement_ended()`.

Until you write a handler, the generated one calls `E.command_fallback()`, which shows the GUI's default reply. In Simple Click these replies are "Can't INTERACT with it" (left click), "Can't EXAMINE it" (right click) and "Can't USE ... with ..." (an item on an object). That is better than silence, but Chapter 12 asked you to write your own lines.

> **Note:** If you have seen Popochiu examples that use `_on_look`, `_on_use` or `_on_enter`, those names are not part of Popochiu 2.2.1. The names above are the right ones. The 9 Verb and Action Bar (Sierra-style) GUIs can also call functions named after their verbs, such as `on_look_at()`, with no leading underscore; with Simple Click you do not need them.

The pattern: walk, then act

Most click handlers have this shape:

```
func _on_click() -> void:
    await C.player.walk_to_clicked()   # go there, wait until arrived
    await C.player.say("Some line.")   # speak, wait until the text is gone
```

`walk_to_clicked()` makes the player walk to the clicked object's walk-to point. If you do not set one, the walk-to point is `(0, 0)`, which means the object's own position; if that spot is not on the walkable area, the player walks as close to it as the walkable area allows. Set **Walk To Point** for every clickable object so the player stops in a sensible place.

say

`say(dialog: String)` makes the character speak a line. A speech bubble or text appears over the character, and the function finishes when the text is done. Because it takes time, always `await` it:

```
await C.player.say("A drawer that hums faintly.")
await C.Frog.say("I am the landlord.")
```

Two `say` calls in a row, each with `await`, show two lines one after the other. Without `await` the second line would appear over the first.

Right-click is cheap and valuable

Write a `_on_right_click()` for every object. As Chapter 12 said, the look line is the cheapest writing in the game. Usually the player does not need to walk for a look, so you can skip `walk_to_clicked()`:

```
func _on_right_click() -> void:
    await C.player.say("A drawer that hums faintly.")
```

Queues, briefly

Popochiu offers `E.queue([...])` for running a list of actions as a cutscene, and every action has a `queue_` twin. We use plain `await` in this chapter. Do not call a `queue_*` function outside of `E.queue([...])`; it silently does nothing. Chapter 16 explains queues.

14.7 Room hooks and travel between rooms

Hooks

A room script has virtual functions, called hooks, that Popochiu calls at moments of the room's life:

Hook	Called
`_on_room_entered()`	When the room is loaded and about to be shown
`_on_room_transition_finished()`	When the fade-in has ended and the player can act
`_on_room_exited()`	When the player leaves
`_on_restore_from_savegame()`	When the room is restored by loading a save

The difference between the first two is important. `_on_room_entered()` runs when the room is in the scene tree but still hidden behind the transition, so it is the place to *arrange* things (position the player, hide props). `_on_room_transition_finished()` is the place for things the player should *see* happen (a line of speech on arrival).

Open the room script (`room_bedroom.gd`) from the Main tab with the **Open in Script** button. The generated file already contains all four hooks, each with only `pass` in its body, plus two lines near the top that link the room to its state (`const Data := preload(...)` and `var state: Data = load(...)`; leave them alone). To add behaviour, replace the `pass` of the hook you want. The `_on_room_entered()` function should end up like this (a fragment; everything else in the file stays as generated):

```
func _on_room_entered() -> void:
    if C.player.last_room == "Hallway":
        await C.player.teleport_to_marker("FromHallway")
```

`C.player.last_room` is the *name of the room the player came from*, as a string; Popochiu sets it just before it leaves a room. It is an empty string on the first room of a new game. Here, if we arrive from the Hallway we appear at a marker in front of the Door; otherwise the player stays where it is (at its place in the editor, or in the middle of the room if you did not place it). `teleport_to_marker()` is a coroutine like the walk functions, so it gets an `await` too.

Travel

To move to another room:

```
await R.goto_room("Hallway")
```

- The argument is the room's **script name** as shown in the dock, `"Hallway"`, in quotes.
- The default is a fade out and in. `R.goto_room("Hallway", false)` skips the transition.
- It is awaitable: `await` waits for the fade-out, and the call returns as soon as Popochiu has asked Godot to change scene, *before* the new room exists. Godot then frees the old room (with the script that called `goto_room`) and builds the new one, whose hooks run. So never put code after `await R.goto_room(...)` in the old room's scripts; put "on arrival" code in the new room's `_on_room_entered()` or `_on_room_transition_finished()`.

Do not try `R.Hallway.go_to()` or something similar; it does not exist.

Both rooms need ways in and out

A room with no exit traps the player. Plan every door with:

1. A clickable object in room A that calls `goto_room("B")`.
2. A marker in room B where the player appears (`FromA`).
3. A clickable object in room B that goes back.
4. A marker in room A where the player appears when they return (`FromB`).

14.8 Worked example: the Drawer, as first written

Here is the Drawer script you will type in the lab. It is the **original, simple version** from the first course. It remembers whether the key was taken in a local variable. Read it carefully, because Section 14.9 shows why this version has a problem.

```
# res://game/rooms/bedroom/props/drawer/prop_drawer.gd
# Attached to the Drawer prop in the Bedroom.
@tool
extends PopochiuProp

# Remembers if we already took the key.
# This lives inside this script. Section 14.9 tests how well it remembers;
# Chapter 15 moves it to Globals.
var key_taken: bool = false

func _on_click() -> void:
    await C.player.walk_to_clicked()

    if key_taken:
        await C.player.say("Just old socks now.")
    else:
        await C.player.say("A key! It is warm. Why is it warm?")
        key_taken = true
        await I.Key.add()

func _on_right_click() -> void:
    await C.player.say("A drawer that hums faintly.")
```

What it does:

1. Walks to the drawer.
2. If the key is already taken: a joke line.
3. Otherwise: a line, then `key_taken = true`, then `I.Key.add()` puts the item `Key` in the inventory. `add()` is a coroutine (the inventory bar may animate), so it gets an `await`. We set the flag *before* waiting, so the fact "the key is taken" is recorded even while the animation runs.

`I.Key` exists only if you created an inventory item named `Key` in the dock.

And the Door, also in its first version:

```
# res://game/rooms/bedroom/hotspots/door/hotspot_door.gd
# Attached to the Door hotspot in the Bedroom.
@tool
extends PopochiuHotspot

# Is the door unlocked? Local for now; Chapter 15 moves it to Globals.
var unlocked: bool = false
```

```
func _on_click() -> void:
    await C.player.walk_to_clicked()

    if unlocked:
        await R.goto_room("Hallway")
    else:
        await C.player.say("Locked. The keyhole is breathing.")

func _on_right_click() -> void:
    await C.player.say("A door. It looks shy.")

# Called when the player uses an inventory item on the door.
func _on_item_used(item: PopochiuInventoryItem) -> void:
    if item == I.Key:
        await C.player.walk_to_clicked()
        await C.player.say("It fits!")
        unlocked = true
        await I.Key.remove()
    else:
        await C.player.say("That won't open it.")
```

Notice:

- `_on_item_used` receives the item that was used, and we compare it with `I.Key`. If it is something else, a default line answers (Chapter 12, Rule 5).
- `I.Key.remove()` takes the key out of the inventory. Like `add()`, it is awaitable.
- The file names follow Popochiu's naming pattern, which is the same for every room object: `prop_drawer.gd` for the prop `Drawer`, `hotspot_door.gd` for the hotspot `Door` (and `hotspot_back_door.gd` for `BackDoor`). The dock's **Open in Script** button always opens the right one.

> **Why it works:** This is the lock-and-key of Chapter 12 in 20 lines. The key is added by one script, and used by another, and what links them is the inventory: `I.Key`.

14.9 Where does the game remember things?

The code above passes a quick test. You take the key, use it on the door and go through. That is exactly the check the first course asks for. But a game must also remember what happened when the player leaves and comes back, and you should test that on purpose.

Predict first

Before you run the experiment in the lab, write down your prediction for this sequence:

1. Take the key from the drawer.
2. Use the key on the door.
3. Walk through the door into the Hallway.
4. Walk back into the Bedroom.
5. Click the drawer.
6. Click the door.

What should happen in a *correct* game? The drawer should say "Just old socks now." The door should open straight away.

What Popochiu actually does

When the player leaves a room, Godot throws the room scene away, and when the player comes back it builds a fresh copy, with every script starting from its initial values. If that were the whole story, `key_taken` and `unlocked` would be `false` again and the drawer would hand out a second key.

But Popochiu 2.2.1 does more. Just before leaving a room, it makes a note of each clickable prop, hotspot, region and walkable area in it: the **variables declared in the object's script** (its own and those of the Popochiu class it extends, such as `walk_to_point` and `clickable`), as long as each variable has a simple *static type*: `bool`, `int`, `float`, `String` or an `Array`. Node properties that come from Godot itself, such as `position` and `visible`, are not script variables, so they are *not* in the note for props and hotspots (characters are handled separately and do keep their position). When the room is built again, Popochiu copies the noted values back before `_on_room_entered()` runs. (The official tutorial relies on this: its "add a prop" page keeps a `var first_time_seen := true` in a prop script.) So for the scripts above, which declare `var key_taken: bool = false` and `var unlocked: bool = false`, the source code says:

- the drawer **remembers**: it says "Just old socks now";
- the door **remembers**: it opens straight away.

Run the experiment anyway (Lab 14, Part G) and write down what you see. A prediction is only a prediction until you test it.

Where this memory breaks

Popochiu's note-taking has limits, and each limit is an easy way to make a bug:

- **Only statically typed variables are noted.** `var key_taken: bool = false` and `var key_taken := false` both have the type `bool`. But `var key_taken = false`, without a type, is a *Variant*, and Popochiu skips it. That one missing `: bool` makes the drawer forget and give a second key. You will try this in the lab.
- **Only simple types are noted.** A `Dictionary` or a `Vector2` in a prop script is not noted and starts from its initial value again.
- **Hidden or moved props come back.** Because `visible` and `position` are not noted for props, a prop you hid with code reappears when the room is rebuilt, unless the room's `_on_room_entered()` hides it again from a remembered fact (Chapter 15 does exactly this for the bone and twine).
- **Only room objects are noted.** A plain variable in the *room's* own script (`room_bedroom.gd`) is not part of the note.
- **Nobody else can read it easily.** `key_taken` lives inside the Bedroom's Drawer. A script in the Hallway, the Frog's dialogue or the boss fight cannot ask the Drawer, because the Drawer does not exist while the player is in another room.

Why this matters for a real game

In Chapter 12 you learned the rule: *never make the game unwinnable* and *the world should remember what happened*. A drawer that regrows its key breaks the puzzle, and here it would take only one missing type annotation. The facts of the game, such as "the key has been taken" or "the door is open", are needed by several scripts in several rooms, so they belong in one place that every script can read and that lives for the whole game.

The fix, previewed

Chapter 15 moves the two facts to `Globals`, where they live for the whole game and can be read from anywhere:

```
# preview: not yet! In Chapter 15 you will use these
Globals.key_taken
Globals.door_unlocked
```

`Globals` is saved with the game as well, which will make the saving in Chapter 18 almost free. Do not change the scripts to `Globals` now; the point of the lab is to see the problem first.

> **Try it:** Add `print("Drawer clicked, key_taken = ", key_taken)` as the first line of `_on_click()` in the Drawer script. Run the game and watch the Output panel as you leave and return to the Bedroom. This is the debugging habit from Chapter 11: when you suspect state, print it.

Lab 14: Bedroom, Hallway, Player, Drawer and Door

Time: 60 to 90 minutes. **Requirement:** the `weird-hotel` project from Lab 13, committed.

Part A: pictures

1. Make these PNGs using the method in Section 14.2, or any pictures you like: `bedroom_bg.png` and `hallway_bg.png` (project resolution, probably 1920 x 1080), `player.png` (80 x 160), `drawer.png` (200 x 140), `key.png` (64 x 64).
2. In the FileSystem dock make a folder `res://art/` and copy the five files in. Check that they show thumbnails.

Part B: the player and the key

3. In the Popochiu dock, Main tab, press + next to Characters. Name it `Player`. Because it is the first character, the popup asks whether it should become the Player-controlled Character (PC): tick it. Create it.
4. Check that the player icon appears on the `Player` row. (If not: the row's **Options** button > **Set as Player-controlled Character (PC)**.)
5. Open the Player scene (**Open in Editor**). Select its `Sprite2D` child, drag `player.png` onto **Texture**, and set **Offset** to `(0, -80)` (Section 14.4). Save the scene (Ctrl+S).
6. Press + next to Inventory items. Name it `Key`. Open it, select its root node (a `TextureRect`) and drag `key.png` onto its **Texture**. Save. (Chapter 15 covers items in depth.)

Part C: the Bedroom

The coordinates below assume a 1920 x 1080 project. For another resolution, scale them.

7. Press + next to Rooms. Name it `Bedroom`. Tick the box that offers to make it the main scene. Create. Open it in the editor (**Open in Editor**). A heart icon on its row confirms it is the main scene.
8. Room tab > + next to Props. Name: `Background`. Leave "Will this prop have interaction?" unticked. Create. Select it and set its **Texture** to `bedroom_bg.png`. Do not move it: it is already centred at `(960, 540)` and fills the screen.

9. Room tab > + next to Walkable areas. Name: `Floor`. Select it; in the 2D view drag its four corner handles to the four corners of the floor: roughly the bottom third of the picture, from y = 720 down to y = 1060, nearly the full width. Keep it simple.
10. Room tab > + next to Markers. Name it `FromHallway`. Set its **Position** to about `(1650, 850)`, on the floor in front of where the door will be.
11. Room tab > **Add character to room** (in the *Characters in room* group): choose `Player`. Set the new `CharacterPlayer` node's **Position** to about `(300, 900)`, on the floor at the left. (If you skip this step, Popochiu puts the player in the middle of the room.)
12. Room tab > + next to Props. Name: `Drawer`. Tick "Will this prop have interaction?". Create. Select it and:
    - set **Texture** to `drawer.png`;
    - set **Position** to about `(700, 760)`;
    - drag the corners of its interaction polygon (the small square in the middle of the drawer) out to the corners of the picture, so the whole drawer is clickable;
    - set **Walk To Point** to `(0, 120)`, a spot on the floor just below the drawer, and **Baseline** to `70`, the bottom edge of the 140-pixel-tall picture.
13. Room tab > + next to Hotspots. Name: `Door`. Create. Select it and:
    - set **Position** to about `(1800, 560)`, the right edge of the room;
    - drag the corners of its polygon into a tall rectangle, about 200 wide and 600 high; make it a little too large;
    - set **Walk To Point** to about `(-150, 290)`, a spot on the floor in front of the door.
14. Save the scene (Ctrl+S).

Part D: the scripts

15. In the Main tab, row `Bedroom`, click **Open in Script**. The file already contains `func _on_room_entered() -> void:` with `pass` in it. Replace that `pass` with the two-line `if` from Section 14.7. Do not add a second `_on_room_entered()`.
16. In the Room tab, open the script of `Drawer` and replace its content with the Drawer script in Section 14.8. Keep the first lines `@tool` and `extends PopochiuProp`.
17. Open the script of `Door` and replace its content with the Door script in Section 14.8. Check the first lines (`@tool`, `extends PopochiuHotspot`).
18. Save all (Ctrl+Shift+S) and check the Output and Debugger panels for red errors. The line `R.goto_room("Hallway")` is fine for the editor even though the Hallway does not exist yet: the room name is only a string. If you opened the Door and clicked it before Part E, Popochiu would print an error saying it can't go to room Hallway because it doesn't exist.

Part E: the Hallway

19. Create a room `Hallway` (the main-scene box is no longer offered, because the Bedroom already is the main scene). Open it.
20. Add a non-interactive prop `Background` with `hallway_bg.png`. Add a walkable area `Floor` and shape it over the floor, as in step 9.
21. Add a marker `FromBedroom` at about `(300, 850)`, near the left side.
22. Optional: add the Player to the room (Room tab > Add character to room) near the marker. It is not needed, because the hook in step 25 places the player on arrival.
23. Add a hotspot `BackDoor`. Set its **Position** to about `(120, 560)`, shape its polygon into a tall rectangle on the left edge, and set its **Walk To Point** to about `(150, 290)`.

24. Open the BackDoor's script and replace its content with:

```
# res://game/rooms/hallway/hotspots/back_door/hotspot_back_door.gd
@tool
extends PopochiuHotspot

func _on_click() -> void:
    await C.player.walk_to_clicked()
    await R.goto_room("Bedroom")

func _on_right_click() -> void:
    await C.player.say("The door I came from. It waved.")
```

25. Open the Hallway room script (Main tab, row `Hallway`, **Open in Script**) and replace the `pass` inside its existing `_on_room_entered()` so that the function reads (a fragment; leave the rest of the file as generated):

```
func _on_room_entered() -> void:
    if C.player.last_room == "Bedroom":
        await C.player.teleport_to_marker("FromBedroom")
```

26. Save all.

Part F: test the puzzle

27. Press **F5**. The game starts in the Bedroom with the Player on the floor at the left.
28. Click the floor. The Player walks there. Move the mouse over the Drawer and the Door: their names appear (Simple Click's hover text).
29. Click the Door. The Player walks to the door's walk-to point and says the locked line.
30. Right-click the Drawer. A look line. Left-click it. The key goes into the inventory. To see the inventory, move the mouse to the top edge of the game window: the inventory bar slides down.
31. Click the Drawer again: "Just old socks now."
32. Use the Key on the Door: in the inventory bar, click the Key to select it, then click the Door. "It fits!" and the key leaves your inventory.
33. Click the Door. Fade to the Hallway; the Player appears at `FromBedroom`.
34. **Commit:**

```
git add -A
git commit -m "Bedroom, Hallway, Player, drawer key and door (local variables)"
git push
```

(Skip `git push` if you have no remote repository.)

Part G: test the memory on purpose

35. In the Hallway click the `BackDoor` hotspot to return to the Bedroom. The Player should appear at `FromHallway`.
36. Click the Drawer. Write down what happens. Click the Door. Write down what happens. Compare with your prediction from Section 14.9.
37. Now break the memory on purpose. Close the game. In the Drawer script change `var key_taken: bool = false` to `var key_taken = false` (remove `: bool`). Save.

38. Run the game again and repeat: take the key, use it on the Door, go to the Hallway, come back, click the Drawer. Write down what happens. Then answer in writing: *Which variable forgot, in which script does it live, and why did removing the type change anything?*
39. Leave the drawer in this forgetful state; Chapter 15 starts from it. Add a note to your notes file: "Drawer forgets when its variable has no type; facts belong in Globals (Chapter 15)."
40. Commit again (`git add -A`, then `git commit -m "Drawer memory experiment"`).

Expected result

- The Bedroom and Hallway exist; the game starts in the Bedroom; the Player walks on the floor.
- The Drawer gives the Key once; a second click gives the socks line.
- The Key opens the Door; the Door fades to the Hallway; the BackDoor returns you, and each time you arrive at the right marker.
- In step 36, with typed variables, Popochiu 2.2.1's source code says the Drawer gives the socks line and the Door opens at once. In step 38, with the untyped variable, the Drawer gives a **second** Key. Whatever you see, you recorded it, and you can say which variable is responsible.
- Two git commits, as in steps 34 and 40.

Chapter summary

- A room is a scene whose root is a `PopochiuRoom`, with six container nodes (`WalkableAreas`, `Props`, `Hotspots`, `Regions`, `Markers`, `Characters`) that hold its objects. Create all of them through the dock; each new object appears at the centre of the room.
- A placeholder background is a plain-coloured PNG at the project's resolution, set as the **Texture** of a non-interactive prop named `Background`; put your art in `res://art/`.
- A walkable area is a polygon (its `Perimeter` child); markers are named `Marker2D` positions used with `teleport_to_marker` and `walk_to_marker`. Edit polygons with Popochiu's handles in the 2D view.
- `C.player` is the current player character, chosen in the dock. Rooms add the player automatically (in the middle) unless you place it; walking is awaitable.
- A prop is a picture you can click, a hotspot is an invisible zone, a region reacts to someone walking in.
- The handlers are `_on_click()`, `_on_right_click()` and `_on_item_used(item)`; `_on_look`, `_on_use` and `_on_enter` do not exist.
- Always `await` walking and speaking; write a look line for every object.
- Rooms have hooks (`_on_room_entered()` and more), and you travel with `await R.goto_room("Name")`.
- Popochiu notes the statically typed simple variables of props and hotspots when you leave a room and restores them on return, but an untyped variable, a complex type or a room-script variable is forgotten, and no other room can read them. The facts of the game belong in `Globals` (Chapter 15).

Review questions

1. Which three files does the dock create for a room named `Bedroom`, and what is each for?
2. How do you make `Bedroom` the room where the game starts?
3. What is the difference between a prop and a hotspot? Give one example of each from Weird Hotel.
4. True or false: a character can walk anywhere inside the room even if there is no walkable area.
5. What does a marker do, and which node type is it?
6. What is the difference between `C.player` and `C.Frog`?
7. Why do we write `await` before `C.player.say("Hi")`?

8. Which function runs when the player right-clicks a prop, and which runs when an inventory item is used on it?
9. What happens to `_on_look`, `_on_use` and `_on_enter` in Popochiu 2.2.1?
10. Write the line that takes the player from the Bedroom to the Hallway.

Exercises

1. **Practice.** Add a hotspot named `Wall` to the Bedroom that covers the upper part of the background. Give it a `_on_right_click()` that describes the room in one deadpan sentence.
2. **Practice.** Give the Drawer a different second-click joke, and add a third state: after the third click it says "I think the drawer is embarrassed." Use a second variable, `var clicks: int = 0`, as a counter. Leave the room and come back: does the count continue? Then remove the `: int` and test again. Explain the difference using Section 14.9.
3. **Apply.** Add the Jar (a prop) and the Ceiling (a hotspot) to the Hallway. Give each a `_on_right_click()` look line from the Chapter 12 design table. Leave `_on_click()` with a placeholder line such as "Not yet." (Chapter 17 will fill them in.)
4. **Apply.** Add a Region to the Hallway in front of the BackDoor. When the player enters it for the first time, say a line. Use the fragment in Section 14.5. Note that "for the first time" needs a remembered fact; decide where you would keep it so that it still works after leaving the Hallway and coming back.
5. **Challenge.** Add a third room, `Closet`, joined to the Hallway by a door, with entrance markers on both sides. Make sure the player appears in the right place when entering from each side. Draw the room map and update the design document from Chapter 12.

Common mistakes

- **Symptom:** The player does not move when you click the floor. **Cause:** No walkable area, the polygon is still the tiny default square, or the room's **Has Player** is off. **Fix:** Check the Room tab for a walkable area, drag its corners over the floor, and check **Has Player** on the room's root node.
- **Symptom:** Two lines of speech overlap, or the player talks while still walking. **Cause:** A missing `await`. **Fix:** Put `await` before every `walk_to...`, `say`, and `goto_room`.
- **Symptom:** `Invalid call. Nonexistent function '_on_look'`-style errors, or the right-click does nothing. **Cause:** The handler has the wrong name. **Fix:** Use `_on_right_click()`.
- **Symptom:** `I.Key` is not recognized. **Cause:** The item was not created through the dock, or has a different name (`key`, `Key` with a space). **Fix:** Create `Key` in the dock; names are case-sensitive.
- **Symptom:** `R.goto_room("Hallway")` shows an error. **Cause:** The room does not exist, or the name has a different spelling. **Fix:** Use exactly the script name shown in the dock.
- **Symptom:** The player appears in the middle of the room, or in the wall, after a door. **Cause:** The marker is misplaced, or `_on_room_entered()` does not teleport (a second `_on_room_entered()` was added below the generated one, or the marker name is misspelt). **Fix:** Keep one `_on_room_entered()`, and check that the marker's name in the hook matches the marker in the Room tab, character by character.
- **Symptom:** The player walks to the middle of the room when you click the Door. **Cause:** Only the hotspot's polygon was moved; the hotspot's position (and so its walk-to point) is still at the room's centre. **Fix:** Set the hotspot's **Position** to the door, then its **Walk To Point** (Section 14.5).
- **Symptom:** Only a small square in the middle of a prop reacts to clicks. **Cause:** The interaction polygon is still the default 24 x 24 square. **Fix:** Drag its corners over the picture.
- **Symptom:** Clicks stop working for a moment. **Cause:** The GUI is blocked during a cutscene or a queue. **Fix:** This is normal; it unblocks when the action finishes.

Further reading

- Creating the first room: https://carenalgas.github.io/popochiu/how-to-develop-a-game/rooms/create-the-first-room/
- Adding a prop (includes a custom variable in a prop script): https://carenalgas.github.io/popochiu/how-to-develop-a-game/rooms/add-a-prop/
- Adding a walkable area: https://carenalgas.github.io/popochiu/how-to-develop-a-game/rooms/add-a-walkable-area/
- Selecting the main character: https://carenalgas.github.io/popochiu/how-to-develop-a-game/characters/select-the-main-character/
- Working with game state (Globals and per-object state): https://carenalgas.github.io/popochiu/the-engine-handbook/working-with-game-state/
- The Popochiu documentation home: https://carenalgas.github.io/popochiu/
- Godot documentation, Area2D (the base of clickable objects): https://docs.godotengine.org/en/stable/classes/class_area2d.html

Chapter 15: Inventory, Puzzles, and State

Learning objectives

By the end of this chapter you will be able to:

- Create inventory items with the Popochiu dock and use the inventory API (`add`, `add_as_active`, `remove`, `quantity_owned`, `I.is_item_in_inventory`) correctly.
- Explain how the inventory GUI turns a selected item into an "active item" and how `_on_item_used` receives it.
- Write lock-and-key logic for a prop or hotspot, including the "wrong item" and "already done" cases.
- Combine two items into a new one, handling both directions (Twine on Bone and Bone on Twine) without duplicating code.
- Explain why the Chapter 14 Drawer forgets its key (its variable has no type, so Popochiu does not keep it when the room reloads), and why puzzle state belongs in `Globals` or a per-object state script even though typed prop variables are remembered.
- Restore a room's look after loading a save, and test a puzzle chain with a checklist and a softlock analysis.

Key terms

- **Inventory item**: a Popochiu object (a `PopochiuInventoryItem`) that the player can carry; it has its own scene, script, and state script.
- **Inventory API**: the methods and properties on `I` and on each item (`I.Key.add()`, `I.Key.quantity_owned`, and so on).
- **Active item**: the inventory item the player has currently selected; it acts as the cursor and is "used" on the next thing clicked.
- `_on_item_used`: the virtual function that runs when an active item is used on a prop, hotspot, character, or another item.
- **Lock-and-key puzzle**: a puzzle where one thing (the lock) stays blocked until the player finds and uses another thing (the key).
- **State**: the facts the game must remember (the key was taken, the door is unlocked).
- **Globals**: the script `res://game/popochiu_globals.gd`, an autoload whose variables any script can read as `Globals.name`.
- **State script**: a per-object script (`room_bedroom_state.gd` and similar) holding variables that belong to one room, character, or item.
- **Softlock**: a game state from which the player can no longer finish the game, but the game does not end either.
- **Idempotent**: safe to run twice; running it a second time changes nothing more than the first time did.

15.1 Inventory items in Popochiu

In Chapter 14 you built rooms, a player, a Drawer prop, and a Door hotspot. The Drawer already did `I.Key.add()`, but we treated the Key as a given. In this section we look at what an inventory item actually is.

Creating an item

An inventory item is created from the **Main** tab of the Popochiu dock, in the **Inventory items** group, with the "+" button. You type a name. Use PascalCase with no spaces, because the name becomes the way you reach the item in code: an item named `Key` is reachable as `I.Key`.

For each item the dock writes a folder under `res://game/inventory_items/`. For the `Key` it contains:

```
res://game/inventory_items/key/
    inventory_item_key.tscn    the scene (a TextureRect with the picture)
    inventory_item_key.gd      the script you edit
    inventory_item_key.tres    the item's registry resource
    inventory_item_key_state.gd  the per-item state script
```

Two things to note. First, an inventory item extends `PopochiuInventoryItem`, which itself extends `TextureRect`, a Control. Inventory items are drawn by the interface, not by the room. Second, the dock also regenerates `res://game/autoloads/i.gd` so that `I.Key` exists. If `I.Key` is red in the script editor right after creating the item, save everything and wait a second; the autoload script is regenerated by the plugin.

> **Warning:** Never rename or move these files in the FileSystem dock. The state script, the autoload script, and `popochiu_data.cfg` all remember the paths. Use the Popochiu dock to create and delete items.

Every item needs a texture. Use a small placeholder picture, such as a 32 by 32 colored square, and keep the real art for later. In the item's scene, the picture is the `texture` property of the root `TextureRect`.

Starting in the inventory

In the dock's item row, a **star** icon marks an item that is already in the inventory when the game starts. Leave it off for puzzle items. A game that starts with a Key in the pocket has no key puzzle.

The inventory API

Everything you need is on the item itself, plus a few helpers on `I`. These names are exact for Popochiu 2.2.1.

Call	What it does
`I.Key.add()`	Puts one Key in the inventory. Full form: `add(quantity := 1, character := null)`.
`I.Key.add_as_active()`	Adds the item and also makes it the active item, so the cursor becomes the item at once.
`I.Key.remove()`	Takes the item out. Full form: `remove(quantity := 0, character := null)`.
`I.Key.replace(new_item)`	Swaps this item for another one.
`I.Key.quantity_owned`	How many the player owns now (an `int`).
`I.Key.ever_collected`	`true` once the item has been collected at least once in this run, even if it is gone now. It is **not stored in save files**, so after a load it is `false` again for any item the player no longer holds. Never use it for a puzzle fact; use a `Globals` flag.
`I.Key.max_quantity`	How many the player may hold at once (Inspector property, default `1`).
`I.is_item_in_inventory("Key")`	`true` if the item with that name (a `String`) is in the inventory.

Call	What it does
`I.get_item_quantity("Key")`	The quantity, by name.
`I.active`	The currently selected item, or `null`.
`I.deselect_active()`	Clears the selection.

There is no `I.Key.has()` and no `I.is_in_inventory()`. If you see them in an old tutorial, they do not exist in this version. There is also `I.Key.in_inventory`, which works but is deprecated since Popochiu 2.2 and is removed in 2.3, so use `I.Key.quantity_owned > 0` instead.

> **Note:** `add()` and `remove()` are coroutines: they finish when the inventory bar has played its animation. If you do not write `await`, the change to the inventory still happens at once (the item counts as owned or gone immediately) and the animation plays while your script carries on. This chapter deliberately leaves them without `await` where state must change before speech.
>
> **Note:** Each character has its own inventory since Popochiu 2.2. The calls above act on the player character by default. The optional `character` argument is how you would give an item to someone else. Weird Hotel only uses the player's inventory.

Two ways to ask the same question

These two lines mean the same thing:

```
if I.Key.quantity_owned > 0:
    print("I have the key")

if I.is_item_in_inventory("Key"):
    print("I have the key")
```

Use the first when you are in a script that knows `I.Key` exists (a typo fails at parse time, which is good). Use the second when the name comes from data, such as a dialog option, a string in a save file, or a function that takes an item name.

> **Try it:** Create an item named `Key` with a placeholder picture. In the Drawer script from Chapter 14, add `print(I.Key.quantity_owned)` before and after `I.Key.add()`. Run the game (F5), open the drawer, and check the Output panel. Expected result: it prints `0` and then `1`. Click the drawer again: the key must not duplicate. (If it does, that is the bug we fix in Section 15.4.)

15.2 The inventory GUI and using items

The GUI template you chose in Chapter 13 draws the inventory. In the **Simple Click** template the inventory is a foldable bar at the top of the screen. In the 9-Verb template it is a grid at the bottom. Either way, each carried item is shown with its picture, and the same ideas apply.

Selecting an item: the active item

Clicking an item in the inventory does not "use" it; it selects it. The selected item becomes the **active item**: the cursor shows the item's picture and `I.active` points to it. The next thing the player clicks receives the item.

In the **Simple Click** template (read from its source in Popochiu 2.2.1) the gestures are:

Gesture	Result
Left-click an item in the top bar	Selects it (it becomes the cursor)
Left-click a prop, hotspot or character while an item is selected	Calls that object's `_on_item_used(item)`
Left-click another inventory item while an item is selected	Calls the clicked item's `_on_item_used(item)`
Click (left or right) on empty room space while an item is selected	Puts the item away (deselects)
Right-click an object or item with nothing selected	Calls `_on_right_click()` (look)
Right-click an object while an item is selected	Nothing happens

There is no drag and drop in Simple Click: select, then click the target. After any `_on_item_used` finishes, Popochiu deselects the item by itself. Other templates differ, and their code lives in `res://game/gui/`.

What matters for scripting is one thing: when an active item is clicked on something, the engine calls `_on_item_used(item)` on the thing that was clicked, and passes the active item as the argument.

The handler

The signature is the same for props, hotspots, characters, and inventory items:

```
func _on_item_used(_item: PopochiuInventoryItem) -> void
```

The argument is the item that was used. You decide what happens by comparing it with the item objects on `I`:

```
func _on_item_used(item: PopochiuInventoryItem) -> void:
    if item == I.Key:
        print("the key was used on me")
    else:
        print("some other item was used on me")
```

Compare objects (`item == I.Key`), not names. This is the pattern the original course used and it is correct for Popochiu 2.2.1.

The default reply

The script the dock generates for a prop, hotspot, character or item already contains an `_on_item_used` that calls `E.command_fallback()`. In Simple Click that shows a message box such as "Can't USE Key with Door". If you delete the function, the base class version does nothing at all, and the click is silent. A good adventure never leaves the player with silence, and a message box is not in the character's voice. Replace the fallback with a line of your own:

```
func _on_item_used(_item: PopochiuInventoryItem) -> void:
    await C.player.say("That won't work.")
```

Always keep a final `else` that answers. A game full of silent hotspots feels broken.

Looking at an item

Inventory items also have `_on_right_click()`. A common use is the "look" description, spoken by the player:

```gdscript
# In inventory_item_key.gd. Keep the other functions your template wrote.
func _on_right_click() -> void:
    await C.player.say("It is warm. Keys should not be warm.")
```

Leave `_on_click()` as the template wrote it. It calls `E.command_fallback()`, and in Simple Click that is what selects the item; if you replace it, the item stops becoming active. (The generated `_on_right_click()` also calls the fallback, which shows "Nothing to see in this item"; that is why you override it.)

> **Try it:** Give the Key a right-click description. Run the game, take the key, and right-click it in the inventory. Expected result: the player speaks your line. Then select the key and click the Door; the Door's `_on_item_used` should run (you will write the correct version in the next section; for now it can be the Chapter 14 version).

15.3 Lock-and-key logic

Every adventure puzzle is a lock and a key, often in disguise. Chapter 12 introduced the idea on paper; here is the code-level version.

The three parts

1. **The key is acquired**: an item is added to the inventory, and a fact is recorded ("the key was taken").
2. **The key is applied**: `_on_item_used(item)` checks which item arrived and decides what to do.
3. **The world changes and stays changed**: another fact is recorded ("the door is unlocked"), and later code reads it.

The first and third parts are about state, and Section 15.4 spends its time on that. The second part is about the handler.

A puzzle is a small state machine

Before writing code, write down the facts and the transitions. For the Bedroom, with the facts `key_taken` and `door_unlocked`:

State	`key_taken`	`door_unlocked`	What the player can do
A (start)	false	false	Search the drawer. The door is locked.
B	true	false	Use the Key on the Door.
C	true	true	Walk through the Door.

The only legal moves are A to B (open the drawer) and B to C (use the key on the door). The state "false, true" (door open, key never taken) is not in the table. It should be impossible. The code should not rely on it being impossible, though: the Door will check only `door_unlocked`, never `key_taken`.

Writing the table first has a second benefit. You immediately see the questions the code must answer: what does the drawer say in state B or C? What does the Door say when the player uses the Key in state C? Each is a line of dialogue and a branch.

The Door handler, complete

Here is the Bedroom Door written for this chapter. It already uses `Globals`, which Section 15.4 creates. Read it for its shape now; do not paste it yet.

```
# Door (hotspot, Bedroom). Keep the header lines the dock generated.
@tool
extends PopochiuHotspot

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.door_unlocked:
        await R.goto_room("Hallway")
    else:
        await C.player.say("Locked. The keyhole is breathing.")

func _on_right_click() -> void:
    if Globals.door_unlocked:
        await C.player.say("An unlocked door. The keyhole looks relieved.")
    else:
        await C.player.say("A locked door. It would like a key.")

func _on_item_used(item: PopochiuInventoryItem) -> void:
    await C.player.walk_to_clicked()
    if item == I.Key:
        if Globals.door_unlocked:
            # Already open: do NOT consume the key.
            await C.player.say("It is already unlocked. The door looks smug.")
            return
        # Change state first, then talk.
        I.Key.remove()
        Globals.door_unlocked = true
        await C.player.say("It fits!")
    else:
        await C.player.say("That won't open it.")
```

Four design choices are hidden in that code, and each prevents a real bug:

1. `await R.goto_room("Hallway")`: `goto_room` is awaitable; the string is the room's script name exactly as shown in the dock. The original course omitted the `await`. It often works without, but with `await` the rest of the function waits for the room change.
2. **Using the key when already unlocked does not remove it.** Without this branch, a second Key (Chapter 16 gives you one as a reward) would be eaten for nothing.
3. **State changes before speech.** `I.Key.remove()` and `Globals.door_unlocked = true` run before the `say`. If the player saves, quits, or a cutscene interrupts during the line, the world is already consistent.
4. **The wrong-item reply exists.** Using the Bone on the Door gives a line instead of silence.

> **Why it works:** `await C.player.say(...)` pauses your function until the line has been spoken. Anything after it runs later. So any state change you put after a `say` is not safe if the speech can be interrupted or the game saved during it. Put changes first, talk second.
>
> **Try it:** On paper, add a fourth fact `door_open_seen` for "the player has been through the door once". Draw the new table. How many rows does it have? (Answer in Section 15.7: each new independent boolean doubles the number of possible combinations, which is why we keep the number of flags small.)

15.4 The state problem and its solution

At the end of Chapter 14 you saw the drawer "forget". With the type removed from its variable, take the key, leave the room through the Door, come back through the Hallway's BackDoor, and the drawer offers the key again. Let us understand exactly why before fixing it.

Why the Chapter 14 Drawer forgets

At the end of Lab 14 (step 37) the Drawer script contains this line, with no type:

```
var key_taken = false
```

A `var` at the top of a script belongs to the **node instance** made from that script. The Drawer node lives inside the Bedroom scene. When you leave the Bedroom, Popochiu unloads the room scene; when you return, it loads the scene again and builds new nodes, and each new node runs its initialisers. That is how Godot nodes work (Chapter 2: a scene is a recipe, and each instance is new).

Popochiu 2.2.1 softens this, as Section 14.9 showed. Before it unloads a room, it notes the **statically typed** simple variables (`bool`, `int`, `float`, `String`, `Array`) of each prop and hotspot, and it copies them back into the new nodes before `_on_room_entered()` runs. With `var key_taken: bool = false` the drawer would remember. But `var key_taken = false` has no type (it is a Variant), Popochiu skips it, and the new Drawer starts with `key_taken` back at `false`. One missing `: bool` is the whole bug.

You could fix it by putting the type back. This chapter does something better, because even a typed prop variable is a poor home for a puzzle fact:

- **Only the Drawer can read it.** The Door, the Frog's dialogue, a cutscene in the Hallway or the boss fight cannot ask the Drawer, because the Drawer exists only while the player is in the Bedroom. Facts such as "the door is unlocked" are needed in several rooms and by several objects.
- **It is easy to break by accident.** A missing type, a `Dictionary`, or a variable in the room's own script is not noted, and nothing warns you.
- **The facts are scattered.** With a flag in every prop, there is no single place to read what the game knows, which makes testing and debugging slow.
- **Globals is saved with the game, in one place.** Chapter 18's saving then needs almost no work, and you can open the save file and find every fact under one heading.

The lesson is general: **a fact the game must remember needs a home that lives longer than any one room, and that every script can read.**

Popochiu gives you two such homes.

Home 1: Globals

`Globals` is a script, `res://game/popochiu_globals.gd`, registered as an autoload named `Globals` (Chapter 5). It lives as long as the game runs. Open it from the FileSystem dock (it is not in the Popochiu dock). It starts with `extends Node`. You declare variables in it:

```
# res://game/popochiu_globals.gd
extends Node

# Bedroom puzzle
```

```
var key_taken: bool = false
var door_unlocked: bool = false
```

From any script you then read and write `Globals.key_taken`. Popochiu also saves and loads them automatically when the variables are "safe" types: `bool`, `int`, `float`, `String`, and arrays. A `Dictionary`, a `Vector2`, a `Color`, a custom Resource or a node need you to write `on_save()` and `on_load(data)` functions in Globals yourself. We stay with booleans.

> **Warning:** You must declare a variable in `popochiu_globals.gd` before using it. Writing `Globals.key_taken = true` when the variable is not declared is a runtime error, even though the file exists. The original course's global-variable code only worked after this declaration step.

Home 2: per-object state scripts

Rooms, characters, and inventory items each have a state script created by the dock (`room_bedroom_state.gd`, and so on). A variable declared there is saved with that object and reached as `R.Bedroom.state.some_var`, `C.Frog.state.some_var`, or `I.Key.state.some_var`. In the dock, each row has an icon that opens the state script.

For example, to count how many times the player searched the drawer, you could declare in `room_bedroom_state.gd`:

```
# In room_bedroom_state.gd, below the lines the dock generated.
var drawer_searches: int = 0
```

and use it from the Drawer:

```
R.Bedroom.state.drawer_searches += 1
```

(Props do not have state scripts. A prop's facts live in the state of its room, or in Globals.)

Which home to use

Question	Globals	Object state script
Who needs to read it?	Many rooms, characters, dialogs	Mostly one object
Example in Weird Hotel	`door_unlocked` (the Hallway and cutscenes ask about it)	`drawer_searches` (only the Bedroom cares)
Name clashes	You choose unique names yourself	Namespaced by object
Where do you find it?	One file, easy to list	Spread over many files
Typical beginner risk	Globals becomes a junk drawer of 60 variables	You forget which state file holds the fact

Rule used in this book: a fact that more than one room or dialog asks about goes in Globals. A fact that only one room cares about may go in that room's state. Because the book's naming contract fixes `key_taken` and `door_unlocked` as globals, and later chapters read them from several places, they live in Globals.

> **Note:** Keep every Globals variable listed in one place. Chapter 18 (Section 18.2) has a table of all of them. Chapter 15 adds four, Chapters 16 and 17 add more.

Refactoring the Drawer and the Door

We now move the two Bedroom facts into Globals. The refactor has four steps, and you should do them in this order.

1. Add the two declarations to `popochiu_globals.gd` (shown above) and save.
2. In the Drawer script, delete `var key_taken = false` (the untyped line Lab 14 left there) and replace each use with `Globals.key_taken`.
3. In the Door script, delete `var unlocked: bool = false` and replace each use with `Globals.door_unlocked`. Note the new name: the contract uses `door_unlocked`.
4. Run the game once, search the drawer, leave, come back.

The new Drawer script:

```
# Drawer (prop, Bedroom). Keep the header lines the dock generated.
@tool
extends PopochiuProp

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.key_taken:
        await C.player.say("Just old socks now.")
        return
    # State first, speech second.
    I.Key.add()
    Globals.key_taken = true
    await C.player.say("A key! It is warm. Why is it warm?")

func _on_right_click() -> void:
    await C.player.say("A drawer that hums faintly.")
```

The Door script is the one in Section 15.3.

> **Why it works:** The Drawer asks `Globals.key_taken`, a variable in a node that never gets rebuilt. The Drawer itself keeps no flag at all, so nothing depends on what Popochiu notes about it; the first thing it does on click is ask Globals, which remembers.

One more consequence: the Drawer decides by the **flag**, not by the inventory. Do not write `if I.is_item_in_inventory("Key")` to decide whether the drawer is empty. The player may have used the key already, so the inventory would be empty and the drawer would give a second key. Facts about the world are flags. Facts about what the player holds are inventory queries. Do not mix them.

> **Try it:** Run the game. Take the key. Go through the Door (use the key first). In the Hallway, click the `BackDoor` hotspot you built in Lab 14 (Part E) to return to the Bedroom; you arrive at the `FromHallway` marker. Click the drawer. Expected result: "Just old socks now." Before the refactor, with the untyped variable, the drawer would have offered the key again.

Restoring after save and load

Globals values are saved and loaded for you. A saved game from a state where `key_taken` is true will load with `Globals.key_taken` true. That solves the data. It does not always solve the **picture**. If something in a room changes how it looks (a pickup disappears, a door swings open), then after loading, the room must look right

again.

Popochiu does **not** save whether a prop is visible, clickable or where it stands. From the source: when you leave a room, Popochiu stores only the statically typed simple variables of its props and hotspots (Popochiu's own, such as `alpha` and `current_frame`, and any you declare with a type), the `enabled` flag of regions and walkable areas, and the position and visibility of *characters*. A prop that is disabled (not clickable) is skipped altogether. A prop you hid with `hide()` is shown again the next time the room is built. The robust habit is to write one function in the room script that **applies the state to the scene**, and call it from two places:

```
func _on_room_entered() -> void:
    _apply_state()

func _on_restore_from_savegame() -> void:
    _apply_state()

func _apply_state() -> void:
    # fragment: Section 15.5 shows the full Hallway version
    pass
```

`_on_room_entered` runs when the room is entered normally. `_on_room_transition_finished` does **not** run when a game is loaded, which is why the loaded case needs its own function, `_on_restore_from_savegame`. Because `_apply_state` only sets things to match the flags, running it twice does no harm. That is idempotence, the term you met at the start of the chapter.

15.5 Pickups in the Hallway: Bone and Twine

Now we give the player two more things to collect. They sit in the Hallway as props.

Creating the items and the props

1. In the dock's Main tab, create two inventory items: `Bone` and `Twine`, and a third one, `BoneFlute`. Give each a placeholder picture. Do not tick the star icon on any of them. We call it Twine, not String, because String is already the name of a GDScript type (an item named `String` would make Popochiu generate `I.String`, which clashes with it).
2. Open the Hallway room (the dock's "open in editor" icon on its row).
3. In the **Room** tab, in the Props group, click "+" and create a prop named `BonePile`. Leave "Will have interaction?" checked. Give it a small placeholder picture and move it to a spot on the floor.
4. Create a second prop named `TwineBall`, the same way.

We name the props `BonePile` and `TwineBall` rather than `Bone` and `Twine` so you never confuse "the object in the room" with "the item in your pocket". The prop is where the thing lies; the item is what you carry.

The global flags

Add two lines to `popochiu_globals.gd`:

```
# Hallway pickups
var bone_taken: bool = false
var twine_taken: bool = false
```

The prop scripts

Open `BonePile`'s script (the Room tab has a script icon on each row):

```
# BonePile (prop, Hallway). Keep the header lines the dock generated.
@tool
extends PopochiuProp

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.bone_taken:
        # A hidden prop should not be clickable, but this makes sure.
        return
    I.Bone.add()
    Globals.bone_taken = true
    disable()
    await C.player.say("A bone. It was just lying there, looking available.")

func _on_right_click() -> void:
    if Globals.bone_taken:
        return
    await C.player.say("A bone. Someone chewed it politely.")
```

The `TwineBall` script is the same shape:

```
# TwineBall (prop, Hallway). Keep the header lines the dock generated.
@tool
extends PopochiuProp

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.twine_taken:
        return
    I.Twine.add()
    Globals.twine_taken = true
    disable()
    await C.player.say("A ball of twine. It is slightly damp. I will not ask.")

func _on_right_click() -> void:
    if Globals.twine_taken:
        return
    await C.player.say("Twine. Good for tying things together.")
```

Three details to notice:

- `disable()` is a Popochiu method on every clickable: it hides the node (`visible = false`) and sets `clickable = false`. Popochiu also switches off a node's mouse input whenever it is hidden (the clickable script reacts to `visibility_changed`), so a hidden prop receives no clicks. `enable()` does the opposite. (Both are coroutines that take one frame; no `await` is needed here.)
- The guard `if Globals.bone_taken: return` is a second line of defence that costs two lines. With the source verified it should never fire, but a guard on a one-time reward is cheap insurance.
- The flag, the item, and the picture change together, before the `say`.
- Props also have an Inspector property **Link to item**. If you type `Bone` there, the prop disables itself when that item is in an inventory *or has ever been collected in this run*. It would work in one session, but `ever_collected` is not saved (Section 15.1), so after a load the pile could reappear. That is why this book uses its own flag and `_apply_state()`.

Keeping the Hallway right after leaving and loading

The `disable()` happens on a node that gets rebuilt when the room reloads. So the Hallway room script must disable the props again from the flags. Open the Hallway's room script (the room row's "open script" icon). Keep the header lines the dock generated; the functions below are what you add. The first `if` is the Chapter 14 code that places the player when arriving from the Bedroom:

```
# Hallway room script (room_hallway.gd).
func _on_room_entered() -> void:
    if C.player.last_room == "Bedroom":
        await C.player.teleport_to_marker("FromBedroom")
    _apply_state()

func _on_restore_from_savegame() -> void:
    _apply_state()

func _apply_state() -> void:
    if Globals.bone_taken:
        get_prop("BonePile").disable()
    if Globals.twine_taken:
        get_prop("TwineBall").disable()
```

Inside a room script, `get_prop(name)` is the room's own method; from anywhere else, `R.get_prop(name)` asks the current room. It finds the prop by its script name (the name you typed in the dock), which must match exactly, including capital letters. It is safe to call in `_on_room_entered`: Popochiu sets the room as current and the props are already in the tree before that hook runs. It prints an error and returns `null` if no prop has that name.

> **Try it:** Take the bone, leave the Hallway, and come back. Expected result: the bone pile is gone and the Bone is in your inventory. Without the `_apply_state` function, the pile would reappear while the Bone is still in your pocket, and you could farm Bones. That is a duplication bug, the cousin of the forgetful drawer.

15.6 Combining items

The player now carries a Bone and a Twine. Combining them gives a BoneFlute. The tricky part is that the player can do it in two ways: use the Twine on the Bone, or the Bone on the Twine.

Which item receives the call?

When an item is the active item and the player clicks another inventory item, the handler that runs is `_on_item_used(item)` **of the clicked item**, with the active item as the argument. So with the Twine active and the Bone clicked, Bone's script runs with `item == I.Twine`. With the Bone active and the Twine clicked, Twine's script runs with `item == I.Bone`. Some GUI templates let the player drag one item onto another, and the same logic applies to whichever item is the target.

You cannot know in advance which way a player will go, and you should not force one. The fix is to handle both directions. The naive way is to copy the recipe into both scripts. That works until you change the recipe in one place and forget the other. A better way is one shared function.

A shared recipe

Recipes are game logic, not state, but they need to be reachable from two scripts. The simplest honest place is `Globals`, which every script can reach. Add this function to `popochiu_globals.gd`, below the variables:

```
# Craft the flute from the Bone and the Twine.
# Called by both Bone and Twine when they are combined.
func make_flute() -> void:
    if I.Bone.quantity_owned < 1 or I.Twine.quantity_owned < 1:
        return
    # State first.
    I.Bone.remove()
    I.Twine.remove()
    I.deselect_active()
    I.BoneFlute.add()
    await C.player.say("I made a flute out of a bone. Normal.")
```

Then each item script is only a few lines. Open the Bone's item script:

```
# inventory_item_bone.gd. Keep the other functions your template wrote.
extends PopochiuInventoryItem

func _on_item_used(item: PopochiuInventoryItem) -> void:
    if item == I.Twine:
        await Globals.make_flute()
    else:
        await C.player.say("Those don't go together.")
```

and the Twine's:

```
# inventory_item_twine.gd. Keep the other functions your template wrote.
extends PopochiuInventoryItem

func _on_item_used(item: PopochiuInventoryItem) -> void:
    if item == I.Bone:
        await Globals.make_flute()
    else:
        await C.player.say("Those don't go together.")
```

> **Note:** Inventory item scripts do not start with `@tool` the way prop scripts do. Keep whatever header your file already has.

Why does `make_flute` begin with a guard? If both scripts fire in a strange order, or the recipe is called twice (say, by a double click), the second call finds nothing to combine and returns quietly. A recipe must be idempotent against accidents.

> **Why it works:** Both item scripts end in the same line, `await Globals.make_flute()`. There is exactly one place that knows the recipe. Changing it to need a second piece of twine means editing one function.

Making the new item interesting

The new item should answer a click like any other. Give `BoneFlute` a right-click description and a wrong-use reply:

```gdscript
# inventory_item_bone_flute.gd. Keep the other functions your template wrote.
extends PopochiuInventoryItem

func _on_right_click() -> void:
    await C.player.say("It plays one note. The note is sad.")

func _on_item_used(_item: PopochiuInventoryItem) -> void:
    await C.player.say("I should not do that to a flute.")
```

If you want the player to see the new item at once, replace `I.BoneFlute.add()` with `I.BoneFlute.add_as_active()`. The flute then becomes the cursor immediately. It is a nice touch for a reward item, but it can be confusing if the player was about to click something else, so use it deliberately. This book uses plain `add()`.

Counting copies

Because `quantity_owned` is a number, an item can be owned several times (a coin item, for example), but only if its **Max quantity** property in the Inspector is raised above the default of `1`. Every item in this book keeps the default. With it, calling `I.Key.add()` while a Key is already held does not give a second Key: Popochiu prints a warning ("It is already in ... inventory") and does nothing. `remove()` with no argument removes the whole stack. So the Chapter 16 reward is safe, but the player would silently get nothing if the Frog added a Key to someone already holding one; Chapter 16 therefore checks first.

15.7 Testing puzzles systematically

Puzzles fail in three ways: the player cannot do the intended thing, the player can do something unintended, and the player reaches a state where nothing works. A checklist catches the first two. A state analysis catches the third.

The puzzle test table

Write the table before you play. Each row is one action in one state, with the expected result. Tick it as you run the game. Here is the table for this chapter's chain.

#	Start state	Action	Expected result	Pass
1	New game	Click Door	"Locked." Stay in Bedroom	[ ]
2	New game	Use Bone on Door (cheat: see Lab)	"That won't open it."	[ ]
3	New game	Click Drawer	Key added; line spoken	[ ]
4	Key taken	Click Drawer again	"Just old socks"; Key count stays 1	[ ]
5	Key taken	Click Door	Still locked	[ ]
6	Key taken	Use Key on Door	"It fits!"; Key gone	[ ]
7	Door unlocked	Click Door	Hallway loads	[ ]
8	In Hallway	Click BonePile	Bone added; pile gone	[ ]
9	Bone taken	Leave and return	Pile still gone; one Bone	[ ]
10	In Hallway	Click TwineBall	Twine added; ball gone	[ ]
11	Both held	Use Twine on Bone	Flute added; both gone	[ ]

#	Start state	Action	Expected result	Pass
12	Both held	Use Bone on Twine	Same as 11	[ ]
13	Both held	Use Bone on Bone	"Those don't go together."	[ ]
14	Flute held	Use Flute on Door	A reply, no crash	[ ]
15	Mid-chain	Save, quit, load	All flags, items, pickups the same	[ ]

Three kinds of tests are in this table. Rows 1 to 7 are the **happy path**: the order the designer imagined. Rows 2, 4, 5, 13, 14 are **wrong-input tests**: every item on every target that is not its partner. Rows 9 and 15 are **persistence tests**: leave, return, save, load.

The wrong-input tests deserve a method. For N items and M targets there are N times M combinations. You do not need to test them all by hand, but you do need to look at every item you can carry against every target you can click once, because one silent target feels like a bug to the player. Spend ten minutes on it.

> **Warning:** The test table must be run on a **fresh game**, not on one where you have been poking for an hour. State from an earlier test hides bugs. Quit and restart between test groups.

Testing without replaying everything

Replaying from the start to test row 15 is slow. Add a temporary cheat while you build. For example, in the Bedroom room script:

```
# TEMPORARY test helper. Delete before shipping.
func _on_room_transition_finished() -> void:
    if OS.is_debug_build():
        I.Bone.add()
        I.Twine.add()
        Globals.bone_taken = true
        Globals.twine_taken = true
```

`OS.is_debug_build()` is `true` when you run from the editor. Remember to remove or disable the cheat. A helper left in a shipped game gives the player free items and hides the real puzzle path.

For save and load, use the gear (settings) button at the top right of the Simple Click GUI: its popup has **Save** and **Load** buttons. If you want a code route while testing, a temporary line `E.save_game(1, "test")` in a click handler and `E.load_game(1)` in another will do (Chapter 18 adds F9 and F10). Saves are JSON files in the user data folder (open it with **Project > Open User Data Folder**).

Softlock analysis

A **softlock** is a state from which the game cannot be finished and nothing tells the player. The worst bugs in adventure games are softlocks, because the player does not even know to report them. Find them by reasoning, before the player does.

A practical method with four questions, asked of every item and every flag:

1. **Can the player lose the item?** Does anything remove it (`remove()`) in a way that may not give the promised result? Every consumption must be paired with the thing it pays for, in the same function.
2. **Can the player make the required action impossible?** For example, if the Bone could be destroyed and there is only one, the flute could never be made.

3. **Is each one-time event guarded?** A one-time reward must set its flag so it cannot repeat, and the flag must be set whether or not the player sees the whole line.
4. **Is there an impossible combination of flags that the code would misread?** (For example, "door unlocked" with "key never taken".)

Apply them to our chain:

Item or flag	Can it be lost?	Result
Key	Used on Door: consumed, and the door is unlocked in the same function	Safe
Key (used twice)	Second use finds the door unlocked and does not consume	Safe
Bone / Twine	Only removed by `make_flute`, which gives the Flute in the same function	Safe
Bone / Twine pickups	Single copy each, guarded by `bone_taken` / `twine_taken`	Safe (cannot be taken twice; cannot vanish)
Flute	Chapter 16: removed by the Frog only when the reward is delivered	Safe, if written as in Chapter 16

One danger remains that no table can fully catch: **a dead end made by speech order.** If the pickup were written as `await say(...)` followed by `add()`, a player who skipped a cutscene or quit during the line could end up with a hidden prop, a set flag, and no item. This is why every handler in this chapter changes state before speaking.

The flag count problem

Each independent boolean flag doubles the possible states. Four flags give 16 combinations; ten give 1,024. You cannot test all of them. Two habits keep the number manageable:

- **Derive, do not duplicate.** If "player has a flute" can be answered by `I.is_item_in_inventory("BoneFlute")`, do not add a flag `has_flute` that must always agree.
- **Make flags monotone.** A flag that only goes from false to true (never back) has far fewer reachable combinations than one that flips.

> **Try it:** Take your puzzle test table and add one row for each wrong item on each target you have so far (Key, Bone, Twine, and BoneFlute against Door, Drawer, BonePile, and TwineBall). That is 16 rows. Run through them quickly. Expected result: you find at least one target that stays silent. Give it a reply.

Lab 15: The key, door, and flute chain

Time: 60 to 90 minutes. **You need:** the `weird-hotel` project at the end of Chapter 14, with Bedroom, Hallway, Player, the Drawer prop, the Door hotspot and the Hallway's BackDoor working with local variables (the Drawer's `key_taken` left untyped, as at the end of Lab 14).

Part A: Move the state into Globals

1. Open `res://game/popochiu_globals.gd` in the FileSystem dock. Under `extends Node`, add:

```
extends Node

# Bedroom puzzle
var key_taken: bool = false
var door_unlocked: bool = false

# Hallway pickups
var bone_taken: bool = false
var twine_taken: bool = false

# Craft the flute from the Bone and the Twine.
func make_flute() -> void:
    if I.Bone.quantity_owned < 1 or I.Twine.quantity_owned < 1:
        return
    I.Bone.remove()
    I.Twine.remove()
    I.deselect_active()
    I.BoneFlute.add()
    await C.player.say("I made a flute out of a bone. Normal.")
```

2. Replace the Drawer script with the version in Section 15.4.
3. Replace the Door script with the version in Section 15.3.
4. Run the game. Do rows 1 to 7 of the test table. Then, in the Hallway, click the `BackDoor` to return to the Bedroom and click the drawer. Expected: "Just old socks."

Part B: Items

5. In the Main tab, create inventory items `Bone`, `Twine`, and `BoneFlute` (you already have `Key`). Give each a placeholder picture.
6. Add the right-click descriptions to `Key` and `BoneFlute` as in Sections 15.2 and 15.6.
7. Write the Bone and Twine item scripts exactly as in Section 15.6.

Part C: Hallway pickups

8. Open the Hallway scene. In the Room tab create props `BonePile` and `TwineBall` with placeholder pictures. Place them on the floor.
9. Write both prop scripts from Section 15.5.
10. Add `_on_restore_from_savegame` and `_apply_state` to the Hallway's room script, and make `_on_room_entered` match Section 15.5 (the Chapter 14 `teleport_to_marker("FromBedroom")` check stays, followed by `_apply_state()`).

Part D: Test

11. Add the temporary cheat from Section 15.7 to the Bedroom room script, but only if you want to skip the key step. Remember to remove it.
12. Run the full test table from 15.7, with a fresh start for rows 1 to 7, 8 to 14, and 15.
13. Do the softlock questions on paper for each of your five items and flags. Write down one sentence for each.
14. Commit: `git add -A`, `git commit -m "Chapter 15: inventory, globals, flute chain"`.

Expected result: from a fresh game you can take the key, unlock the door, enter the Hallway, take the bone and the twine, combine them in either direction into a BoneFlute, and every step survives leaving a room, saving, and loading. Using any item on any wrong target gives a spoken reply, never silence. The drawer never gives a second key, and the piles never reappear.

Chapter summary

- Create inventory items in the dock's Main tab. Reach them as `I.Name`. Add and remove with `add()`, `add_as_active()`, and `remove()`.
- Ask "do I have it?" with `I.Name.quantity_owned > 0` or `I.is_item_in_inventory("Name")`. There is no `has()`. `in_inventory` is deprecated.
- Selecting an item in the GUI makes it the active item. Clicking a target calls the target's `_on_item_used(item)`, which compares `item` with `I.Name`.
- Every `_on_item_used` should end with a spoken `else`. Silence feels like a bug.
- A node is rebuilt when its room reloads. Popochiu copies back the typed simple variables of props and hotspots, but an untyped variable (the Chapter 14 Drawer's), a complex type or a room-script variable starts over, and no other room can read any of them. Anything the game must remember lives in `Globals` (declare the variable first) or in a per-object state script.
- Use Globals for facts many scripts ask about, and a room's state for facts only that room needs.
- Change state first, then speak. Facts about the world are flags; facts about the player's hands are inventory queries.
- Put a `_apply_state()` function in the room script and call it from `_on_room_entered` and `_on_restore_from_savegame`.
- Handle item combination in both directions by calling one shared function.
- Test with a table: happy path, wrong inputs, and persistence. Hunt softlocks by asking whether each item or flag can be lost, made impossible, repeated, or misread.

Review questions

1. Which method adds an inventory item and also makes it the cursor at once?
2. True or false: `I.Key.has()` is the correct way to test whether the player holds the Key in Popochiu 2.2.1.
3. Name two ways to check that the player owns the Key, and say when you would prefer the second.
4. When the player selects an item in the inventory and clicks a prop, which function runs, and where?
5. Why does the Chapter 14 Drawer (as Lab 14 left it) offer the key again after you leave and return to the Bedroom, and why is `Globals` a better home for `key_taken` than a typed variable in the Drawer?
6. You write `Globals.stone_found = true` but forgot to edit `popochiu_globals.gd`. What happens?
7. Give one fact that belongs in Globals and one that belongs in a room's state script, and explain the difference.
8. Why must a recipe that combines two items be reachable from both items' scripts?
9. In the Door's `_on_item_used`, why does the Key branch check `Globals.door_unlocked` before calling `I.Key.remove()`?
10. Name the two room functions you call `_apply_state()` from, and say why the second one is needed.

Exercises

1. **Practice (easy).** Give the Door a different reply for each wrong item: a Bone, a Twine, and a BoneFlute. Use `if` and `elif` on `item`. Test each.
2. **Practice (easy).** Add a right-click description to the Bone, the Twine, and the Key. Write them in the voice of a hotel guest who is slightly too calm.
3. **Apply (medium).** Add a counter `drawer_searches` to the Bedroom's state script and increase it each time the player searches the empty drawer. After the third search, have the player say something strange once. Make sure the count survives leaving and re-entering the room, and explain why the

room's state script is the right home for it.

4. **Apply (medium).** Design a second recipe on paper: the `Bug` item plus the `BoneFlute` gives a new item of your choice. Then write the item scripts (both directions) and a shared function `make_bug_flute()` in Globals. Decide whether the old items should be removed, and run the four softlock questions on your recipe.
5. **Challenge (hard).** Draw the full state table for all six flags and facts (`key_taken`, `door_unlocked`, `bone_taken`, `twine_taken`, plus "has Flute" and "has Key") and mark every state that cannot be reached by legal play. Then add a debug check in Globals that prints a warning if an impossible combination ever appears, and trigger it deliberately with a temporary cheat.

Common mistakes

- **Symptom:** the drawer gives the key again after leaving the room. **Cause:** the flag is a local `var` in the Drawer without a type (`var key_taken = false`), which Popochiu does not keep. **Fix:** declare it in `popochiu_globals.gd` and use `Globals.key_taken`.
- **Symptom:** `Invalid access to property 'key_taken' on 'Globals'`, or a similar error. **Cause:** the variable is not declared in `popochiu_globals.gd`, or is spelled differently. **Fix:** add `var key_taken: bool = false` and check the spelling at every use.
- **Symptom:** `I.Key.has()` or `I.Key.is_in_inventory()` reports an invalid call. **Cause:** these methods do not exist in Popochiu 2.2.1. **Fix:** use `I.Key.quantity_owned > 0` or `I.is_item_in_inventory("Key")`.
- **Symptom:** using the Twine on the Bone works, but using the Bone on the Twine says "Those don't go together". **Cause:** only one of the two item scripts handles the recipe. **Fix:** call the shared `Globals.make_flute()` from both.
- **Symptom:** a taken pickup is back in the room after you return, and you can take it again. **Cause:** the room does not re-apply its state on entering. **Fix:** write `_apply_state()` and call it from `_on_room_entered` and `_on_restore_from_savegame`.
- **Symptom:** after loading a save, a pickup that should be gone is visible. **Cause:** the room only restored the picture in `_on_room_transition_finished`, which does not run on load. **Fix:** also call `_apply_state()` in `_on_restore_from_savegame`.

Further reading

- Popochiu documentation home: https://carenalgas.github.io/popochiu/
- Popochiu, working with game state (Globals and state scripts): https://carenalgas.github.io/popochiu/the-engine-handbook/working-with-game-state/
- Popochiu, creating the first room: https://carenalgas.github.io/popochiu/how-to-develop-a-game/rooms/create-the-first-room/
- Godot, GDScript basics (await, coroutines, static typing): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

Chapter 16: Dialogue Trees and Cutscenes

Learning objectives

By the end of this chapter you will be able to:

- Design a dialog tree on paper using hubs, one-shot options, gated options, and endings.
- Create a dialog resource and a speaking character with the Popochiu dock, and start the dialog from a click.
- Script a dialog with `_option_selected`, `turn_on_options`, `turn_off_options`, `_show_options`, and `stop`, or with per-option handlers.
- Drive characters with `say`, `face_*`, and `walk_to`, and know when an action needs `await` and when it needs a `queue_` twin.
- Build a cutscene with `E.cutscene` and `E.queue`, add camera effects with `E.camera`, and design it so skipping it cannot break the game.
- Use Globals flags to make conversations and cutscenes remember what happened, and write dialogue that sounds like a person.

Key terms

- **Dialog tree**: a menu of things the player can say, where each choice can change the menu that comes next.
- **Hub**: a node in the tree that the conversation keeps returning to; the main menu of a conversation.
- **One-shot option**: an option that can be chosen once and then disappears.
- **Gated option**: an option that stays hidden until a condition is met (the player knows something, or holds an item).
- **Dialog option**: one entry of a dialog, with an `id`, the `text` the player sees, and flags such as `visible` and `disabled`.
- **Cutscene**: a scripted sequence in which the player cannot act; the game tells a piece of story.
- **Queue**: an ordered list of actions run one after another by `E.queue` or `E.cutscene`.
- `queue_` **twin**: the version of a character or camera action that returns a `Callable` for use inside a queue, instead of running at once.
- **Beat**: one small unit of a scene: a line, a movement, a pause, a camera move.
- **Voice**: the consistent way a character speaks: word choice, rhythm, and what they care about.
- **Flag**: a `bool` in `Globals` that records that something happened.

16.1 Dialog trees and their design

Dialogue in an adventure game is not a transcript. It is a small interactive program. The player chooses a line, the other character replies, and the set of available choices may change. Popochiu shows the menu and speaks the lines; your script decides what each choice does.

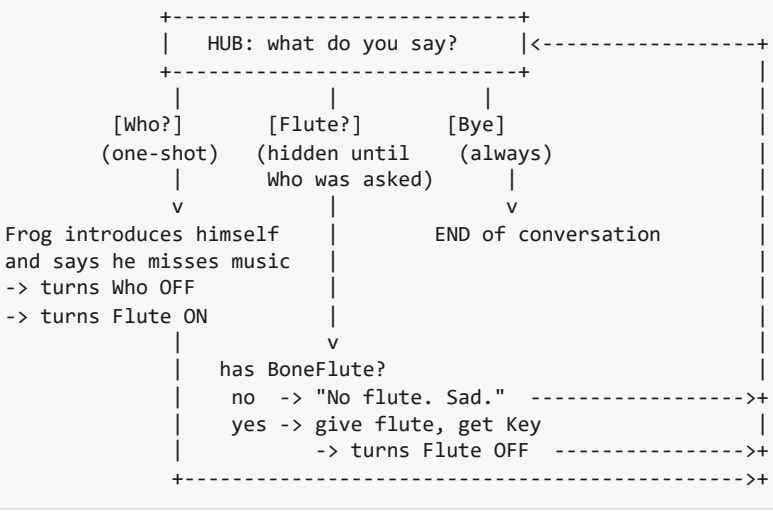
The four building blocks

Almost every adventure conversation is built from four kinds of nodes.

1. **Hub.** The menu the conversation keeps coming back to. After the Frog answers one question, the menu appears again with the remaining questions. A hub is what makes a dialog feel like a conversation rather than a monologue.
2. **One-shot option.** A question that is only worth asking once. After "Who are you?", asking it again would be silly, so the option turns itself off.
3. **Gated option.** An option that is hidden at first and appears when something changes. A "Do you need a flute?" option should not exist until the player has heard the Frog miss music. Gating keeps the menu short and lets the conversation reveal the plot.
4. **Ending.** A way out. The player must always have an exit option, and the conversation must end cleanly, returning control to the game.

A tree for the Frog

Here is the Frog conversation of Weird Hotel as a diagram. Arrows are what each choice does.



Notice three properties of this tree. Every non-ending branch returns to the hub. The only exit is "Bye". And the "Flute" option is both gated (it appears after "Who") and a one-shot (it disappears when the reward has been given). Real dialog trees are built from the same few parts, just more of them.

Design rules for trees

- **Keep the hub small.** Three to five visible options is comfortable. More than six makes the player read a list instead of play a game.
- **Always provide an exit.** A hub without a "Bye" traps the player.
- **Never let the menu become empty.** If every option is a one-shot and all have been used, the dialog must either end by itself or still show an exit. Make "Bye" always available.
- **Gate by what the player knows.** An option the player could not reasonably think of yet should not be shown.
- **Make progress visible.** When an option unlocks something, the answer should hint at it ("I miss music"), so the player is not forced to try every option at random.

- **A dialog is not the only place for plot.** Do not put clues that are required to finish the game only inside optional options without a repeatable way to hear them again.

> **Try it:** On paper, design a tree for a second character, the Hotel Cat, with a hub and four options: "Who are you?" (one-shot), "Seen a bone?" (gated, appears after you have met the Cat), "Can I pet you?" (repeatable, with three different replies), and "Bye". Draw the arrows and mark the one-shot and gated options. Check the rule "the menu is never empty".

16.2 Writing good dialogue and voice

Programmers often write dialogue last, quickly, and it shows. A game with clever puzzles and flat lines feels like a worksheet. Writing dialogue is a craft you can learn with a few habits.

Principles

1. **Short lines.** A speech bubble that fills half the screen will be skipped. Aim for one or two short sentences per `say`. If a thought needs more, split it into two `say` calls with a beat between.
2. **One job per line.** A line can do one of these: give a fact, show character, ask something, or make a joke. A line that tries all four is mush.
3. **Voice means consistent choices.** Decide what each character cares about and how they speak, and keep to it. The Frog is a landlord: dry, formal, bored, obsessed with rent and carpets. He never says "awesome". The player character is calm in the face of absurdity, which is where much of the humor comes from.
4. **Subtext over exposition.** Compare "As you know, I am the landlord of this hotel, which has strange rules" with "Mind the carpet. It bites." The second tells you the world without announcing it.
5. **Hints belong in character.** If a puzzle needs a clue, let a character give it in their own way. "I miss music" is a hint. "You need to give me the flute" is a walkthrough.
6. **Menu text is what the player means, not what they say.** The option text is shown in the menu, then spoken by the player with `D.say_selected()`. Make it short, and make it read like the player talking: "Who are you?" works in both places.
7. **Vary repeats.** If a click can be repeated, give the player several lines, or the last line should acknowledge the repetition.

Before and after

Weak	Better	Why
"Hello, I am the Frog. I am the landlord of this hotel."	"I am the landlord. Mind the carpet. It bites."	The second shows the world.
"You have to bring me a flute to get the key."	"I used to have a band. Now I have rent."	A hint in character, not an order.
"I do not have that item."	"No flute. Sad."	The reply carries the Frog's mood.
"OK. Thank you for the flute. Here is the key."	"Mm. Out of tune. Perfect. Take this."	Ends with a joke and the reward.

Reading aloud

Read your lines aloud. Your ear catches stilted phrases that your eye misses. If you stumble, the player will too. If you cannot tell which character said a line without the name, the voices are not distinct enough.

Varying a repeated line

Godot arrays have `pick_random()`, which returns a random element. It is a cheap way to avoid the same bubble every time:

```
await C.Frog.say(["Rent.", "Still rent.", "I said rent."].pick_random())
```

Use it sparingly: variety helps for throwaway lines, but story-critical lines should be fixed and deliberate.

> **Try it:** Take the Frog's four "Better" lines above and rewrite them in the voice of a different landlord, a pirate or a robot. Read both versions aloud. Which words changed, and which stayed? The ones that stayed are the content; the ones that changed are the voice.

16.3 Dialogs and characters in Popochiu

The Frog character

Create the NPC first. In the dock's **Main** tab, in the **Characters** group, click "+" and name it `Frog`. Do **not** set it as the player character (that is the "Set as Player Character" option in the row's dropdown; `Player` already has it). Give the Frog a placeholder picture, such as a green square.

The dock writes `res://game/characters/frog/character_frog.tscn`, `.gd`, `.tres`, and `character_frog_state.gd`. As with items, it also regenerates the `C` autoload so `C.Frog` exists.

Then put the Frog in the Hallway:

1. Open the Hallway scene.
2. In the **Room** tab, find the group **Characters in room** and use the add button ("Add character to room").
3. Choose `Frog`.
4. Move the new `Frog` node to where the landlord should sit. The Inspector of the character has a **Text Color** property (`text_color`) for its speech bubbles. A different color per speaker helps the player.

A character scene contains a `Sprite2D` (set its **Texture** for the placeholder picture) and an empty `AnimationPlayer`. With no animations, Popochiu's idle, walk and talk calls find nothing to play and quietly do nothing, so a still picture works.

The dialog resource

Dialog trees in Popochiu are **Resources**, not scenes. Create one with the "+" in the **Dialog trees** group of the Main tab, and name it `ChatWithFrog`. The dock creates:

```
res://game/dialogs/chat_with_frog/
    dialog_chat_with_frog.gd     the script you edit
```

```
    dialog_chat_with_frog.tres   the resource that holds the options
```

There are two ways to define the options.

In the Inspector. Select `dialog_chat_with_frog.tres` in the FileSystem dock. In the Inspector, find the **Options** array, click **Add Element**, and fill in the fields for each option:

Field	Meaning
`id`	The name your script uses (`Who`). No spaces.
`text`	What the menu shows and the player speaks (`Who are you?`).
`visible`	Uncheck to start hidden (a gated option).
`disabled`	A hard off switch; `turn_on_options` has no effect on a disabled option.
`always_on`	Only affects looks: an option the player has already chosen is normally drawn in a different style, and `always_on` keeps it drawn as "not yet chosen". It does not make an option stay available.

In code. The same options can be built in the dialog script with a function the engine calls:

```
func _on_build_options(_existing_options: Array[PopochiuDialogOption]) -> Array[PopochiuDialogOption]:
    return [
        create_option("Who", {text = "Who are you?"}),
        create_option("Flute", {text = "Do you need a flute?", visible = false}),
        create_option("Bye", {text = "Goodbye."}),
    ]
```

This chapter uses the Inspector, because a beginner can see every option at a glance. The code form is useful for dialogs built from data.

Create these three options for `ChatWithFrog`:

id	text	visible
`Who`	Who are you?	yes
`Flute`	Do you need a flute?	no
`Bye`	Goodbye.	yes

Starting the dialog

The conversation starts from code, normally from the NPC's click handler. Open the Frog's script:

```
# Frog (character, Hallway). Keep the header lines the dock generated.
@tool
extends PopochiuCharacter

func _on_click() -> void:
    await C.player.walk_to_clicked()
    await C.player.face_clicked()
    await D.ChatWithFrog.start()

func _on_right_click() -> void:
    await C.player.say("A frog in a cardigan. He does not blink at the right times.")
```

`D.ChatWithFrog.start()` is awaitable: the line after it runs when the dialog has ended. That is useful for things like "the Frog says goodbye, then the player walks off".

The dialog script and the engine's loop

When `start()` is called, Popochiu blocks the interface, builds the options (once), calls the dialog's `_on_start()` (signature `func _on_start() -> void`), and shows the menu. When the player picks an option, the engine calls `_option_selected(opt)`. Your function does something with `opt.id`, and then calls `_show_options()` to show the menu again. The cycle is:

```
start() -> _on_start() -> menu shown
   -> player picks -> _option_selected(opt) -> your code -> _show_options() -> menu shown
   -> ... -> stop() -> dialog ends -> start() returns
```

The dialog's script extends `PopochiuDialog` (a Resource, so `get_tree()` does not exist in it; use `E.get_tree()` when you need one). Keep the first lines the dock wrote (`@tool` and `extends PopochiuDialog`) and write the functions below them.

Controlling options

You change the menu with these calls, available inside the dialog script. The first two take **arrays of ids**, even for one id.

Call	Effect
`turn_off_options(["Who"])`	Hides the option (until turned on).
`turn_on_options(["Flute"])`	Shows the option (unless it is `disabled`).
`turn_off_forever_options(["Who"])`	Hides it permanently.
`opt.turn_off()`, `opt.turn_on()`, `opt.turn_off_forever()`	The same, called on the option object that arrived.
`get_option("Who")`	Returns the option object by id.
`D.say_selected()`	Makes the player speak the selected option's text.
`stop()`	Ends the conversation.
`_show_options()`	Shows the menu again (does nothing if the dialog was stopped).

The documented style is `turn_off_options(["Id"])`, and this book uses it.

The shape of `_option_selected`

```
@tool
extends PopochiuDialog

func _option_selected(opt: PopochiuDialogOption) -> void:
    match opt.id:
        "Who":
            await D.say_selected()
            await C.Frog.say("I am the landlord. Mind the carpet. It bites.")
        "Bye":
            await D.say_selected()
            stop()
        _:
```

```
            stop()
    _show_options()
```

Three habits are in this skeleton. The `_:` branch (the "anything else" case of `match`) calls `stop()` so an id you forgot to handle ends the conversation instead of freezing the game. Each branch speaks the player's choice first with `D.say_selected()`. And `_show_options()` is the very last line, so the menu reappears after any branch that did not `stop()`.

The other style: per-option handlers

Instead of one `match`, you can define one function per option, named `_on_option_` plus the id in snake case. The id `Who` becomes `_on_option_who`, and `DogJoke` would become `_on_option_dog_joke`:

```
func _on_option_who(_opt: PopochiuDialogOption) -> void:
    await D.say_selected()
    await C.Frog.say("I am the landlord. Mind the carpet. It bites.")
```

The engine awaits the function and then shows the menu again by itself, so you must not call `_show_options()` here. If you want to end the dialog, call `stop()`.

> **Warning:** Choose one style per dialog. The engine calls `_option_selected(opt)` **without waiting for it**, and then, if a matching `_on_option_<id>` function exists, calls that one and waits for it. If you define both, the two bodies run at the same time, and if both show the menu, it appears twice. The `match` style keeps all branches in one function you can read at a glance, so the Frog uses it.
>
> **Try it:** Create the Frog, the dialog, and the three options. Write the click handler and the skeleton above with only the `Who` and `Bye` branches. Run the game, walk to the Hallway, click the Frog. Expected result: the player walks over, a menu with "Who are you?" and "Goodbye." appears (the Flute option is hidden), "Who" gives the Frog's line and returns to the menu, and "Goodbye." ends the conversation and gives control back.

16.4 Scripting the Frog conversation

Now we write the real tree with flags and the reward. The design has four flags in Globals. Open `popochiu_globals.gd` and add:

```
# Hallway story (Chapter 16)
var hallway_intro_seen: bool = false   # the entry cutscene has played
var frog_met: bool = false             # the player has spoken to the Frog before
var frog_who_asked: bool = false       # "Who are you?" has been asked
var flute_given: bool = false          # the flute has been traded for the key
```

Each flag has one meaning. That discipline is what keeps the combinations testable.

Derive the menu from the flags

A tempting design is to rely on `turn_on_options` and `turn_off_options` alone. Popochiu does keep each dialog's options (visible, disabled, used) and writes them into save files, so this mostly works. But the dialog is a Resource that Godot loads once and keeps for the whole run, so its option states carry on between games

started without quitting, and a menu that is its own memory is a second copy of facts you already store as flags. The robust design makes the **flags the truth** and the menu a view of the flags. At the start of every conversation, the dialog sets its own options from the flags. The `_on_start()` function is made for this.

```
func _on_start() -> void:
    # The menu is a view of the flags, not a memory of its own.
    if Globals.frog_who_asked:
        turn_off_options(["Who"])
        turn_on_options(["Flute"])
    if Globals.flute_given:
        turn_off_options(["Flute"])
    # The function must await something; this is the line the template contains.
    await E.get_tree().process_frame
```

The dock's template for `_on_start()` awaits one frame in exactly this way, with a comment saying the method must await something to work. Keep that line and add the `if` statements above it. Use `_on_start()` for setup only; do not start another dialog in it.

The give-flute branch and its edge case

The Frog accepts the flute and gives a key. Before writing it, we must think about state, because the Key is also the drawer's item.

Recall the flags: `key_taken` records that the drawer was emptied, and `door_unlocked` records that the Bedroom door is open. The player can only be in the Hallway if `door_unlocked` is true, which means the first Key was already consumed. So when the Frog hands over a Key, there is no locked door left for it. This is the edge case: a reward that duplicates a puzzle item the player has already used up.

There are three clean ways to handle it.

Option	Idea	Problem
A. Reward is a second Key, and the Door tolerates it	Frog gives `Key`. Using it on an unlocked Door gives a line and does not consume it.	The reward needs a purpose later.
B. Reward is a different item	Invent `FrogKey` and so on.	Breaks the naming contract; more items to maintain.
C. Reward is a line only	Frog says "have a key" and nothing is added.	Feels cheap for the work of building the flute.

We choose **option A**. The reward is the `Key` item, described as the Frog's spare key, and the pieces that make it safe are:

1. **The flags describe the world, not the inventory.** `key_taken` means "the drawer has been emptied". The Frog's reward does not touch it, so the drawer still says "old socks", and does not offer a third key. `door_unlocked` is not touched either.
2. **The Door tolerates an extra Key.** In Chapter 15 the Door's `_on_item_used` checks `Globals.door_unlocked` first, replies "It is already unlocked", and returns without calling `I.Key.remove()`. Using the spare key on the open door does not eat it, and the player is not stuck holding something that triggers an error.
3. **The reward is guarded so it cannot be doubled.** The Frog checks `I.is_item_in_inventory("Key")` before giving a Key. If the player somehow already holds one (for example, from a debug cheat), the Frog does not take the flute and says so, so the flute is never consumed without a reward.

4. **The flute is only removed together with the reward.** Both happen in the same function, flag first, speech afterwards.

What the spare key *opens* is a design question for you. In this chapter it is a keepsake that proves the trade; a front door in the Hallway would be a natural lock for it, and Exercise 4 asks you to add that. The game as built in Chapters 14 to 18 ends after the boss fight, so the spare Key is never required to finish it.

> **Why it works:** The design goal is that no combination of earlier actions can make the reward harmful. The reward neither changes the drawer's flag nor unlocks the door a second time, and the Door tolerates a Key it does not need. This is the softlock analysis of Chapter 15 applied to a new action.

The complete dialog script

```
# ChatWithFrog (dialog). Keep the header lines the dock generated.
@tool
extends PopochiuDialog

func _on_start() -> void:
    if Globals.frog_who_asked:
        turn_off_options(["Who"])
        turn_on_options(["Flute"])
    if Globals.flute_given:
        turn_off_options(["Flute"])
    await E.get_tree().process_frame

func _option_selected(opt: PopochiuDialogOption) -> void:
    match opt.id:
        "Who":
            await D.say_selected()
            # State first, then talk.
            Globals.frog_who_asked = true
            turn_off_options(["Who"])
            turn_on_options(["Flute"])
            await C.Frog.say("I am the landlord. Mind the carpet. It bites.")
            await C.Frog.say("I used to have a band. Now I have rent.")
        "Flute":
            await D.say_selected()
            await _flute_talk()
            if Globals.flute_given:
                stop()
        "Bye":
            await D.say_selected()
            await C.Frog.say("Rent is due. Always.")
            stop()
        _:
            stop()
    _show_options()

func _flute_talk() -> void:
    if not I.is_item_in_inventory("BoneFlute"):
        await C.Frog.say("No flute. Sad.")
        return
    if I.is_item_in_inventory("Key"):
        # Defensive: never take the flute unless the reward can be given.
        await C.Frog.say("Your hands are full of key. Come back with room.")
        return
    # Record it, remove the flute, give the key, all together, before any speech.
    # (remove() and add() change the inventory at once; only their animation is awaited.)
    Globals.flute_given = true
    turn_off_options(["Flute"])
    I.BoneFlute.remove()
    I.Key.add()
    await C.Frog.say("A bone flute. Hm.")
```

```
    await C.player.say("It plays one sad note.")
    await C.Frog.say("Perfect. Take my spare key.")
```

Read it with the tree from Section 16.1 beside it. Every arrow in the diagram has a line of code: "Who" turns itself off and the Flute on; "Flute" without the item returns to the hub; "Flute" with the item gives the reward and then ends the conversation; "Bye" ends it.

A few points deserve explanation.

- **The `stop()` inside "Flute"** ends the conversation after the trade, which feels natural. The `_show_options()` at the bottom is then a no-op. If you prefer the Frog to stay in the hub after the trade, remove the `if Globals.flute_given: stop()` lines; the Flute option is already off.
- **`_flute_talk` is a helper.** It is an ordinary function in the dialog script. Because it contains `await`, the call is `await _flute_talk()`.
- **The Frog speaks about the item before the player's reply.** The order of `say` calls is the order of speech. Reorder them freely; `say` calls always run one at a time because we `await` each one.
- **The state is changed before any speech in the trade.** Even if the player skips with ESC, the trade is complete.

The Frog's click, with flags

The first time the player clicks the Frog and the later times should differ. Update the Frog's script:

```
# Frog (character, Hallway). Keep the header lines the dock generated.
@tool
extends PopochiuCharacter

func _on_click() -> void:
    await C.player.walk_to_clicked()
    await C.player.face_clicked()
    if Globals.flute_given:
        await C.Frog.say("The rent is paid in music. You paid.")
    elif Globals.frog_met:
        await C.Frog.say("You again. Rent.")
    else:
        Globals.frog_met = true
        await C.Frog.say("Yes? I am busy. Being a frog.")
    await D.ChatWithFrog.start()

func _on_right_click() -> void:
    await C.player.say("A frog in a cardigan. He does not blink at the right times.")

# An item used on the Frog. We compare script names (strings) so this script
# also works before the Bug item exists (Chapter 17 creates it).
func _on_item_used(item: PopochiuInventoryItem) -> void:
    await C.player.walk_to_clicked()
    await C.player.face_clicked()
    match item.script_name:
        "Bug":
            # The hint promised in Chapter 12: the Bug is kept, the Frog points at the flute.
            await C.Frog.say("A bug. Nice. But I want music, not snacks.")
        "BoneFlute":
            await C.Frog.say("Ask me properly.")
        _:
            await C.Frog.say("I do not accept gifts. Except flutes.")
```

Note how one `if` ladder uses three flags to choose among three greetings. This is the simplest form of **branching with flags**: read the flags at the top of a handler, pick a branch, change the flags in that branch, and carry on.

16.5 Character actions, sequences, and the queue

Dialog is one kind of scripted behavior. The other kind is choreography: characters turning, walking, and speaking in order. This section gathers the character actions you need and explains how Popochiu keeps them in order.

Actions and `await`

Most character actions take time: a line takes seconds to speak, a walk takes seconds to finish. They are coroutines (Chapter 5). If you `await` them, the next line of your script waits. If you do not, the script races ahead and the actions overlap.

```
# Correct: one after the other
await C.player.say("I will walk to the door.")
await C.player.walk_to(Vector2(120, 200))

# Wrong: the walk starts while the player is still talking
C.player.say("I will walk to the door.")
await C.player.walk_to(Vector2(120, 200))
```

The first rule of Popochiu scripting is therefore: `await` **every action whose finish you care about.**

The actions you need

All are methods on a character, for example `C.player` or `C.Frog`.

Action	What it does
`say("text")`	The character speaks a line. Awaitable.
`face_up()`, `face_down()`, `face_left()`, `face_right()` (and the diagonals `face_up_right()`, `face_down_right()`, `face_down_left()`, `face_up_left()`)	Turn to a direction.
`face_direction(dest: Vector2)`	Turn towards a point.
`face_clicked()`	Turn towards the thing just clicked.
`face_away()`	Turn away.
`walk_to(pos: Vector2)`	Walk to a point.
`walk_to_clicked()`	Walk to the clicked object.
`walk_to_prop(id)`, `walk_to_hotspot(id)`, `walk_to_marker(id)`	Walk to a named object or marker.
`teleport_to_position(pos)`, `teleport_to_marker(id)`	Jump there at once.
`grab()`	Play the grabbing animation.
`idle()`	Go back to standing.
`play_animation(label, fallback := "")`	Play a named animation.
`fade_in()`, `fade_out()`	Fade the character.

For a character to walk to a marker you must first create the marker in the room's **Markers** group (Room tab, "+") and give it a name, say `FrogSpot`. Then:

```
# fragment: inside any handler or sequence
await C.Frog.walk_to_marker("FrogSpot")
```

The Frog and the player use the same methods, because both are `PopochiuCharacter`s.

Animation, honestly

Walking, talking, and idling are played by the character's own `AnimationPlayer`, with animation names Popochiu looks up for each direction (the defaults are `idle`, `walk` and `talk`, with optional direction suffixes such as `_d`, `_l`, `_r`). Setting those up with real art is a big topic and the book's placeholder squares do not need it: if the `AnimationPlayer` has no animations at all, `play_animation` returns without error, so the Frog can speak and turn with a still picture. When you do have animations, `await C.Frog.play_animation("name", "fallback")` plays one and waits for it to finish, falling back to the second name if the first does not exist (and to `idle` if you give none). If the character has an `AnimationPlayer` with animations but neither name exists, Popochiu prints an error. The character script has override points (`_play_idle`, `_play_walk`, `_play_talk`, `_play_grab`) for custom behavior.

Queues: running a list of actions

`await` makes a script a straight line. For longer choreography, Popochiu offers a list-based style: build an array of instructions and run it with `E.queue(...)` or `E.cutscene(...)`. Each character action has a `queue_` **twin**: `say` has `queue_say`, `walk_to` has `queue_walk_to`, `face_down` has `queue_face_down`. The twin does **not** run the action. It returns a `Callable` that the queue calls when its turn comes.

```
# fragment: a queue of instructions, one after another
await E.queue([
    C.player.queue_face_right(),
    C.Frog.queue_say("Ah. The new guest."),
    C.player.queue_say("Hello."),
])
```

Compare the two styles for the same sequence.

```
# The await style
await C.player.face_right()
await C.Frog.say("Ah. The new guest.")
await C.player.say("Hello.")
```

Both do the same thing. `E.queue` and `E.cutscene` are coroutines, so `await` them when the code after them must wait. The queue is easier to read when the list is long, and it has special properties that matter in a cutscene.

> **Note:** Not every action has a `queue_` twin. There is `queue_face_up()`, `queue_face_right()` and the other seven fixed directions, `queue_say`, `queue_walk_to`, `queue_walk_to_marker`, and so on, but **no** `queue_face_direction`. For a one-off step without a twin you can put your own function in the list, as in `func(): await C.player.face_direction(C.Frog.global_position)` (a lambda is a `Callable`, and the queue accepts any `Callable`).

A queue list also accepts plain strings. `"Frog: Ah. The new guest."` makes the character whose script name is `Frog` say the line (a name followed by a colon), and a string of only dots is a pause: `"."` waits a quarter of a second, `".."` half a second, `"..."` one second. A string with no colon is shown in the centered system-text box.

> **Warning:** Never start `E.queue` or `E.cutscene` from inside an instruction of another queue. A queue refuses to start while another is playing, and waits for it to end, while the outer queue waits for the inner one: the game freezes with the interface blocked.
>
> **Warning:** The `queue_` twins only work **inside** `E.queue([...])` or `E.cutscene([...])`. Calling `C.player.queue_say("Hi")` on its own line does nothing at all, and prints no error. If a line of dialogue never appears, check whether it is a bare `queue_` call.

What E.queue and E.cutscene do for you

- `E.queue(instructions, show_gui := true)` runs the instructions in order and **blocks the player's input** while it runs. At the end the interface is unblocked again. (This means you do not call `G.block()` or `G.unblock()` yourself inside a queue.)
- `E.cutscene(instructions)` is the same idea for story scenes, and in addition the player can **skip** it by pressing ESC (the `popochiu-skip` input action).

Waiting inside a sequence

In the await style, a pause is `await E.wait(1.0)`. `E.wait(time := 1.0)` waits that many seconds (during a skipped cutscene it waits one frame instead). Inside a list, the queue twin is `E.queue_wait(1.0)`, which returns a `Callable` like any other twin, and the strings `"."`, `".."` and `"..."` are shorthand pauses. For a custom step, the engine offers `E.queueable(node, method, params, signal_name)`, a helper that turns any method of any node into a queue item; by default it waits for the method to finish, or for the named signal on the node if you pass `signal_name`.

Skipping and why state must not live in the cutscene

The player skips with the `popochiu-skip` input action (ESC by default); Popochiu acts when the key is *released*. From the source, skipping does **not** cut the list short. It sets a flag, plays a short cutscene transition, and then every remaining instruction still runs, but the ones that take time return at once: `say`, `E.wait`, `grab`, `idle`, the string forms, and the inventory pop-up. Camera zooms and shakes, and walking, are not shortened. One more thing is skipped completely: a step made with `E.queueable` is not called at all while the flag is set (the source marks this as unfinished). So a custom function wrapped by `E.queueable` can silently never run. The design that makes you safe is simple:

> **Rule:** A cutscene tells the story; it never carries the only copy of a game fact.

Put every state change before the cutscene or after it, outside the list of instructions. The Hallway intro sets its flag on the first line and then runs the cutscene. If the player skips, the flag is already set and the Frog and the player are in a fine state. If the cutscene contained `queue_`d state changes and a skip dropped them, the game would be broken for players who skip.

16.6 Camera effects and the Hallway intro

The camera

Popochiu has a main camera reached as `E.camera`. Its effects are asynchronous like character actions, so they have `queue_` twins:

Call	Effect
`E.camera.shake(strength := 1.0, duration := 1.0)`	Shake the view and wait for `duration` seconds.
`E.camera.shake_bg(strength := 1.0, duration := 1.0)`	The same shake, but the script does not wait ("bg" = background task).
`E.camera.stop_shake()`	Stop shaking.
`E.camera.change_zoom(target := Vector2.ONE, duration := 1.0)`	Tween the Camera2D zoom. Like any Godot 4 `Camera2D`, a larger vector zooms in.
`E.camera.change_offset(offset)`	Set the camera offset.

Each has a `queue_` twin: `E.camera.queue_shake(...)`, `E.camera.queue_shake_bg(...)`, `E.camera.queue_change_zoom(...)`, `E.camera.queue_change_offset(...)`. The shake `strength` is a distance in game pixels: every frame the view jumps to a random offset of up to that many pixels in each direction. A strength of `1.0` is almost invisible; at 1080p, 6 to 12 is a clear shake. A queue stops any shake still running when it ends. (The source comment on `change_zoom` says the opposite about zoom direction; that is a leftover from Godot 3. The method simply sets `Camera2D.zoom`, so Godot 4 rules apply.)

Camera effects are seasoning. A tiny shake when the Frog slams the desk, a slow zoom when something important is said, each used once, make a scene. Constant shake makes players seasick, and an effect that never resets (a zoom you forget to undo) breaks the next scene. Whatever you change, change it back before the cutscene ends.

The first-visit intro

We want a short scene the first time the player enters the Hallway. Use two room functions with different roles:

- `_on_room_entered()` runs when the room is entered. Use it for **setting up** (the `_apply_state()` of Chapter 15).
- `_on_room_transition_finished()` runs when the fade is over and the player can see the room. Use it for **things the player should watch**. It does not run when a save is loaded, which is exactly right: loading a game must not replay the intro.

Open the Hallway's room script and bring it to this state. It keeps the Chapter 14 arrival code and the Chapter 15 `_apply_state()`; keep the header lines the dock generated (`@tool`, `extends PopochiuRoom`, and the `Data`/`state` lines).

```gdscript
# Hallway room script (room_hallway.gd).
func _on_room_entered() -> void:
    if C.player.last_room == "Bedroom":
        await C.player.teleport_to_marker("FromBedroom")
    _apply_state()

func _on_room_transition_finished() -> void:
```

```
    if Globals.hallway_intro_seen:
        return
    # Set the flag first: a skip or a save cannot make the scene play twice.
    Globals.hallway_intro_seen = true
    # The player stands left of the Frog, so they face right and he faces left.
    # Swap the two face calls if your layout is the other way round.
    await E.cutscene([
        E.camera.queue_change_zoom(Vector2(1.15, 1.15), 0.8),
        C.player.queue_face_right(),
        C.Frog.queue_face_left(),
        C.Frog.queue_say("Ah. The new guest."),
        C.Frog.queue_say("You are eleven minutes late."),
        E.camera.queue_shake(8.0, 0.5),
        C.Frog.queue_say("Also, the carpet bites. Mind your ankles."),
        C.player.queue_say("Noted."),
        E.camera.queue_change_zoom(Vector2.ONE, 0.6),
    ])

func _on_restore_from_savegame() -> void:
    _apply_state()

func _apply_state() -> void:
    if Globals.bone_taken:
        get_prop("BonePile").disable()
    if Globals.twine_taken:
        get_prop("TwineBall").disable()
```

Walk through the intro as a list of beats:

Beat	Instruction	Purpose
1	Zoom in	Focus: something is about to happen.
2	Both characters face each other	The scene has two people in it.
3 to 4	Two Frog lines	Establish voice: formal and bored.
5	Short shake	The punctuation: the Frog slams the desk.
6	Third Frog line	A hint about the world.
7	Player answers "Noted."	The player's calm voice, a small joke.
8	Zoom back	Return to normal.

If the player presses ESC, the flag is already set, so the intro will not come back. The speech steps finish at once, and the camera steps still run to the end, so the final zoom back always happens and the camera is never left zoomed (Section 16.5). The Frog's `walk`-free scene leaves both characters where they stood.

> **Try it:** Run the game, walk through the unlocked door, and watch the intro. Press ESC during it on a second playthrough (reset the flag by starting a new game). Expected result: the scene plays once; ESC skips it; after skipping, the zoom is normal and the Frog is clickable. Load a save made in the Hallway and confirm the intro does not replay.
>
> **Note:** All the twins used here (`queue_face_right`, `queue_face_left`, `queue_say`, and the three camera twins) exist in Popochiu 2.2.1. If you prefer the await style, write the same steps as separate `await` lines inside `_on_room_transition_finished`; the design (flag first, state outside the cutscene) does not depend on the queue form.

16.7 Testing dialogue

Dialogue has bugs that code does not: dead ends, lines that repeat, options that never show. Test it as a tree.

Coverage

A good target is **branch coverage**: every option, in every available state, played at least once. Write the table before you test. For ChatWithFrog:

#	Starting flags	Action	Expected
1	New game, no flute	Click Frog	First-meeting greeting; menu: Who, Bye
2	After 1	Choose Who	Two Frog lines; menu now: Flute, Bye
3	After 2	Choose Flute (no flute)	"No flute. Sad."; menu returns
4	After 3	Choose Bye	Frog line; dialog ends; control returns
5	Re-click	Click Frog	"You again."; menu: Flute, Bye (Who gone)
6	Flute held	Choose Flute	Trade; flute removed; Key added; dialog ends
7	After 6	Click Frog	"The rent is paid in music."; menu: Bye only
8	After 6	Use Key on Bedroom Door (if you go back)	"Already unlocked"; Key stays
9	After 6	Click Drawer	"Old socks"; no extra Key
10	Any	Save in the Hallway, quit, load, click Frog	Same menu as before the save
11	After 1, flute never made	Go to Bedroom and back	Intro does not replay
12	Any	Use the Key, Bone or Twine on the Frog	A Frog line, never silence

The questions to ask

- **Dead ends.** From every menu state, is there a way out? (Yes, "Bye" is always visible.)
- **Orphaned options.** Is there an option that can never become visible? (If `Flute` stays hidden whatever you do, the puzzle is blocked.)
- **Repetition.** Which lines can the player see twice? Are they still acceptable the second time?
- **Order independence.** Does the conversation still work if the player does things in a different order, such as making the flute before ever meeting the Frog? (In our design, yes: the Flute option is gated on "Who was asked", not on the flute. The player must ask "Who" once, which costs nothing.)
- **Interrupted states.** Quit mid-conversation and load: does the Frog still make sense? (The menu is rebuilt from flags in `_on_start`.)

> **Try it:** Play the table above. Then try to break the Frog: click him ten times in a row, choose "Goodbye" at every menu, use the Bone on him, use the Key on him. Expected result: nothing crashes and nothing is silent. The Frog's `_on_item_used` answers every item with a line in his voice; if one of your attempts is silent, find out why.

Lab 16: The Frog and the first-visit intro

Time: 60 to 90 minutes. **You need:** the project at the end of Lab 15: Bedroom, Hallway, Drawer, Door, items Key, Bone, Twine, BoneFlute, and the working puzzle chain with Globals.

Part A: Globals

1. Open `popochiu_globals.gd` and add below the Chapter 15 variables:

```
# Hallway story (Chapter 16)
var hallway_intro_seen: bool = false
var frog_met: bool = false
var frog_who_asked: bool = false
var flute_given: bool = false
```

Part B: The Frog and the dialog

2. In the dock's Main tab, create a character `Frog` (not the player). Give it a placeholder picture.
3. Open the Hallway scene. In the Room tab, add `Frog` to the room using the characters group. Move it to the landlord's spot.
4. In the dock's Main tab, create a dialog tree named `ChatWithFrog`.
5. Select the dialog's `.tres` in the FileSystem dock. In the Inspector add three Options: `Who` ("Who are you?"), `Flute` ("Do you need a flute?", `visible` unchecked), and `Bye` ("Goodbye.").
6. Replace the dialog script with the complete version in Section 16.4. Keep the header lines the dock generated.
7. Replace the Frog's script with the version in Section 16.4.
8. The Frog script in Section 16.4 already has an `_on_item_used` that answers any item with a line in his voice (and reacts to the Bug that Chapter 17 adds). Check that it is in your file.

Part C: The intro

9. Open the Hallway room script. Merge in the functions from Section 16.6. Keep any code Chapter 14 and 15 put there.
10. Run the game. Do the happy path: Bedroom, key, door, Hallway.

Part D: Test

11. Play the intro. Check the zoom, the shake, and the order of the lines. Adjust the timing numbers until it reads well.
12. Run table rows 1 to 12 from Section 16.7. Use a fresh start between groups.
13. Check the edge case deliberately: with the flute in hand and `Key` already in the inventory (use a temporary cheat), the Frog should refuse and the flute must remain. Remove the cheat.
14. Save in the Hallway after the trade, quit, load, and click the Frog.
15. Commit: `git add -A`, `git commit -m "Chapter 16: Frog dialog and Hallway intro"`.

Expected result: entering the Hallway for the first time plays a short skippable scene with a camera zoom and shake; clicking the Frog starts a conversation with Who, Flute (hidden until Who is asked), and Bye; offering the flute without owning it is refused; with the flute in hand the Frog takes it, gives a Key, and ends the conversation; the reward never breaks the drawer or the door; and everything survives a save and load.

Chapter summary

- A dialog tree is built from hubs, one-shot options, gated options, and endings. Always keep an exit, and never let the menu run empty.
- Write short lines with one job each, keep each character's voice consistent, prefer subtext to exposition, and give hints in character.
- Create a dialog from the dock's Dialog trees group. Define options in the Inspector (`id`, `text`, `visible`, `disabled`) or in code with `_on_build_options`.
- Start it with `await D.ChatWithFrog.start()`. Handle choices in `_option_selected` with a `match` and a `_:` branch that calls `stop()`, ending with `_show_options()`; or use per-option `_on_option_<id>` functions, but never mix the two styles.
- `turn_on_options(["Id"])` and `turn_off_options(["Id"])` take arrays. Rebuild the menu from Globals flags in `_on_start()` so it is correct after leaving or loading.
- Character actions (`say`, `face_*`, `walk_to`) are coroutines: `await` them. In `E.queue` and `E.cutscene`, use the `queue_` twins; outside a queue they do nothing.
- `E.camera` gives `shake`, `change_zoom`, and `change_offset`, each with a `queue_` twin. Reset whatever you change.
- Cutscenes can be skipped, so never keep the only copy of a game fact inside one. Set flags before or after.
- A reward that duplicates a puzzle item needs a design: here the spare Key is safe because the flags describe the world, the Door tolerates an extra Key, and the Frog never takes the flute unless the reward can be given.

Review questions

1. Name the four building blocks of a dialog tree, and say which one stops the player from getting trapped.
2. In which function do you handle the player's choice, and what does the `_:` branch of the `match` do?
3. True or false: `turn_off_options("Who")` is the correct call to hide the option `Who`.
4. What does `D.say_selected()` do?
5. Why is `_show_options()` placed at the end of `_option_selected` and not inside each branch? What happens if you also call it inside a per-option handler?
6. Why does the Frog's dialog set options from flags in `_on_start()` instead of relying only on `turn_off_options` calls made earlier?
7. You write `C.Frog.queue_say("Hello")` on its own line in a handler. What happens, and why?
8. Which input skips an `E.cutscene`, and what rule keeps skipping from breaking the game?
9. Why does the intro run in `_on_room_transition_finished` rather than `_on_room_entered`, and what does that mean for loading a saved game?
10. After the Frog gives the spare Key, which flags change, which do not, and why does the drawer not give a second key?

Exercises

1. **Practice (easy).** Add a fourth option to `ChatWithFrog`, `Carpet` ("Why does the carpet bite?"), as a repeatable hub option with three different Frog replies chosen by `pick_random()`. Test that it never ends the conversation.

2. **Practice (easy).** The Frog's `_on_item_used` has a `match` on `item.script_name` with branches for Bug and BoneFlute and one default. Add separate branches for Key, Bone and Twine, each with its own line in the Frog's voice.
3. **Apply (medium).** Add a marker `FrogSpot` in the Hallway. Make the Frog walk to it at the end of the intro cutscene (using `C.Frog.queue_walk_to_marker("FrogSpot")`) and say one more line. Place the marker so the walk looks good. Confirm that skipping the cutscene still leaves the Frog usable.
4. **Apply (medium).** Give the spare Key a target: the front door of the hotel. Add a hotspot `FrontDoor` to the Hallway whose `_on_item_used` with the Key sets a new flag `front_door_open` (declare it in Globals), removes the Key, and says a line, and where a repeat use is tolerated. Do **not** change the Jar for this: Chapter 17 rewrites the Jar script. Run the softlock questions from Chapter 15 on it.
5. **Challenge (hard).** Design and implement a second conversation, `ChatWithCat`, for a Cat character, with a hub, a one-shot, a gated option unlocked by a Globals flag set in a different room, and an ending. Draw the tree first, write a coverage table with at least eight rows, and play it.

Common mistakes

- **Symptom:** the conversation freezes after a choice and the menu never returns. **Cause:** `_show_options()` is missing, or an unhandled id has no `_:` branch. **Fix:** end `_option_selected` with `_show_options()` and keep the `_:` branch calling `stop()`.
- **Symptom:** the menu appears twice, or lines repeat. **Cause:** both `_option_selected` and a per-option function are defined, and both show the menu. **Fix:** use one style per dialog.
- **Symptom:** `turn_off_options("Who")` gives a type error, or does nothing. **Cause:** the function takes an array of ids. **Fix:** write `turn_off_options(["Who"])`.
- **Symptom:** a line in a cutscene never appears and nothing is printed. **Cause:** a `queue_` call used outside `E.queue` or `E.cutscene`, or a plain `say` placed in the middle of a queue list. **Fix:** use `queue_say` inside the list, or switch the whole sequence to `await` calls.
- **Symptom:** the Flute option is visible too early, or never shows. **Cause:** `visible` is checked in the Inspector, or `disabled` is checked (so `turn_on_options` does nothing). **Fix:** uncheck `visible` for the hidden start; leave `disabled` off.
- **Symptom:** after skipping or loading, the Frog or the Hallway is in the wrong state. **Cause:** a flag or an item change was placed after a `say` or inside the cutscene list. **Fix:** change state first, outside the list, and then talk.

Further reading

- Popochiu, script your first dialogue: https://carenalgas.github.io/popochiu/how-to-develop-a-game/script-your-first-dialogue/
- Popochiu, working with game state: https://carenalgas.github.io/popochiu/the-engine-handbook/working-with-game-state/
- Popochiu documentation home: https://carenalgas.github.io/popochiu/
- Popochiu source, the `E` singleton (queue, cutscene, wait): https://github.com/carenalgas/popochiu/blob/v2.2.1/addons/popochiu/engine/popochiu.gd
- Godot, GDScript basics (coroutines and `await`): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

Chapter 17: Mini-games and the Boss Fight

Learning objectives

By the end of this chapter you will be able to:

- Explain why a mini-game is best built as an ordinary Godot scene that sits on top of a Popochiu adventure.
- Design a scene that follows the "contract pattern": it emits `finished(won: bool)` once, and the caller waits for it with `await`.
- Choose a `CanvasLayer` layer number that puts a mini-game above the Popochiu GUI but below the cursor.
- Block and unblock the Popochiu interface with `G.block()` and `G.unblock()` while a mini-game runs.
- Build and wire up two mini-games for Weird Hotel: the jar "catch the bug" game and the ceiling boss dodge fight.
- Add Input Map actions, tune difficulty over time, add "game feel" (a hit flash and a screen shake), and write the death-and-respawn flow.
- Test each mini-game alone with F6 and write defensive code that cannot end twice or touch freed nodes.

Key terms

- **Mini-game**: a small, self-contained game with its own rules, started from inside a larger game.
- **Contract pattern**: an agreement between two pieces of code about how they talk; here, "the scene emits `finished(won)` exactly once, and the caller waits for it".
- **Signal**: a message a node can emit; other code connects to it or waits for it with `await`.
- `await`: a GDScript keyword that pauses the current function until a signal is emitted or a coroutine finishes. The function becomes a coroutine.
- **CanvasLayer**: a node that draws its children on a separate layer. A higher `layer` number draws on top of a lower one.
- **Input Map**: the project's list of named actions (such as `move_left`) and the keys, buttons or sticks bound to them.
- **Interface block**: Popochiu's state in which room clicks and GUI clicks are ignored. You turn it on with `G.block()` and off with `G.unblock()`.
- **Invulnerability window**: a short time after being hit during which the player cannot be hit again.
- **Game feel**: the small effects (flashes, shakes, sounds) that make actions feel physical.
- **Respawn**: putting the player back in a safe place after a defeat.
- **Difficulty curve**: how a game gets harder as time passes.
- **Defensive code**: code that checks its own assumptions (is this already done? is this node still alive?) so mistakes do not crash it.

17.1 Why a plain Godot scene inside an adventure?

A Popochiu game is built from rooms, characters, props and dialogs. Those are excellent for talking, walking and solving puzzles. They are not good at everything. Suppose you want the player to catch a bug that hops around the screen, or to dodge falling blocks with the keyboard. Those need different rules: real-time movement, timers, collision checks, lives. You could try to bend Popochiu to do it, but Popochiu is built for clicking, not dodging.

You already have the better tool. Godot lets you build any scene you like, and Chapters 6 to 9 taught you how. So the plan is:

1. Build the mini-game as a normal Godot scene, in its own folder (`res://minigames/`).
2. Put it on top of the adventure when it is needed.
3. Wait until it says it is over.
4. Remove it, and carry on with the adventure.

This has three big advantages.

It is simple. The mini-game does not need to know anything about rooms, inventories or dialogs. It only needs to know its own rules. That makes it easy to write and easy to test.

It is reusable. Because the mini-game does not depend on Weird Hotel, you could copy `catch_bugs.tscn` into a different game and it would work there too.

It is easy to change. If you later want the jar to launch a different game, you change one line in the jar's script. The adventure does not care what is inside.

The price is that you must be careful at the **border** between the two worlds. Two sets of code are both listening for the mouse. Two sets of code may both want to be in charge. This chapter is mostly about managing that border safely.

> **Note:** "Mini-game" does not mean "small and bad". The boss fight in this chapter is about 200 lines of GDScript and would be a fine game on its own. It is "mini" only in the sense that it is a short section inside a bigger game.

What Weird Hotel will gain

By the end of the chapter, Weird Hotel has two new things:

Where	What	Result
Jar (Hallway prop)	Catch-the-bug mini-game	Win: the `Bug` item is added to your inventory. Lose: click the jar to try again.
Ceiling (Hallway hotspot)	Dodge-the-blocks boss fight	Win: the ceiling is quiet. Lose: you wake up in the Bedroom, keeping your items and your puzzle progress.

The scene files are `res://minigames/catch_bugs.tscn` with `catch_bugs.gd`, and `res://minigames/boss_fight.tscn` with `boss_fight.gd`.

17.2 The contract pattern

When two pieces of code work together, they need an agreement. We call it a **contract**. A good contract is short and has no loopholes. Ours has four clauses.

1. The mini-game is a scene whose root node declares `signal finished(won: bool)`.
2. The mini-game emits `finished` **exactly once**, when it ends. `won` is `true` for success, `false` for failure.
3. The **caller** (the Jar or Ceiling script) creates the scene, adds it to the tree, waits with `await game.finished`, and then removes it.
4. The mini-game never touches rooms, items or dialogs. It only reports its result. What the result means is the caller's business.

Here is the flow as a picture.

```
Jar script (caller)                      CatchBugs scene (mini-game)
-------------------                      ---------------------------
walk to jar, say a line
instantiate the scene  ------------------> exists, _ready() runs
add it to the tree
block the Popochiu GUI
await game.finished   (caller pauses)      player clicks bugs, timer runs
                                           ... game ends ...
                      <------------------  finished.emit(won)
caller wakes up, gets `won`
unblock the GUI
free the scene
if won: give the Bug item
```

Signals and await, quickly

You met signals in Chapter 5. A signal is a named message. A node declares it with the `signal` keyword and sends it with `.emit(...)`:

```gdscript
signal finished(won: bool)

func _example() -> void:
    finished.emit(true)
```

Anything can wait for a signal by putting `await` in front of it:

```gdscript
var won: bool = await game.finished
```

When GDScript reaches that line, the function **pauses**. The rest of the game keeps running normally. When `game` emits `finished`, the function wakes up on the next line, and `won` holds the value that was emitted.

The rule for what `await` returns, from the Godot documentation:

Signal arguments	`await signal` gives you
none	`null` (nothing useful)
one argument	that argument's value
two or more	an `Array` of the values

Our signal has exactly one argument, `won`, so `var won: bool = await game.finished` is correct.

A function that contains `await` is called a **coroutine**. Any function that calls it with `await` also pauses. That is why Popochiu code is full of `await C.player.say("...")`: each line waits for the previous action to finish. Our mini-game call is the same kind of line.

> **Why it works:** Popochiu's own click handlers are coroutines. When the player clicks the jar, Popochiu calls `_on_click()`. Inside, we `await` the mini-game. The adventure is not frozen; the mini-game simply sits on top, and the jar's script is paused at one line until the result arrives.

What can go wrong with a contract

A contract protects you only if both sides keep it. These are the classic ways it breaks, and the code in this chapter prevents each one.

Failure	What happens	Our defence
The mini-game never emits `finished`	The caller waits forever; the player is stuck behind a blocked GUI	Every mini-game has a time limit or a lose condition, and one `_end()` function that is the only place that emits
The mini-game emits twice	The caller wakes once, but other listeners get a duplicate (for example a second reward)	A `done` flag at the top of `_end()`
The mini-game frees itself, then emits	The caller may use a dead object	The caller frees the scene, not the scene itself
Someone frees the mini-game while the caller is waiting	The caller never wakes up	Only the caller frees it, and only after `finished`
The caller forgets to unblock the GUI	The player cannot click anything	`G.unblock()` runs on the very next line after `await`

> **Warning:** If a node is freed while one of its own functions is paused at an `await`, Godot prints an error such as "Resumed function after await, but class instance is gone." The mini-game's `_end()` awaits a short timer. This is safe as long as nobody frees the mini-game during that half second. That is one more reason the caller, which only frees after `finished`, must be the one in charge of freeing.

Who frees the scene?

There are two common designs. In the first, the mini-game calls `queue_free()` on itself after emitting. In the second, the caller frees it after `await`. We use the second, for three reasons: the caller may want to read a value from the mini-game after it finishes (a score, a time); the owner of an object should be the one to destroy it; and it makes the F6 test simple, because the scene just stays on screen showing the result.

17.3 Layers, blocking, and input

The mini-game must appear above the adventure. It must also stop the adventure reacting to the mouse. Three things control that: the CanvasLayer number, the Popochiu interface block, and each Control's mouse filter.

CanvasLayer numbers

A `CanvasLayer` is a node that draws its children on its own layer, independent of the camera. Layers with larger `layer` numbers are drawn on top. The default 2D world (rooms, characters, props) is on layer 0. From the Popochiu source (version 2.2.1), the Popochiu engine scene `E` has these layers:

Layer number	What is drawn there
0	The room, characters, props (the normal 2D world)
10	The Popochiu graphical interface (verb bar, inventory, dialog options)
15	Screen transitions (the fade curtain)
100	The Popochiu cursor

We want the mini-game above the GUI (10) and the transitions (15), but below the cursor (100), so the player's cursor is still drawn over the game. **We use layer 50.** The original bare-bones course used 100, which hides the cursor behind the game. Layer 50 is better.

> **Note:** The numbers 10, 15 and 100 come from reading Popochiu's source code, not from a documentation page, because the Popochiu docs do not yet cover this. If a future version changes them, the pattern is the same: pick a number between the GUI layers and the cursor.

Because a `CanvasLayer` ignores the camera, the Popochiu camera shake (`E.camera.shake`, Chapter 18) does **not** move your mini-game. If you want the mini-game to shake, you shake it yourself. We do that in section 17.7.

Blocking the Popochiu interface

Popochiu props, hotspots and characters are `Area2D` nodes that react to the mouse. If the player clicks on top of your mini-game, the click might still reach a prop in the room behind. Popochiu has a switch for this:

```
G.block()      # the interface and room objects ignore the player
G.unblock()    # back to normal
G.is_blocked   # true while blocked
```

From the Popochiu source, clickables ignore input while `G.is_blocked` is true. So the standard order is:

1. `G.block()`
2. Add the mini-game to the tree.
3. `await game.finished`
4. `G.unblock()`
5. Free the mini-game.

> **Warning:** There is one exception. If you start a mini-game from inside `E.queue([...])` or `E.cutscene([...])`, those functions already block the interface at the start and unblock it at the end. Then you must **not** call `G.unblock()` yourself in the middle. Our Jar and Ceiling scripts do not use `E.queue`; they use plain `await`, so they do their own blocking. Popochiu has a helper for the queue case, `E.queueable(node, method, params, signal_name)`, which waits on a signal inside a queue. We do not need it in this book. If you try it, remember two facts from the source: `E.queue` cannot be started inside another queue, and `E.queueable` steps are not called at all if the player skips a cutscene. The combination of a queue and a mini-game is not an officially documented recipe.

Mouse filters on Controls

`Control` nodes (Button, ColorRect, Label) have a **Mouse Filter** property:

Mouse Filter	Meaning
Stop	The control receives mouse events and they go no further
Pass	The control receives them and passes them to nodes behind it
Ignore	The control is invisible to the mouse

The default is Stop. That is what we want for the big dark background of a mini-game: clicks on it stop there. For decorations that should never receive clicks (a label, falling blocks), Ignore is a good choice. In this chapter, the full-screen background is Stop and the blocks we create in code are set to Ignore.

Using both `G.block()` and a Stop background is "belt and braces". If one fails, the other still protects the room.

> **Note:** In the Simple Click GUI, `G.block()` also switches the cursor to the "wait" picture and tells the cursor to ignore changes. A mini-game wants a normal pointer, so the caller scripts below add two lines right after `G.block()`: `Cursor.unblock()` and `Cursor.show_cursor("normal")`. `Cursor` is another Popochiu autoload (it draws the cursor on layer 100). When the caller later runs `G.unblock()`, the GUI restores the right cursor by itself.

Pausing: what we do not do

Godot can pause the whole tree with `get_tree().paused = true`. It is tempting: pause the adventure, play the mini-game, unpause. Do not do this here. Popochiu has no pause handling of its own. Characters walk with tweens, and a paused tree freezes them. Worse, your mini-game would freeze too unless you set its `process_mode` to Always. The mini-game layered on top of a running (but blocked) adventure is simpler and has fewer surprises.

If you ever do need a true pause, the Godot setting is `process_mode = Node.PROCESS_MODE_ALWAYS` on the nodes that must keep running while the tree is paused.

Where to add the scene in the tree

A node must be in the scene tree to run. We add the mini-game to the root of the tree, with `get_tree().root.add_child(game)`. The root is the top-level window node, above the current scene, so the mini-game does not belong to any room. That has an important consequence: **when the room changes,**

the mini-game does not disappear by itself. You must free it yourself, which the contract already says.

> **Try it:** Open any existing Weird Hotel prop script and find a `_on_click`. On paper, list which of the five contract clauses it would need to start a mini-game. You will check your list against the real Jar script in the next section.

17.4 The Jar: catch the bug

Let us build the first mini-game. The player must click a hopping bug five times in ten seconds.

The scene

Create the scene `res://minigames/catch_bugs.tscn`. Use **Scene > New Scene**, then **Other Node**, and choose `CanvasLayer`. Build this tree.

```
CatchBugs (CanvasLayer)          script: catch_bugs.gd, Layer = 50
└── Blocker (ColorRect)          Full Rect, color black with alpha 0.6, Mouse Filter = Stop
    ├── Target (Button)          text "BUG", Custom Minimum Size 64 x 64
    └── Label (Label)            top-left, font size about 28
```

Steps:

1. Create the folder `minigames` in the FileSystem dock (right-click `res://` > **New Folder**).
2. In the new scene, select the root and rename it `CatchBugs`. In the Inspector set **Layer** to `50`.
3. Add a `ColorRect` child named `Blocker`. In the toolbar above the viewport use **Layout > Anchor Preset > Full Rect**. In the Inspector set **Color** to black and drag the alpha slider to about 0.6.
4. Add a `Button` child to `Blocker` named `Target`. Set **Text** to `BUG`, and under **Layout > Custom Minimum Size** set 64 and 64.
5. Add a `Label` child to `Blocker` named `Label`. Under **Theme Overrides > Font Sizes** set the font size to 28. Set the font color to white.
6. Save with Ctrl+S as `res://minigames/catch_bugs.tscn`.

The node names must match exactly, because the script finds them by path.

The script

Attach a new script to `CatchBugs` and save it as `res://minigames/catch_bugs.gd`.

```gdscript
# res://minigames/catch_bugs.gd
# Attached to the root node (CanvasLayer) of catch_bugs.tscn.
# Contract: emits finished(won) exactly once. The caller frees this scene.
class_name CatchBugsGame
extends CanvasLayer

signal finished(won: bool)

const NEEDED: int = 5            # hits needed to win
const TIME_LIMIT: float = 10.0   # seconds
const START_HOP: float = 1.2     # seconds before the bug hops by itself
const MIN_HOP: float = 0.5       # the hop gets this fast at the end
const TOP_MARGIN: float = 48.0   # keeps the bug away from the label
const END_PAUSE: float = 0.7     # show the result for a moment

var hits: int = 0
```

```gdscript
var time_left: float = TIME_LIMIT
var hop_left: float = START_HOP
var done: bool = false           # stops the game from ending twice

@onready var target: Button = $Blocker/Target
@onready var label: Label = $Blocker/Label


func _ready() -> void:
    target.pressed.connect(_on_target_pressed)
    _move_target()
    _update_label()


func _process(delta: float) -> void:
    if done:
        return
    time_left -= delta
    hop_left -= delta
    if time_left <= 0.0:
        time_left = 0.0
        _update_label()
        _end(false)
        return
    if hop_left <= 0.0:
        _move_target()
    _update_label()


func _on_target_pressed() -> void:
    if done:
        return
    hits += 1
    if hits >= NEEDED:
        _update_label()
        _end(true)
    else:
        _move_target()
        _update_label()


# Jump the bug to a random spot and decide when it hops next.
func _move_target() -> void:
    var area: Vector2 = get_viewport().get_visible_rect().size
    var max_x: float = maxf(area.x - target.size.x, 0.0)
    var max_y: float = maxf(area.y - target.size.y, TOP_MARGIN)
    target.position = Vector2(randf_range(0.0, max_x), randf_range(TOP_MARGIN, max_y))
    # The more bugs caught, the faster it hops on its own.
    hop_left = lerpf(START_HOP, MIN_HOP, float(hits) / float(NEEDED))


func _update_label() -> void:
    label.text = "Bugs: %d/%d    Time: %.1f" % [hits, NEEDED, time_left]


# The ONLY place that ends the game, and the only place that emits.
func _end(won: bool) -> void:
    if done:
        return
    done = true
    target.disabled = true
    label.text = "Got it!" if won else "Too slow!"
    await get_tree().create_timer(END_PAUSE).timeout
    finished.emit(won)
    if get_tree().current_scene == self:
        print("CatchBugs finished, won = ", won)   # only when run alone with F6
```

Reading the script

`class_name CatchBugsGame`. This gives the script a global name so the caller can use the type `CatchBugsGame`. The name must not clash with a built-in Godot class, and ours does not. The `signal finished(won: bool)` line is clause 1 of the contract.

The constants. Every number you may want to tune sits at the top with a name. Changing `NEEDED` from 5 to 8 is a one-character edit; nothing else changes.

`done`. This flag is the guard against ending twice. Look at `_end`: the first line says "if already done, leave". A second call, from a late click or a timer, does nothing. Look also at `_on_target_pressed`: it checks `done` first because a button can still receive a click in the half second after the game ends.

`_process`. Runs every frame with `delta`, the time since the last frame (Chapter 6). We subtract `delta` from two countdowns: the game timer and the hop timer. When the game timer reaches zero, we clamp it to zero so the label never shows a negative number, and we end the game.

`_move_target`. Picks a random position inside the visible area. `get_viewport().get_visible_rect().size` is the size of the game window in game pixels, so it still works if you change resolution. `maxf(a, b)` returns the larger float and protects against a window smaller than the button. `lerpf(a, b, t)` slides from `a` to `b` as `t` goes from 0 to 1. So after zero hits the bug waits 1.2 seconds before hopping, and as you approach five hits the wait shrinks to 0.5 seconds. That is the difficulty curve for this game.

`_end`. Sets `done`, disables the button, shows a message, waits 0.7 seconds with `await get_tree().create_timer(END_PAUSE).timeout` (a one-shot timer that needs no node), then emits `finished`. The pause lets the player see "Got it!" before the game disappears.

The last two lines run only when the scene was started with F6 (see 17.8). In that case the scene is the main scene, `get_tree().current_scene` is the scene itself, and we print a message so you can see the contract working with no adventure present.

> **Why it works:** The mini-game contains no Popochiu code at all: no `C`, `I`, `G` or `E`. That is deliberate. It means the scene runs by itself with F6, and it means a bug in the adventure cannot break the mini-game, nor the other way round.

The Jar prop

Now the caller. In Weird Hotel the Jar is a prop in the Hallway. If you did not create it in Chapter 14 (Exercise 3), create it now: Room tab of the Hallway, Props, "+", name `Jar`, tick "Will have interaction?", give it a placeholder picture and place it on the floor. Open its script from the Popochiu dock (Room tab, Props, Jar, script icon). Keep the `@tool` and `extends PopochiuProp` lines that Popochiu generated, and replace the rest of the file with this. The look line and the wrong-item reply from the Chapter 12 design table are included, so nothing is silent.

```
# res://game/rooms/hallway/props/jar/prop_jar.gd
# The Jar prop in the Hallway: starts the catch-the-bug mini-game.
@tool
extends PopochiuProp

const CATCH_BUGS := preload("res://minigames/catch_bugs.tscn")
```

```
func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.bug_caught:
        await C.player.say("The jar is empty now. The bug is in my pocket.")
        return
    await C.player.say("Something in the jar wants out.")

    var won: bool = await _play_catch_bugs()

    if won:
        # State first, then talk.
        Globals.bug_caught = true
        I.Bug.add()
        await C.player.say("Got it. It is wiggling in my pocket.")
    else:
        await C.player.say("Too slow. I can click the jar to try again.")


func _on_right_click() -> void:
    if Globals.bug_caught:
        await C.player.say("An empty jar. It looks proud of itself.")
    else:
        await C.player.say("Something in the jar wants out.")


func _on_item_used(_item: PopochiuInventoryItem) -> void:
    await C.player.say("The jar does not want that. It wants a bug.")


# Starts the mini-game, waits for the result, cleans up, and returns the result.
func _play_catch_bugs() -> bool:
    var game: CatchBugsGame = CATCH_BUGS.instantiate() as CatchBugsGame
    G.block()                           # room and GUI ignore the mouse
    Cursor.unblock()                    # but the pointer must still work...
    Cursor.show_cursor("normal")        # ...and look normal, not "wait"
    get_tree().root.add_child(game)     # _ready() of the mini-game runs now
    var won: bool = await game.finished # pause here until the result arrives
    G.unblock()                         # give the player the interface back
    game.queue_free()                   # the caller frees the mini-game
    return won
```

The exact file path depends on your project; the dock created the file for you, so open the script from the dock instead of typing a path. The path in the comment is the usual pattern for a prop named `Jar` in a room named `Hallway`. Do not move or rename these files by hand; use the dock.

Line by line:

- `preload("res://minigames/catch_bugs.tscn")` loads the scene file once, when the script loads. The result is a `PackedScene`, a blueprint.
- `.instantiate()` makes a live copy of the blueprint. `as CatchBugsGame` tells GDScript what type the copy is, so `game.finished` is checked when you write it.
- `Globals.bug_caught` (declared in Section 17.6) is a world fact: "the jar is empty". We use a flag, not `I.Bug.ever_collected`, because `ever_collected` is not stored in save files and would be forgotten after a load (Chapter 15). The flag is set before any speech, as always.
- `I.Bug.add()` adds one Bug to the inventory. The Bug is a collectible: nothing consumes it, but the Frog comments on it (Chapter 16) and the ending mentions it.
- The Jar has a look line and a reply for wrong items, so right-clicking or using the Key on it is never silent.
- The retry path costs nothing. When the player loses, `_on_click` ends, the jar is clickable again, and the next click starts a fresh copy of the mini-game. Each attempt is a new instance, so there is no leftover state to reset.

> **Warning:** The item `Bug` must exist, and `Globals.bug_caught` must be declared (Section 17.6). Create the item first: Popochiu dock, Main tab, Inventory items, the "+" button, name it `Bug`, give it a placeholder picture. If you skip this, `I.Bug` is an error. Per the naming contract for Part III, items are Key, Bone, Twine, BoneFlute and Bug.

Make it testable in the adventure

1. Run the game with F5.
2. Walk to the Hallway and click the Jar.
3. The dark overlay appears with a button labelled BUG hopping around.
4. Click it five times. "Got it!" shows for a moment, the overlay disappears, the player speaks, and the Bug appears in the inventory.
5. Run again and let the time run out. "Too slow!" appears, and the Jar can be clicked again.

> **Try it:** Change `NEEDED` to 3 and `TIME_LIMIT` to 20 in `catch_bugs.gd`, and play through. Then set them back, or choose values that feel fair. You should not have to open the Jar script for this. That is the reward of a clean contract.

17.5 The Input Map and the boss fight

The boss fight needs keyboard movement. In Chapter 6 you saw that the **Input Map** turns physical keys into named actions. The code asks "is `move_left` pressed?" instead of "is the A key pressed?" That has two benefits: you can change keys without changing code, and players can remap keys later.

Add the four actions

1. Open **Project > Project Settings** and click the **InputMap** tab. (Godot spells it "InputMap" on this tab.)
2. In the text box at the top ("Add New Action") type `move_left` and press **Add**.
3. In the list, find `move_left` and click the + at its right. A dialog asks you to press a key. Press **A** and click **OK**.
4. Click + again, press the **Left arrow** key, and click **OK**. Now two keys map to the action.
5. Repeat for `move_right` (D and Right arrow), `move_up` (W and Up arrow), and `move_down` (S and Down arrow).
6. Close Project Settings. Godot saves the map to `project.godot`.

> **Note:** The boss script also contains a small safety net: if an action is missing from the Input Map, it adds the keys itself for that run and prints a warning. That is only a net. You should still add them in the editor, so they appear in the project and you can see and change them.

The scene

Create `res://minigames/boss_fight.tscn`.

```
BossFight (CanvasLayer)          script: boss_fight.gd, Layer = 50
├── Backdrop (ColorRect)         Full Rect, then offsets -32/-32/32/32; dark, alpha 0.85
└── Arena (Control)              Full Rect, Mouse Filter = Ignore
```

```
    ├── Hero (ColorRect)          size 24 x 24, color yellow
    └── Label (Label)             top-left, white, font size 24
```

Steps:

1. New scene with root `CanvasLayer`, rename `BossFight`, **Layer** = 50.
2. Add `ColorRect` child `Backdrop`. Anchor Preset > Full Rect. Then in the Inspector under **Layout > Transform**, set the Offset Left and Top to `-32` and Right and Bottom to `32`. The backdrop is now slightly larger than the screen. This is so the screen shake in 17.7 never reveals a gap at the edge. Set **Color** to a dark blue-grey with alpha about 0.85. The Mouse Filter stays Stop.
3. Add a `Control` child named `Arena`. Anchor Preset > Full Rect. Set **Mouse Filter** to Ignore.
4. Add a `ColorRect` child of `Arena` named `Hero`. Set the color to yellow (for example 1.0, 0.9, 0.3).
5. Add a `Label` child of `Arena` named `Label`. White text, size 24.
6. Save.

The script

Attach a script to `BossFight` and save it as `res://minigames/boss_fight.gd`.

```gdscript
# res://minigames/boss_fight.gd
# Attached to the root node (CanvasLayer) of boss_fight.tscn.
# Contract: emits finished(won) exactly once. The caller frees this scene.
class_name BossFightGame
extends CanvasLayer

signal finished(won: bool)

const SURVIVE_TIME: float = 20.0
const START_LIVES: int = 3
const HERO_SIZE: Vector2 = Vector2(24, 24)
const BLOCK_SIZE: Vector2 = Vector2(20, 20)
const HERO_SPEED: float = 320.0       # pixels per second
const TOP_LIMIT: float = 0.35         # hero cannot go above 35% of the screen height
const FALL_START: float = 200.0       # block speed at the start
const FALL_END: float = 520.0         # block speed at the end
const SPAWN_START: float = 0.55       # seconds between blocks at the start
const SPAWN_END: float = 0.25         # seconds between blocks at the end
const INVULN_TIME: float = 1.0        # safe time after a hit
const END_PAUSE: float = 0.8
const BLOCK_COLOR: Color = Color(0.9, 0.15, 0.2)
const ACTION_KEYS: Dictionary = {
    "move_left": [KEY_A, KEY_LEFT],
    "move_right": [KEY_D, KEY_RIGHT],
    "move_up": [KEY_W, KEY_UP],
    "move_down": [KEY_S, KEY_DOWN],
}

# The caller may switch this off (Chapter 18, accessibility) before adding the scene.
var shake_enabled: bool = true

var lives: int = START_LIVES
var time_left: float = SURVIVE_TIME
var spawn_left: float = 0.0
var invuln_left: float = 0.0
var done: bool = false
var blocks: Array[ColorRect] = []
var _flash_tween: Tween
var _shake_tween: Tween

@onready var arena: Control = $Arena
@onready var hero: ColorRect = $Arena/Hero
@onready var label: Label = $Arena/Label
```

```
func _ready() -> void:
    _ensure_input_actions()
    hero.size = HERO_SIZE
    var area: Vector2 = _arena_size()
    hero.position = Vector2((area.x - HERO_SIZE.x) / 2.0, area.y - HERO_SIZE.y - 24.0)
    _update_label()


func _process(delta: float) -> void:
    if done:
        return
    # progress goes from 0.0 at the start to 1.0 when the time is up
    var progress: float = 1.0 - time_left / SURVIVE_TIME
    invuln_left = maxf(invuln_left - delta, 0.0)
    _move_hero(delta)
    _spawn_blocks(delta, progress)
    _move_blocks(delta, progress)
    if done:                            # a hit on this frame may have ended the game
        return
    time_left -= delta
    if time_left <= 0.0:
        time_left = 0.0
        _update_label()
        _end(true)
        return
    _update_label()


func _move_hero(delta: float) -> void:
    # get_vector turns the four actions into one direction, such as (1, 0) for right
    var dir: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    var area: Vector2 = _arena_size()
    var min_pos: Vector2 = Vector2(0.0, area.y * TOP_LIMIT)
    var max_pos: Vector2 = area - HERO_SIZE
    hero.position = (hero.position + dir * HERO_SPEED * delta).clamp(min_pos, max_pos)


func _spawn_blocks(delta: float, progress: float) -> void:
    spawn_left -= delta
    if spawn_left > 0.0:
        return
    spawn_left = lerpf(SPAWN_START, SPAWN_END, progress)
    var b: ColorRect = ColorRect.new()
    b.color = BLOCK_COLOR
    b.size = BLOCK_SIZE
    b.mouse_filter = Control.MOUSE_FILTER_IGNORE
    b.position = Vector2(randf_range(0.0, _arena_size().x - BLOCK_SIZE.x), 0.0)
    arena.add_child(b)
    blocks.append(b)


func _move_blocks(delta: float, progress: float) -> void:
    var fall_speed: float = lerpf(FALL_START, FALL_END, progress)
    var arena_h: float = _arena_size().y
    var hero_rect: Rect2 = hero.get_global_rect()
    # Walk the array backwards so removing an entry does not skip the next one.
    for i in range(blocks.size() - 1, -1, -1):
        var entry: Variant = blocks[i]
        if not is_instance_valid(entry):
            blocks.remove_at(i)
            continue
        var b: ColorRect = entry
        b.position.y += fall_speed * delta
        if b.position.y > arena_h:
            blocks.remove_at(i)
            b.queue_free()
        elif invuln_left <= 0.0 and b.get_global_rect().intersects(hero_rect):
            blocks.remove_at(i)
            b.queue_free()
```

```
            _hit()
            if done:
                return


func _hit() -> void:
    lives -= 1
    invuln_left = INVULN_TIME
    _flash_hero()
    _shake(10.0, 0.3)
    _update_label()
    if lives <= 0:
        _end(false)


# Game feel: the hero turns red and fades back to normal while it is safe.
func _flash_hero() -> void:
    if _flash_tween != null and _flash_tween.is_valid():
        _flash_tween.kill()
    hero.modulate = Color(1.0, 0.25, 0.25)
    _flash_tween = create_tween()
    _flash_tween.tween_property(hero, "modulate", Color.WHITE, INVULN_TIME)


# Game feel: shake the whole layer by moving its offset to random spots.
func _shake(strength: float, duration: float) -> void:
    if not shake_enabled:
        return
    if _shake_tween != null and _shake_tween.is_valid():
        _shake_tween.kill()
    offset = Vector2.ZERO
    var steps: int = 6
    var step_time: float = duration / float(steps)
    _shake_tween = create_tween()
    for i in range(steps):
        var amount: float = strength * (1.0 - float(i) / float(steps))
        var jitter: Vector2 = Vector2(randf_range(-amount, amount), randf_range(-amount, amount))
        _shake_tween.tween_property(self, "offset", jitter, step_time)
    _shake_tween.tween_property(self, "offset", Vector2.ZERO, step_time)


func _update_label() -> void:
    label.text = "Lives: %d    Survive: %.1f" % [lives, time_left]


func _arena_size() -> Vector2:
    return get_viewport().get_visible_rect().size


# The ONLY place that ends the fight, and the only place that emits.
func _end(won: bool) -> void:
    if done:
        return
    done = true
    for b in blocks:
        if is_instance_valid(b):
            b.queue_free()
    blocks.clear()
    label.text = "You survived!" if won else "Knocked out..."
    await get_tree().create_timer(END_PAUSE).timeout
    finished.emit(won)
    if get_tree().current_scene == self:
        print("BossFight finished, won = ", won)   # only when run alone with F6


# Safety net: add the movement actions if the project does not have them.
func _ensure_input_actions() -> void:
    for action in ACTION_KEYS:
        if InputMap.has_action(action):
            continue
```

```
        push_warning("Input action '%s' is missing from the Input Map. Adding it for this run." %
action)
        InputMap.add_action(action)
        for key in ACTION_KEYS[action]:
            var ev: InputEventKey = InputEventKey.new()
            ev.physical_keycode = key
            InputMap.action_add_event(action, ev)
```

How the boss script works

It looks long, but it is five small ideas.

1. Movement. `Input.get_vector("move_left", "move_right", "move_up", "move_down")` returns a `Vector2`. Pressing right gives `(1, 0)`; pressing right and down gives a diagonal with length 1 (not 1.41), so diagonal movement is not faster. Multiply by speed and `delta`, add to the position, then `clamp` the result between a minimum and maximum corner so the hero stays on screen. The minimum y is 35 percent of the screen height. That is a fairness rule: if the hero could hide at the very top, blocks would spawn on top of them. Keeping the hero in the lower part means every block gets a chance to be seen before it arrives.

2. Spawning. `spawn_left` counts down. When it reaches zero we create a new `ColorRect` in code with `ColorRect.new()`, set its color, size and position, add it to the `Arena`, and store it in the `blocks` array. Setting `mouse_filter` to Ignore means a block can never steal a click.

3. Falling and hitting. Each frame every block moves down by `fall_speed * delta`. Then we test two things. If the block is below the bottom edge, we remove it. Otherwise, if the player is not invulnerable and the block's rectangle overlaps the hero's, it is a hit. `get_global_rect()` returns a `Rect2` in screen coordinates, and `intersects()` tells us if two rectangles overlap. Both nodes are Controls, so both have `get_global_rect()`.

4. Freeing blocks safely. Look at the loop: `for i in range(blocks.size() - 1, -1, -1)` counts backwards. When you remove an element from an array while looping forwards, you skip the element that slides into its place. Going backwards avoids this. The check `is_instance_valid(entry)` protects against a block that was freed by something else. We read the entry as a `Variant` first, because assigning a freed object straight into a typed variable is itself an error. Every removed block is both removed from the array and `queue_free()`d, so nothing leaks.

5. Ending once. `_end()` is the only exit. A hit that takes the last life calls `_end(false)` from inside the loop; the loop then returns, and `_process` checks `done` before it counts down the clock. That ordering decides a tie in favour of the loss: if the final hit lands on the same frame the timer would have reached zero, the player loses. Change the order if you prefer the opposite.

> **Why it works:** The `lerpf(a, b, progress)` function is the whole difficulty curve. `progress` is 0 at the start and 1 at the end. Block speed slides from 200 to 520 pixels per second, and the gap between blocks shrinks from 0.55 to 0.25 seconds. One line per setting. You will tune these in 17.7.

17.6 The Ceiling, death, and respawn

Remember the result

The boss fight can be won once. We need the game to remember that. Open `res://game/popochiu_globals.gd` (Popochiu created it; it starts with `extends Node`). Under the variables you added in Chapter 15, add:

```gdscript
# res://game/popochiu_globals.gd
extends Node

var key_taken: bool = false
var door_unlocked: bool = false
var boss_defeated: bool = false
var just_died: bool = false
var deaths: int = 0
var bug_caught: bool = false
```

If your file has other lines (the Chapter 15 and 16 variables and `make_flute`, and later the quick-save code from Chapter 18), keep them; the block above shows only the lines that matter here. The first two variables are the Part III naming contract. The other four are new in this chapter:

- `bug_caught` records that the jar has been emptied.
- `boss_defeated` stops the fight from restarting after a win.
- `just_died` tells the Bedroom to play the "waking up" scene.
- `deaths` is a counter you can use later (Chapter 18 uses counters for idle lines).

Why globals and not variables inside the Ceiling script? Because a room and its objects can be rebuilt when you leave and return. Anything the game must remember belongs in `Globals` (Chapter 15).

The Ceiling hotspot

In the Hallway, the Ceiling is a hotspot. If you did not create it in Chapter 14 (Exercise 3), create it now: Room tab, Hotspots, "+", name `Ceiling`, then shape its polygon over the top strip of the background. Open its script and replace the body, keeping the generated header lines.

```gdscript
# res://game/rooms/hallway/hotspots/ceiling/hotspot_ceiling.gd
# The Ceiling hotspot in the Hallway: starts the boss fight.
@tool
extends PopochiuHotspot

const BOSS_FIGHT := preload("res://minigames/boss_fight.tscn")


func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.boss_defeated:
        await C.player.say("The ceiling is quiet now.")
        return
    await C.player.say("The ceiling is looking at me.")

    var won: bool = await _play_boss_fight()

    if won:
        # State first, then the ending.
        Globals.boss_defeated = true
        await C.player.say("It stopped. I think I won?")
        await _play_ending()
    else:
        Globals.deaths += 1
        Globals.just_died = true
        await R.goto_room("Bedroom")


func _on_right_click() -> void:
    if Globals.boss_defeated:
        await C.player.say("A ceiling. It has gone back to being a ceiling.")
    else:
        await C.player.say("The ceiling is looking at me.")
```

```
func _on_item_used(_item: PopochiuInventoryItem) -> void:
    await C.player.say("I am not throwing that at the ceiling. It would only be flattered.")


# The end of the game. The player can keep walking around afterwards.
func _play_ending() -> void:
    await C.Frog.say("Forty years that ceiling has been like that.")
    if I.is_item_in_inventory("Bug"):
        await C.Frog.say("And you kept the bug. Good. Bugs are tenants too.")
    await C.Frog.say("Your rent is waived. Mind the carpet on the way out.")
    await G.show_system_text("THE END. Thank you for staying at the Weird Hotel.")


func _play_boss_fight() -> bool:
    var game: BossFightGame = BOSS_FIGHT.instantiate() as BossFightGame
    G.block()
    Cursor.unblock()
    Cursor.show_cursor("normal")
    get_tree().root.add_child(game)
    var won: bool = await game.finished
    G.unblock()
    game.queue_free()
    return won
```

Notice the order on defeat. We unblock the GUI and free the mini-game **inside** `_play_boss_fight`, before we change room. We set `Globals.just_died = true` **before** `await R.goto_room("Bedroom")`. From the source, `goto_room` fades out, stores the room's state, and asks Godot to change scene; the change happens at the end of that frame, and then this hotspot node is freed. Code after the `await` would run in a room that is about to disappear, and any state it set would miss the room's stored state. So the Ceiling does nothing after changing room, and every fact is set first. The "waking up" lines are the Bedroom's job.

`R.goto_room("Bedroom")` takes the room's script name, exactly as shown in the dock, and fades by default (Chapter 14).

Death should be cheap

Look at what dying does not touch:

- **Inventory.** Items belong to the player character, not to the room, so they stay.
- **Globals.** `key_taken`, `door_unlocked`, and `boss_defeated` are unchanged.
- **Other rooms.** Room state is stored when you leave.

Dying costs only the walk back. That is a deliberate design choice, and it matches the genre: adventure games encourage experiments, and a hard setback teaches players to stop experimenting. A good rule of thumb: dying should cost a little time, never a lot of progress. This is that rule in code.

The Bedroom wakes the player

The respawn point is a **marker**. Open the Bedroom, switch to the Room tab in the dock, find **Markers**, click "+" and name it `Bed`. Drag it to a spot beside the bed.

Open the Bedroom's room script. It has the two functions below already (as empty or `pass` templates, or, for `_on_room_entered`, with the Chapter 14 arrival check). Make them look like this. This is a fragment: only these two functions change; leave everything else in the file as Popochiu generated it. Note that the Chapter 14 `FromHallway` check stays, as an `elif`, so that walking back from the Hallway still places the player at the door.

```
func _on_room_entered() -> void:
    if Globals.just_died:
        await C.player.teleport_to_marker("Bed")
    elif C.player.last_room == "Hallway":
        await C.player.teleport_to_marker("FromHallway")


func _on_room_transition_finished() -> void:
    if Globals.just_died:
        Globals.just_died = false
        await C.player.say("Ow. Everything went white.")
        await C.player.say("I am in bed again. The ceiling can wait.")
```

`_on_room_entered` runs as the room loads, behind the fade curtain, so we move the player there. `_on_room_transition_finished` runs when the fade has finished, so the player sees the speech. We reset `just_died` straight away so a later normal visit to the Bedroom does not repeat it.

> **Try it:** Play to the Hallway (key, door), pick up the Bone, then click the Ceiling and lose on purpose. You wake up beside the bed in the Bedroom. Open the inventory: the Bone is still there. Click the Door: it opens straight away and the Hallway intro does not replay, because `door_unlocked` and `hallway_intro_seen` are unchanged.

17.7 Difficulty tuning and game feel

Tuning

A difficulty curve is a set of numbers, and numbers are tuned by playing. A good method:

1. Change **one** constant.
2. Run the scene alone with F6.
3. Play it five times.
4. Decide: too easy, too hard, or good.

The numbers to tune in the boss fight, with what each does:

Constant	Start value	Raise it and...
`SURVIVE_TIME`	20.0	the fight is longer (the required time in the course design is 20 seconds)
`START_LIVES`	3	the player is more forgiving
`HERO_SPEED`	320	the hero dodges more easily
`FALL_START`, `FALL_END`	200, 520	blocks arrive faster; `FALL_END` controls the ending panic
`SPAWN_START`, `SPAWN_END`	0.55, 0.25	**lower** values mean more blocks
`INVULN_TIME`	1.0	the player has more recovery time after a hit
`TOP_LIMIT`	0.35	the hero is confined to a smaller area at the bottom

A fair game feels hard but possible. A useful test is to ask someone who has never seen the game to try it twice. If they die three times in the first ten seconds, the start is too hard. If they win without moving, it is too easy. Beginners often tune only for themselves, and they have already played the game fifty times.

Why the curve rises over time

If every second were the same, the game would be boring after the first five. A rising curve gives a shape: calm start, tense end. The original course asks for "make the blocks fall faster as time goes on". Our `lerpf(FALL_START, FALL_END, progress)` does exactly that, and also speeds up the spawn rate. You can test the shape by temporarily setting `FALL_START` equal to `FALL_END` and feeling the difference.

A hit flash with a Tween

You met `Tween` in Chapter 9. A tween changes a property smoothly over time. In `_flash_hero` we set the hero's `modulate` (a color that multiplies the node's own color) to red, then tween it back to white over one second:

```
hero.modulate = Color(1.0, 0.25, 0.25)
_flash_tween = create_tween()
_flash_tween.tween_property(hero, "modulate", Color.WHITE, INVULN_TIME)
```

The red fade doubles as a visual timer for the invulnerability window: when the hero looks normal again, they can be hit again. Good game feel often carries information, not just decoration.

We store the tween in `_flash_tween` and kill any running one first. Without that, two tweens could fight over the same property. The `is_valid()` check tells us whether a stored tween is still alive.

A screen shake

A shake is a few quick, random moves that fade out. A `CanvasLayer` has an `offset` property that shifts everything on it, so we tween the offset through six random positions with smaller and smaller amounts and finish at zero. The Backdrop is 32 pixels bigger than the screen on every side so the edges never show a gap.

> **Note:** Popochiu has its own camera shake, `E.camera.shake(strength, duration)` (strength is a distance in pixels), which you will use in Chapter 18 for the room. It moves the camera. A `CanvasLayer` does not follow the camera, so that shake does not move the mini-game. That is why the boss fight shakes its own layer.

Design note: effects are cheap, so use few

A flash and a shake on every hit is plenty. If you add a screen flash, particles, a sound, and a slow-motion effect to every event, the player cannot tell what matters. Choose one or two effects for each important event. Chapter 18 returns to this with accessibility: some players need to switch shaking off, which is why `shake_enabled` already exists.

17.8 Testing alone, and writing robust code

F6: run the current scene

F5 runs the whole project. **F6 runs the scene you currently have open**, and nothing else. Open `catch_bugs.tscn` and press F6. The game starts at once, with no hotel behind it. Do the same for `boss_fight.tscn`.

Testing in isolation answers an important question: is a bug in the mini-game, or in the way the adventure uses it? If the scene misbehaves alone, the mini-game is wrong. If it works alone and fails in the hotel, the border (blocking, layers, freeing) is wrong.

When you run a mini-game alone, nothing frees it. After "Got it!" the scene just stays on screen, and the print line in `_end` shows the result in the Output panel. That is the intended behaviour.

What to check each time

For **catch_bugs.tscn**:

Check	Expected
Press F6	The overlay appears; a BUG button is somewhere on screen
Click it 5 times quickly	"Got it!" then Output prints `CatchBugs finished, won = true`
Do nothing for 10 seconds	"Too slow!" then Output prints `... won = false`
Click the button rapidly after the end	Nothing changes; the message prints once

For **boss_fight.tscn**:

Check	Expected
Press F6	The yellow hero stands near the bottom; red blocks begin to fall
Press W A S D and arrow keys	The hero moves in all four directions and stays on screen
Try to reach the top of the screen	The hero stops about a third down
Let three blocks hit	The hero flashes red, the screen shakes, "Knocked out..." appears, Output prints `won = false`
Survive 20 seconds	"You survived!" and `won = true`

Robust code checklist

Review every mini-game against this list before you call it finished.

1. **One exit.** Is there exactly one function that emits `finished`?
2. **A guard.** Does that function begin with `if done: return`?
3. **Always ends.** Is there a timer or a lose condition so the game cannot last forever?
4. **No Popochiu inside.** Does the script avoid `C`, `I`, `G`, `E`, `R`? (It stays testable alone.)
5. **Free what you create.** Blocks, timers and tweens created in code: are they freed or bound to a node that is freed?
6. **Check before use.** Where an object might have been freed, do you call `is_instance_valid()`?
7. **The caller cleans up.** Does the caller unblock, then free?
8. **Input exists.** Are the Input Map actions present?

The mini-game scripts in this chapter satisfy all eight.

> **Try it:** Break the contract on purpose to see what happens. In `catch_bugs.gd`, temporarily comment out the two lines `if done:` and `return` at the top of `_end`. Add two temporary lines at the end of `_ready`: `_end(true)` and `_end(false)`. Press F6 and watch the Output panel: `finished` is emitted twice, with different results. Now restore the guard and run again: it prints once, and the first result wins. Remove the two temporary lines. This is why guards matter even when you believe a path cannot happen.

Lab 17: Both mini-games working in Weird Hotel

Time: 60 to 90 minutes. **Needs:** your Weird Hotel project from Chapter 16, with Bedroom, Hallway and the Frog working. The Jar prop, the Ceiling hotspot and the item Bug are created in this lab if you do not have them.

1. In the Popochiu dock, Main tab, Inventory items, create the item `Bug` if it does not exist. Do not tick "starts in inventory".
2. In the Hallway (Room tab), check that the Jar prop and Ceiling hotspot exist; if you skipped Chapter 14 Exercise 3, create them now (Sections 17.4 and 17.6). Give each a clickable area.
3. Create the folder `res://minigames/`.
4. Build `catch_bugs.tscn` as in 17.4, with the exact node names. Save the script as `catch_bugs.gd`.
5. Open `catch_bugs.tscn` and press F6. Win once and lose once. Both runs must print a result in Output.
6. Add the four Input Map actions as in 17.5.
7. Build `boss_fight.tscn` and `boss_fight.gd` as in 17.5. Press F6. Verify movement, the top limit, a loss after three hits, and a win after 20 seconds.
8. Open `res://game/popochiu_globals.gd` and add `boss_defeated`, `just_died`, `deaths` and `bug_caught` below the variables you already have.
9. Replace the Jar prop's script body with the Jar code from 17.4.
10. Replace the Ceiling hotspot's script body with the Ceiling code from 17.6.
11. In the Bedroom, create a marker named `Bed` and update `_on_room_entered` and `_on_room_transition_finished` as in 17.6.
12. Press F5. Play: take the Key, unlock the Door, go to the Hallway, click the Jar and win. Check the inventory.
13. Click the Jar again: "The jar is empty now."
14. Click the Ceiling and lose on purpose. You must wake up in the Bedroom, with the Bug (and any Bone or Twine you picked up) still in the inventory. Walk back through the Door to the Hallway; the intro must not replay.
15. Click the Ceiling again and win. The Frog speaks the ending lines (mentioning the Bug) and the box "THE END" appears; click to close it. Click the Ceiling a third time: "The ceiling is quiet now."
16. Commit: `git add -A`, `git commit -m "Chapter 17: jar and boss mini-games"`.

Expected result: Two mini-games run on top of the hotel. While one runs, clicking on the room behind it does nothing, and the pointer looks normal. Winning the jar game gives the Bug; losing lets you retry. Winning the boss fight is remembered and plays the ending; losing sends you to the Bedroom with all items and puzzle progress intact. Each mini-game also runs correctly alone with F6.

If something does not work, check section "Common mistakes" at the end of this chapter first.

Chapter summary

- A mini-game is an ordinary Godot scene placed on top of the adventure. It needs no Popochiu code, so it can be tested alone with F6.
- The contract pattern: the scene declares `signal finished(won: bool)`, emits it exactly once, and the caller waits with `var won: bool = await game.finished`.
- Use a `CanvasLayer` with layer 50: above the Popochiu GUI (10) and transitions (15), below the cursor (100).
- Call `G.block()` before adding the mini-game and `G.unblock()` right after the `await`. Do not unblock manually inside `E.queue`.
- The caller creates, awaits, unblocks and frees the scene. The mini-game has a single `_end()` with a `done` guard.
- Input Map actions (`move_left`, `move_right`, `move_up`, `move_down`) let the code ask for intent, not keys, and `Input.get_vector` turns four actions into a direction.
- A rising difficulty curve is built with `lerpf` and a `progress` value from 0 to 1. Game feel comes from small effects such as a `Tween` flash and a shaken `CanvasLayer.offset`.
- Death sends the player to a safe room. Items live on the player and progress lives in `Globals`, so little is lost; the Bedroom script handles the wake-up.

Review questions

1. Give two reasons to build a mini-game as a plain Godot scene instead of using Popochiu rooms and props.
2. Complete the contract: the mini-game emits `finished(won)` _____ time(s), and the _____ frees the scene.
3. What does `var won: bool = await game.finished` return when the signal has one argument?
4. Popochiu's GUI is on layer 10, transitions on 15 and the cursor on 100. Which layer number is the best choice for a mini-game, and why?
5. True or false: after `await game.finished` you must call `G.unblock()` even if the mini-game was lost.
6. What is the advantage of using an Input Map action such as `move_left` instead of checking the A key directly?
7. In `_move_blocks`, why does the loop count backwards from the last index?
8. Multiple choice. Which statement about `E.camera.shake` and the boss fight is correct? (a) It shakes the boss fight automatically. (b) It does not move a CanvasLayer, so the boss fight shakes its own `offset`. (c) It only works when the game is paused. (d) It requires `G.block()`.
9. After the player loses the boss fight, which of these are lost: inventory items, `Globals.door_unlocked`, the position of the player? Explain.
10. Why does the Ceiling script set `Globals.just_died = true` before `await R.goto_room("Bedroom")` and not after?

Exercises

1. **Practice (easy): retune the jar.** In `catch_bugs.gd`, change `NEEDED`, `TIME_LIMIT`, `START_HOP` and `MIN_HOP` to make the game easy for a child, then very hard. Play each version with F6. Write one sentence on how each constant changes the feel. You should not need to edit the Jar script.

2. **Apply (medium): one launch helper.** The Jar and the Ceiling each have a similar `_play_...` function. Write a small helper in `Globals` (or a new autoload) such as `func run_minigame(scene: PackedScene) -> bool` that instantiates the scene, blocks, adds it to the root, awaits `finished`, unblocks, frees, and returns the result. Hint: the mini-games share only the `finished` signal and no common class, so declare the variable without a type (`var game = scene.instantiate()`); GDScript then looks up `finished` at run time. You lose the editor's type checking, which is the price of a generic helper. Update both scripts and verify the lab still works.
3. **Apply (medium): a boss that gets angry.** Change the boss fight so that blocks near the end are wider as well as faster: scale `BLOCK_SIZE.x` with `progress`. Remember that a block size is set when the block is created. Play-test and make sure the last five seconds feel hard but not unfair.
4. **Challenge (hard): the weak spot.** Add a stretch idea from the original course: every few seconds a yellow "weak spot" button appears at a random place in the arena for one second. Clicking it counts as a hit on the boss: if the player clicks it three times, they win early. Hints: create a `Button` in code with `Button.new()`; connect its `pressed` signal; give it a `Timer`-like countdown with `create_tween().tween_interval(1.0)` and `tween_callback`; keep a counter of weak-spot hits and call `_end(true)` when it reaches three. Keep the single exit and the `done` guard. The Arena has Mouse Filter Ignore, but the Backdrop is Stop, and your new button is a Control with its own filter, so the click will reach it.
5. **Challenge (hard): your own mini-game.** Invent a third mini-game that follows the contract and attach it to a new prop in Weird Hotel (for example, a lock that needs a timing click, a "match the pattern" game, or a falling-fruit catcher). Requirements: root `CanvasLayer` on layer 50; `signal finished(won: bool)`; one `_end()` with a guard; a time or lose limit; it runs alone with F6; the caller uses `G.block()` and `G.unblock()`. Write a short design note: what the player does, how they win, how they lose, and what reward the win gives.

Common mistakes

- **Symptom:** the player can still click props behind the mini-game. **Cause:** `G.block()` was not called, and the background node has Mouse Filter Pass or Ignore. **Fix:** call `G.block()` before adding the scene, and set the full-screen background's Mouse Filter to Stop.
- **Symptom:** after the mini-game the game seems frozen; nothing can be clicked. **Cause:** `G.unblock()` was never reached, usually because the mini-game never emitted `finished` or you freed it early. **Fix:** make `_end()` the only exit, never free the scene before `finished`, and put `G.unblock()` on the line after the `await`.
- **Symptom:** the pointer turns into the "wait" picture during a mini-game. **Cause:** `G.block()` makes the Simple Click GUI show the wait cursor. **Fix:** call `Cursor.unblock()` and `Cursor.show_cursor("normal")` right after `G.block()`.
- **Symptom:** the Output panel shows "Invalid call. Nonexistent function" or "Invalid access to property or key 'finished'". **Cause:** the node you instantiated is not the root with the script (wrong scene path) or `class_name` casting failed. **Fix:** check the `preload` path and that `catch_bugs.gd` is attached to the root node.
- **Symptom:** "The action 'move_left' doesn't exist" errors. **Cause:** the Input Map actions were not added or were misspelled. **Fix:** Project > Project Settings > InputMap, add the four names exactly in lower case with underscores.
- **Symptom:** Output prints "Resumed function after await, but class instance is gone". **Cause:** something freed the mini-game while its `_end()` was waiting for the end-pause timer. **Fix:** only the caller frees the scene, only after `finished`.

- **Symptom:** the screen shows a gap or the shaking exposes the room at the edge. **Cause:** the Backdrop is exactly screen-sized. **Fix:** make it 32 pixels larger on every side with negative top-left offsets.
- **Symptom:** after losing the boss fight the player appears in the Hallway or the Bedroom lines never play. **Cause:** `Globals.just_died` was not declared, was set after `goto_room`, or the marker `Bed` is missing. **Fix:** declare it in `popochiu_globals.gd`, set it before the `await R.goto_room(...)`, and create the marker.

Further reading

- Godot docs, Canvas layers: https://docs.godotengine.org/en/stable/tutorials/2d/canvas_layers.html
- Godot docs, CanvasLayer class: https://docs.godotengine.org/en/stable/classes/class_canvaslayer.html
- Godot docs, GDScript basics (the `await` section): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
- Godot docs, Input examples: https://docs.godotengine.org/en/stable/tutorials/inputs/input_examples.html
- Godot docs, Tween class: https://docs.godotengine.org/en/stable/classes/class_tween.html
- Popochiu documentation: https://carenalgas.github.io/popochiu/
- Popochiu on GitHub (source code of the singletons used here): https://github.com/carenalgas/popochiu

Chapter 18: Saving, Polish, Testing and Export

Learning objectives

By the end of this chapter you will be able to:

- Save and load a Popochiu game with `E.save_game` and `E.load_game`, and explain what is saved automatically and what you must store yourself.
- Keep custom data safe across saves using `Globals`, state scripts and the `_on_restore_from_savegame` hook.
- Add music and sound effects through Popochiu's audio cues and the `A` singleton, and add a transition (`T`) and a camera shake (`E.camera.shake`).
- Write escalating "idle lines" for repeated clicks with a counter, and add accessibility options that persist in a settings file.
- Plan and run a playtest with real players, record what happens, and triage the results into a fix list.
- Export Weird Hotel as a Windows program, run a release checklist, and credit the licenses of Godot, Popochiu and your assets correctly.

Key terms

- **Save slot**: a numbered place where one saved game is stored. Popochiu uses slot numbers such as 1.
- `user://`: Godot's path to a per-user, writable folder where saves and settings go. It differs from `res://`, which is read-only in an exported game.
- **Serialization**: turning game data into text or bytes that can be stored, and back.
- **Audio cue**: a named, ready-to-play sound entry Popochiu creates from an audio file, such as `A.sfx_door`.
- **Transition**: an effect, such as a fade, that hides a change of scene.
- **Idle line**: a line of dialogue that changes when the player repeats the same action.
- **Playtest**: a session where a real person plays your game while you watch, without helping.
- **Think-aloud**: asking players to say what they are thinking while they play.
- **Triage**: sorting problems by how serious they are, to decide what to fix first.
- **Export preset**: saved export settings for one platform (for example Windows Desktop).
- **Export template**: the prebuilt engine binary Godot copies into your exported game. It must match your editor's version exactly.
- **Release candidate**: a build you believe is ready to ship, tested one last time.

18.1 Saving and loading in Popochiu

A save file lets the player stop and come back. In a plain Godot game (Chapter 10) you chose what to save and wrote the code. Popochiu does most of that for you, but you need to know exactly what it covers, because the rest is your job.

The two functions

```
# Signatures only (what each function accepts), not lines to type.
E.save_game(slot: int = 1, description: String = "")
E.load_game(slot: int = 1)
```

`E` is the Popochiu engine singleton. Both functions take a slot number (default 1) and return nothing. Saves go to files in `user://`, named `save_<slot>.json` (for example `user://save_1.json`). **Slots 1 to 4** are the ones Popochiu lists: the helpers below only look at those four files. Saving to slot 7 would write a file, but no menu or helper would find it.

Call	What it does
`E.save_game(slot := 1, description := "")`	Writes the file, then emits `E.game_saved`. The description is the text a save menu shows.
`E.load_game(slot := 1)`	Empties the inventories, stops any running dialog, reads the file, and changes to the saved room. It does nothing if the file is missing. The room change happens after the call returns, so do not expect the new room to be ready on the next line.
`E.has_save()`	`true` if **any** of the four slot files exists. It has no slot argument. To test one slot, check `E.get_saves_descriptions().has(1)`.
`E.saves_count()`	How many slot files exist.
`E.get_saves_descriptions()`	A `Dictionary` of slot number to description.

Signals: `E.game_saved`, `E.game_load_started` (a load has begun) and `E.game_loaded(data)` (the data of the file, once the room is ready).

> **Note:** The Popochiu docs page on saving and loading was still a stub when this book was written, so these facts come from the Popochiu source code (version 2.2.1), file `engine/others/popochiu_save_load.gd`.

The Simple Click GUI already contains a Save and a Load button in its settings popup (the gear at the top right). They call these same functions, so you get a save menu for free.

What Popochiu saves for you

From the source, a Popochiu save includes:

Saved automatically	Notes
The room the player is in	On load you land in that room
Every character's inventory	Items belong to characters since Popochiu 2.2
Room state, character state, inventory item state, dialog state	The `*_state.gd` scripts, with the variables you declare in them
Statically typed simple variables (`bool`, `int`, `float`, `String`, `Array`) of props, hotspots, regions and walkable areas, and the position, facing and visibility of characters	For example a prop's `alpha` and `current_frame`, a region's `enabled`, or a `var x: bool` you declared in a prop script. **Not** saved: whether a prop or hotspot is hidden or clickable, where it stands, untyped variables, and anything in a prop that is disabled (Chapter 15's `_apply_state()` exists for this)
Room flags `visited` and `visited_times`	Useful for "first time here" lines

Saved automatically	Notes
The player character's name and position	You load back where you stood
Each dialog's options (visible, disabled, used)	The Frog's menu comes back as it was
The simple variables in `Globals`	`bool`, `int`, `float`, `String`, and plain arrays. **Not** dictionaries (see 18.2)

The last row is why Chapter 15 told you to keep puzzle flags in `Globals`: `key_taken`, `door_unlocked`, `boss_defeated` and `just_died` are all `bool` values, so they are saved with no extra work.

The save file is one JSON object. Its top-level keys are `description`, `player` (with `room`, `id`, `position` and a legacy `inventory` list), `rooms`, `characters`, `inventory_items`, `dialogs`, `globals` and `character_inventories` (a dictionary from character name to a list of `{name, qty}` entries). A key is left out when it would be empty. Items that are *not* in `character_inventories` are not owned after a load.

What you must handle yourself

Not saved for you	What to do
Complex types in `Globals` (a `Color`, a `Vector2`, a `Dictionary`, a custom `Resource`)	Write `on_save()` and `on_load(data)` in `Globals`
Extra data in a room, character or item state that is not a simple type	Write `_on_save()` and `_on_load(data)` in that state script
Anything a mini-game keeps (the mini-game is gone after it finishes)	Report it through the result and store it in `Globals`
Player settings (volume, shake on/off)	Do not put them in save games; use a settings file (18.5)
Things that were started once and keep running (music, timers)	Restart them in `_on_restore_from_savegame` (18.2)

A quick save and quick load

The Popochiu GUI template you chose in Chapter 13 may include its own save and load menu. It is useful to have a plain keyboard shortcut too, so you can test saving without going through menus. We add it to `Globals`, which is an autoload (it exists for the whole game) and extends `Node`, so it can handle input.

First add two Input Map actions (Project > Project Settings > InputMap): `quick_save` bound to **F9**, and `quick_load` bound to **F10**.

Now add this function to `res://game/popochiu_globals.gd`, below the variables. Keep every variable and function that is already there.

```
# Fragment: add to res://game/popochiu_globals.gd
func _unhandled_input(event: InputEvent) -> void:
    if G.is_blocked:
        return                          # no saving during mini-games or cutscenes
    if event.is_action_pressed("quick_save"):
        E.save_game(1, "Quick save")
        print("Saved to slot 1")
    elif event.is_action_pressed("quick_load"):
        if E.has_save():
            E.load_game(1)
```

```
        else:
            print("No save to load")
```

The first line is the important one. While `G.is_blocked` is true (a mini-game, a cutscene, a dialog queue), we refuse to save. Saving in the middle of the boss fight would store the Hallway but not the falling blocks, so loading would drop the player into the hallway with no mini-game and a half-finished state. Blocking the save is the simplest safe design.

> **Note:** `E.has_save()` has no slot argument; it means "is there any save in slots 1 to 4?". Since this code only uses slot 1, that is close enough. (Simple Click also shows "Game saved" in a box when `E.save_game` succeeds, so F9 gives feedback without your `print`.)

Test a save

A save you have not loaded is a save you do not know works. Use this protocol every time you add state.

1. Run the game (F5). Take the Key from the drawer, use it on the Door, walk to the Hallway, and pick up the Bone.
2. Press **F9**. Output shows "Saved to slot 1".
3. Close the game window.
4. Run the game again. You start in the Bedroom, the Door is locked, and the inventory is empty.
5. Press **F10**. You should land in the Hallway with the Bone in the inventory, the bone pile gone, and no intro cutscene.
6. Walk back to the Bedroom and try the drawer. It must not offer the key again (`key_taken` was restored).

To see the file itself, use **Project > Open User Data Folder**. Open `save_1.json` in a text editor. It is JSON text, readable by people. Search for `key_taken`; you should find it set to `true`. Seeing your own variable in the file is a good way to learn what is really stored. The layout can change between Popochiu versions, so do not write code that reads the file; use `E.load_game`.

> **Warning:** A load does **not** call `_on_room_transition_finished()`. Anything you do in that function (a "waking up" speech, starting music) will not happen after a load. If it must happen after a load too, put it in `_on_restore_from_savegame()` as well.
>
> **Try it:** Take the Key, save with F9, then use the Key on the Door so `door_unlocked` becomes true. Now press F10. The Door must be locked again, and the Key back in your inventory, because the save was made before. If it is not, `door_unlocked` is probably stored somewhere that is not saved (for example a variable inside a script rather than `Globals`).

18.2 Making your own data save-safe

Safe types in Globals

A variable in `Globals` with a simple type (`bool`, `int`, `float`, `String`, a plain `Array`) is saved automatically, but only if the type is written in the declaration (`: bool` or `:=`). Popochiu's source checks the declared type, so an untyped `var deaths = 0` is skipped. Always declare the type:

```
var deaths: int = 0
var best_boss_time: float = 0.0
```

```
var seen_intro: bool = false
```

Complex types: on_save and on_load

For anything else, Popochiu gives you two public hooks in `Globals`. Note that these two have **no underscore** at the start, unlike the hooks in rooms and states.

```
# Fragment: add to res://game/popochiu_globals.gd
var lamp_color: Color = Color.WHITE   # a Color is not a "safe" type

func on_save() -> Dictionary:
    # Return a Dictionary of simple values; a Color becomes its HTML text.
    return {"lamp_color": lamp_color.to_html()}


func on_load(data: Dictionary) -> void:
    if data.has("lamp_color"):
        lamp_color = Color(String(data["lamp_color"]))
```

The pattern is "convert to simple data on the way out, rebuild on the way in". Always check `data.has(...)` on the way in. An old save made before you added the variable will not contain it, and without the check the game would crash or reset the value to nonsense.

> **Note:** What `on_load` receives is exactly what `on_save` returned (stored in the file under `globals`, key `custom_data`); the automatic values are restored separately. JSON turns every number into a float, so convert with `int(...)` where you need a whole number. Dictionaries are *not* saved automatically (only the types listed above are), which matters in Section 18.4.

State scripts for rooms, characters and items

Each room, character and inventory item has a state script (`room_hallway_state.gd`, and so on). Variables you declare there are saved with the object. In code you reach them through `R.Hallway.state`, `C.Frog.state` or `I.Key.state`. If a state needs a complex type, use `_on_save() -> Dictionary` and `_on_load(data: Dictionary)` there; the shape is the same as in `Globals`.

Use state scripts for facts that belong to one object ("this lamp is on") and `Globals` for facts that belong to the whole game ("the door is unlocked").

Every Globals variable in Weird Hotel

Chapter 15 promised one list of all the facts the game keeps in `Globals`. Here it is. Your `popochiu_globals.gd` should declare exactly these (plus `make_flute`, `next_line`, `on_save`, `on_load` and `_unhandled_input`).

Variable	Type	Added in	Meaning
`key_taken`	`bool`	15	The drawer has been emptied
`door_unlocked`	`bool`	15	The Bedroom door is open
`bone_taken`	`bool`	15	The bone pile has been picked up
`twine_taken`	`bool`	15	The ball of twine has been picked up
`hallway_intro_seen`	`bool`	16	The Hallway intro has played

Variable	Type	Added in	Meaning
`frog_met`	`bool`	16	The player has greeted the Frog
`frog_who_asked`	`bool`	16	"Who are you?" has been asked
`flute_given`	`bool`	16	The flute has been traded for the spare Key
`bug_caught`	`bool`	17	The jar is empty; the Bug was won
`boss_defeated`	`bool`	17	The ceiling fight was won (the ending has played)
`just_died`	`bool`	17	The Bedroom should play the waking-up lines
`deaths`	`int`	17	How many times the player lost the boss fight
`click_counts`	`Dictionary`	18	Idle-line counters (saved through `on_save`)

All but the last are saved automatically.

_on_restore_from_savegame

After a load, Popochiu restores the saved values but does not replay what the room did when you first walked in. The room script has a hook for that: `_on_restore_from_savegame()`. Use it to rebuild anything that is not saved by itself, like running music:

```gdscript
# Fragment: in the Hallway room script (room_hallway.gd).
# Add the _start_music() call to the two functions you already have.
func _on_room_entered() -> void:
    if C.player.last_room == "Bedroom":
        await C.player.teleport_to_marker("FromBedroom")
    _apply_state()
    _start_music()


func _on_restore_from_savegame() -> void:
    _apply_state()
    _start_music()


func _start_music() -> void:
    if not A.mx_hallway.is_playing():
        A.mx_hallway.play()
```

The guard `is_playing()` is a safety habit; Popochiu itself ignores a request to play a music cue that is already playing. Cue `mx_hallway` is created in the next section.

18.3 Audio in Popochiu

Chapter 9 taught Godot's audio: `AudioStreamPlayer`, buses, volume in decibels. Popochiu wraps all that so that you play sounds by name, with no nodes to create.

Files and cues

1. Put your audio files in a folder **inside** `res://game/`, for example `res://game/audio/`. The Audio tab only scans `res://game/` and its subfolders; files elsewhere (such as `res://audio/`) never appear. Use **OGG** for music (small files) and **WAV** or OGG for short effects. Popochiu accepts `ogg`, `mp3`, `wav` and `opus`.

2. **Name the files with a prefix.** The default prefixes (Project Settings, `popochiu/audio/...`) decide the group:

Prefix	Group	Example file
`mx_`	Music	`mx_hallway.ogg`
`sfx_`	Sound effects	`sfx_pickup.wav`
`vo_`	Voices	`vo_frog_01.ogg`
`ui_`	Graphic interface	`ui_click.wav`

3. Open the **Audio** tab of the Popochiu dock and click **Scan audio files**. Each file with a known prefix gets an **audio cue** (a `.tres` file saved next to the audio file) automatically. A file without a known prefix goes to the **Unassigned** group; open its row menu and choose the group to add it to.
4. When a cue exists, it becomes a property of the `A` singleton. The name is the file name without the extension, in lower case: `A.mx_hallway`, `A.sfx_pickup`.
5. Music cues are created with **Loop** switched on, and Popochiu applies it to the stream, so you do not need to change the import settings. Sound-effect cues do not loop.

> **Note:** Where to get sounds: make your own with a free sound generator, record your own, or use sounds with an open license, for example from opengameart.org, freesound.org or kenney.nl. Always record the author and license for the credits file (18.8). A good rule: nothing goes into the game if you do not know its license.

Playing sounds and music

```
# sound effect: play(wait_to_end := false, position_2d := Vector2.ZERO)
A.sfx_pickup.play()
await A.sfx_door.play(true)      # true = wait until the sound has finished

# music: play(fade_duration := 0.0, music_position := 0.0)
A.mx_hallway.play()
A.mx_hallway.play(2.0)           # fade in over 2 seconds

# stopping
A.mx_hallway.stop(1.5)           # fade out over 1.5 seconds
A.mx_hallway.is_playing()        # true or false
```

Music and sound effects are different things in practice:

	Music	Sound effect
Length	Minutes, loops	Under two seconds
How many at once	One track (a new track replaces the old)	Several can overlap
When to start	When a room is entered, or an event	On a click, a pickup, a hit
Waiting	Almost never	Sometimes (`play(true)`)
Loop	On (the music cue sets it)	Off

Where to put them in Weird Hotel

Add these three lines. The first goes in the Jar's script, the others in the Ceiling's. Both are fragments, to place inside the functions from Chapter 17.

```
# Fragment: Jar _on_click, in the "if won:" branch, before the speech
A.sfx_pickup.play()

# Fragment: Ceiling _on_click, inside "if won:" and "else:" respectively
A.sfx_win.play()
A.sfx_lose.play()
```

The mini-games stay silent on purpose. They report a result and the caller reacts, so the sound effect lives with the adventure. If you want sounds inside the mini-game (a "pop" when you hit the bug), add an ordinary `AudioStreamPlayer` node to the mini-game scene. That keeps the mini-game independent from Popochiu.

> **Warning:** If a cue name is wrong, you get a "nonexistent property" error on `A`, like a typo in any other singleton. The cue name is the file name without the extension, in lower case. Create the cue (scan) before you call it.
>
> **Try it:** Find or make three short sounds (`sfx_pickup.wav`, `sfx_win.wav`, `sfx_lose.wav`) and one loop (`mx_hallway.ogg`). Create the cues and play them in the places above. Press F5 and listen. Then press F9, quit, run, press F10: the music should start again in the Hallway because of `_on_restore_from_savegame`.

18.4 Transitions, camera shake, and idle lines

Transitions with T

`R.goto_room("Hallway")` already fades between rooms. The `T` singleton lets you play a transition yourself:

```
await T.play_transition("fade", 1.0, T.PLAY_MODE.IN_OUT)
```

The arguments are the transition name, the duration in seconds, the mode and an optional color (`T.PLAY_MODE` is the same enum as `PopochiuTransitionLayer.PLAY_MODE`). Names are matched in snake case, so `"Fade"` and `"fade"` both work. The built-in transitions in Popochiu 2.2.1 are `fade`, `fog`, `none`, `sliding_doors`, `tunnel`, `wipe_bottom`, `wipe_left`, `wipe_right` and `wipe_up`; Project Settings > `popochiu/gui/transition_layer/` chooses the default, its duration (1.0 s) and color. `T.show_curtain(color)` and `T.hide_curtain()` cover and uncover the screen directly, without animation.

What the modes do, from the source:

Mode	Animation	Afterwards	Used by `goto_room` for
`OUT`	plays backwards	the curtain stays on screen	leaving a room (cover)
`IN`	plays forwards	the curtain is hidden	entering a room (reveal)
`IN_OUT`	forwards, then backwards	the curtain is hidden	(the default for skipped cutscenes)

A good use: a white cover before the boss fight, so the arena appears as a "cut" and not a pop. Cover the room with `OUT`, and uncover it once the mini-game (on layer 50, above the curtain on layer 15) is on screen. Two fragments for the Ceiling script:

```
# Fragment: Ceiling _on_click, before: var won: bool = await _play_boss_fight()
await T.play_transition("fade", 0.4, T.PLAY_MODE.OUT, Color.WHITE)

# Fragment: _play_boss_fight, right after: get_tree().root.add_child(game)
T.hide_curtain()
```

Test it: the screen should turn white, then the arena appears and, when the fight ends, the room is visible again.

Camera shake with E.camera

Popochiu's camera can shake the whole room:

```
# Signatures only (what each function accepts), not lines to type.
E.camera.shake(strength: float = 1.0, duration: float = 1.0)   # waits for duration
E.camera.shake_bg(strength: float = 1.0, duration: float = 1.0)  # does not wait
E.camera.stop_shake()
```

`strength` is a distance in game pixels: each frame the view jumps by up to that many pixels. `1.0` is barely visible; at 1080p try 6 to 12.

For example, the ceiling could rumble before the fight. Remember the accessibility option from Chapter 17 and read it from the settings autoload that you create in 18.5:

```
# Fragment: Ceiling _on_click, after the first speech line
if GameSettings.screen_shake:
    await E.camera.shake(8.0, 0.8)
else:
    await E.wait(0.8)
```

`shake` already waits for its duration, so the `else` keeps the same pause for players who switched shaking off. `E.wait(seconds)` pauses the script without freezing the game. As Chapter 17 explained, this shake moves the camera, so it does not move the mini-games. They shake themselves.

Idle lines: when the player keeps clicking

If the player clicks the locked Door ten times and hears the same line ten times, the game feels dead. A character with something new to say each time, ending on a joke, makes the world feel alive. This is cheap to do: count the clicks and choose a line from a list.

Add a dictionary and a helper function to `Globals`:

```
# Fragment: add to res://game/popochiu_globals.gd
# A Dictionary is NOT saved automatically, so on_save/on_load below store it.
var click_counts: Dictionary = {}


# Returns the next line in the list. After the last line it repeats the last one.
func next_line(id: String, lines: Array) -> String:
    var n: int = int(click_counts.get(id, 0))
    click_counts[id] = n + 1
    return str(lines[mini(n, lines.size() - 1)])


# Called by Popochiu when the game is saved. Return simple data only.
func on_save() -> Dictionary:
    return {"click_counts": click_counts}
```

```
# Called by Popochiu when a game is loaded, with what on_save returned.
func on_load(data: Dictionary) -> void:
    if data.has("click_counts"):
        click_counts = data["click_counts"]
```

How it works. `click_counts.get(id, 0)` returns the stored count, or 0 if this `id` has never been used. `int(...)` makes sure it is a whole number (numbers read back from a JSON save come out as floats). We increase the count, then pick the line at index `n`, but never beyond the last one: `mini(a, b)` returns the smaller of two integers. Because the dictionary lives in `Globals`, the count survives room changes. Popochiu only saves `Globals` variables of type `bool`, `int`, `float`, `String` and `Array`, so the dictionary needs `on_save()` and `on_load()` (Section 18.2) to survive a save. If you already wrote these two functions for `lamp_color`, put both keys in one `on_save()` and one `on_load()`; a script cannot have two functions with the same name.

Use it in the Door hotspot, in the part of `_on_click` that says the door is locked. Replace only that line:

```
# Fragment: Door _on_click, where the player says the door is locked
await C.player.say(Globals.next_line("Door", [
    "It is locked.",
    "Still locked.",
    "I keep checking. It keeps being locked.",
    "The door and I have reached an understanding. It stays locked.",
]))
```

Idle lines are a good place for personality. Three guidelines: make line two different from line one; make the third line the first joke; stop at the end on a line that works if repeated forever.

> **Try it:** Click the locked Door five times. You should see the four lines in order and then the fourth line again. Save and reload: the next click continues the sequence where it was, because `on_save` stored `click_counts`. Remove `on_save` and `on_load` for a moment and try again: after the load, the sequence starts over.

18.5 Accessibility

Accessibility means that as many people as possible can play your game. It is not an extra for a later version; many fixes cost nothing now and are painful later.

Principles and quick wins for Weird Hotel

Area	Problem for some players	What to do
Color	About 1 in 12 men and 1 in 200 women have some color vision difference.	Never use color as the only signal. In the boss fight, the hero is a bright yellow, the blocks a dark red, and they also differ in size. If you add a "danger" cue, add a shape or a sound too. Check contrast between hero and blocks by converting a screenshot to black and white; both should still be easy to tell apart.
Text	Small or fast text is hard to read.	Use a large default font size in dialog (24 pixels or more at 1080p). Do not rely on speech sounds alone: all dialogue is text.

Area	Problem for some players	What to do
Speed	Some players read slowly.	In Popochiu a line stays until the player clicks (the first click shows the whole line, the second closes it). The Simple Click settings popup (gear, top right) has **Dialogue Speed** and **Dialogue Advancement** controls and a **Volume** control. Behind them: Project Settings `popochiu/dialogs/text_speed` (seconds per letter, default 0.1, lower is faster) and, in code, `E.text_speed`. Popochiu stores the volumes itself in `user://audio_settings.save`; the text speed is not stored between runs.
Motion	Shakes and flashes can cause discomfort.	Offer an option to turn shaking off, and keep flashes short and not repeated more than three times a second.
Timing	Action mini-games exclude some players.	Offer a way to skip or to make it easy (a bonus exercise). The adventure should be finishable even if the boss fight is hard.
Input	Not everyone uses WASD.	Keep keys in the Input Map (you did) and add arrow keys (you did). Consider adding controller buttons.
Sound	Deaf and hard-of-hearing players.	Important events must also appear as text or visuals. Do not make a puzzle depend on a sound alone.

A settings file

Settings (shake on or off, and later volume or text size) belong to the player and not to the saved game. If they were part of a save, loading an old save would silently switch the player's settings back. And any simple variable in `Globals` is saved with the game. So we keep settings in their own autoload and their own file, using `ConfigFile` from Chapter 10.

1. Create a script `res://settings.gd`:

```
# res://settings.gd
# Autoload named GameSettings. Player options, stored outside the save games.
extends Node

const PATH: String = "user://settings.cfg"

var screen_shake: bool = true


func _ready() -> void:
    load_settings()


func load_settings() -> void:
    var cfg: ConfigFile = ConfigFile.new()
    if cfg.load(PATH) != OK:
        return                          # no file yet: keep the defaults
    screen_shake = bool(cfg.get_value("accessibility", "screen_shake", true))


func save_settings() -> void:
    var cfg: ConfigFile = ConfigFile.new()
    cfg.set_value("accessibility", "screen_shake", screen_shake)
    cfg.save(PATH)
```

2. Register it: **Project > Project Settings > Globals** tab > **Autoload**. Click the folder icon, choose `res://settings.gd`, type the name `GameSettings`, and click **Add**. (Chapter 5 introduced autoloads.)

3. Use it. Wherever a mini-game is created, pass the setting before adding it to the tree. In the Ceiling's `_play_boss_fight`, add one line after the `instantiate`:

```
# Fragment: in _play_boss_fight of the Ceiling script, right after "var game ... instantiate"
game.shake_enabled = GameSettings.screen_shake
```

4. To change the value you need a button or checkbox in your options menu (an exercise below). While developing, you can flip it from the debugger or set `screen_shake = false` in the file and watch the effect.

> **Why it works:** `ConfigFile` stores text sections and keys in a `.cfg` file under `user://`, which is always writable, even in an exported game. `get_value(section, key, default)` returns the default when the key is missing, so a new setting works with old settings files.

18.6 Playtesting with real players

You know your game too well. You know the key is in the drawer, the door needs the key, and the jar needs a click. A new player knows none of this. The only way to learn how your game feels to a stranger is to watch a stranger.

The protocol

1. **Recruit** three to five people who have not seen the game. Friends and family are fine. Different ages and gaming experience are better than five people like you.
2. **Prepare** a build (an exported `.exe` is best, see 18.7), a computer, a quiet room, and a notebook or a spreadsheet. Ask permission before you record the screen or voice, and say how you will use it.
3. **Say a short, fixed introduction.** Example: "I am testing the game, not you. You cannot do anything wrong. Please say out loud what you are thinking. I will not answer questions about how to play, but I will take notes. You can stop at any time."
4. **Do not help.** This is the hardest rule. If they are stuck, wait. Wait longer. Count to thirty slowly. Only if they are visibly unhappy, give the smallest possible hint and write down that you gave it.
5. **Watch the screen, not the person.** Where does the mouse go? What do they click that does nothing? Where do they hesitate?
6. **Stop** at a fixed time (20 to 30 minutes) or when they finish.
7. **Debrief** with three open questions: What was the best part? What was confusing? What would you change?

What to record

Make a table with one row per event. Keep it simple enough to fill in while watching.

Time	Where	What happened	Quote or note	Type
02:10	Bedroom	Clicked the bed three times	"Is the bed useful?"	Confusion
04:45	Bedroom	Did not find the drawer	Hovered over the wall	Stuck
09:30	Hallway	Jar mini-game: lost twice, then won	"Too fast!"	Difficulty
12:00	Hallway	Said "oh no" after the boss	laughed	Emotion

Also record these numbers for each player:

- Time to first success on each puzzle (key, door, bone flute, bug, boss).
- Number of deaths and retries in each mini-game.
- Moments where they said "what do I do now?"
- The moment they quit or finished, and their words.

How to read the results

One player's complaint is an anecdote. Two or three players making the same mistake is a **design problem**. Look for patterns:

Pattern	Likely cause	Typical fix
Everyone gets stuck at the same place	Missing clue or hidden object	Add a hint line, make the object more visible, change the clicked response
Players try the same wrong action	Their idea is reasonable	Make that action do something (a funny line, or even the solution)
Players lose the mini-game several times in a row	Too hard	Retune constants (Chapter 17)
Players finish instantly	Too easy	Raise difficulty or lower time
Players do not notice a reward	Feedback is weak	Add a sound, a line or a visual effect
Players quit calmly	Boredom	Shorten, or add variety

Do not argue with the player. If they did not see the drawer, the drawer is not visible enough. "They should have looked more carefully" is not a fix.

Bug triage

After a session you will have a list of many problems: real bugs, design issues and wishes. **Triage** sorts them. Give each one a severity and decide the order.

Severity	Meaning	Example	Action
Blocker	The game cannot be finished, or data is lost	Door never unlocks after a load	Fix now
Major	A big obstacle, but the game can continue	Jar game impossible; boss respawn in the wrong room	Fix before release
Minor	Annoying, cosmetic, or rare	A typo; the cursor flickers once	Fix if time allows
Wish	A new idea	"Add a second boss"	Write in `NOW.md` or a backlog; not now

Then sort by two questions: how many players hit it, and how bad was it for them. A blocker hit by one of five players outranks a minor hit by all five. Fix the top few, rebuild, and test again. Repeat until a fresh player finishes the game without help.

> **Try it:** Play your own game for ten minutes, but pretend you are a stranger: do not use what you know. Write five events in the table above. You will be surprised how many you find even before a real playtest. A real playtest will find more.

18.7 Exporting to Windows

Export turns your project into a program that runs without the Godot editor. This section follows the Godot documentation on exporting projects.

Export templates

Godot copies a prebuilt "player" into your exported game, called an **export template**. You need to install the templates once for each Godot version. They must match your editor version exactly (for example 4.7.2).

1. Open **Editor > Manage Export Templates...**
2. Click **Download and Install**, or, if you already have the template file, **Install From File**. Wait until it reports success.

Create a preset and export

1. Open **Project > Export...**
2. Click **Add...** and choose **Windows Desktop**. A preset appears in the list.
3. Optional settings in the right-hand panel: - Under **Binary Format**, tick **Embed PCK** (`binary_format/embed_pck`) to put the project data inside the executable, so that you get one file. If you leave it off, the export is two files (`.exe` and `.pck`) that must stay together. - **Export With Debug** (a checkbox in the file dialog that opens when you click Export Project) includes debug information. Turn it **off** for the build you give to players, and leave it on while you are still hunting bugs.
4. If an export warning or error appears at the bottom of the dialog, read it: it usually says exactly what is missing (for example, templates).
5. Click **Export Project...** and choose an **empty folder** outside your project, for example `C:\Games\WeirdHotel-0.1\`. Name the file `WeirdHotel.exe`.
6. Open that folder and double-click the `.exe`. The game must start without Godot.

> **Note:** The default `.exe` icon is the Godot icon. Changing it, and setting a product name and version, needs an extra tool called rcedit, set under **Editor > Editor Settings** in the export section. This is optional polish; if you want it, follow the "Changing application icon for Windows" page in the Godot docs.

Popochiu and non-resource files

By default Godot exports the project's resources (scenes, scripts, images, sounds) but not every kind of file. Files such as `.cfg` or `.json` are skipped unless you list them. Popochiu keeps a registry of your game objects in `res://game/popochiu_data.cfg`, and the game cannot find its rooms without it.

You do not need to do anything for that file: Popochiu installs an export plugin (`popochiu_export_plugin.gd`) that adds `popochiu_data.cfg` to every export by itself, as long as the Popochiu plugin is enabled when you export. If you add your *own* `.json` or `.cfg` data files under `res://`, list them in the preset's **Resources** tab

under **Filters to export non-resource files/folders** (for example `*.json`). The lesson is general: always test the exported build, never assume.

Test the exported build

Do these checks on the exported game, not in the editor:

1. Start the game from the `.exe`. Play from the first screen.
2. Save with F9, close the game, start it again and load with F10.
3. Check the audio plays, and music loops.
4. Win and lose the Jar game. Win and lose the boss fight. Check respawn.
5. Find the save and settings files. In an exported game, `user://` is in the user's application data folder under your project's name; do not look for it near the `.exe`.
6. Copy the folder to a second computer (or a new Windows user account) which does not have Godot. Run it there. If it only works on your machine, something is missing.

> **Warning:** Windows may show a "Windows protected your PC" screen for programs that are not digitally signed. That is normal for small games from unknown publishers. Tell your testers it is expected, and that you made the game. Signing a program is a paid process and out of scope for this course.

Release checklist

Print this and tick it for every build you give to others.

- [ ] Project name, version number and description are set (Project Settings > Application > Config).
- [ ] Debug prints (`print`) are removed or harmless, and **Export With Debug** is off.
- [ ] Every Input Map action used by code exists.
- [ ] New game starts correctly, and a full playthrough works start to end.
- [ ] Save and load tested at three places: early, after a puzzle, after the boss.
- [ ] Death and respawn tested with a full inventory.
- [ ] Audio is balanced (no sound much louder than the rest).
- [ ] Text is checked for typos.
- [ ] No placeholder art or temporary text visible (or it is clearly intended).
- [ ] Credits file included and accurate (18.8).
- [ ] Exported build tested on a second computer.
- [ ] A copy of the exact project state is committed to git and tagged (for example `v0.1`).
- [ ] The zip file opens and runs from a fresh folder.

Zip the export folder and name the zip with the version: `WeirdHotel-0.1-windows.zip`.

18.8 Licenses, credits, and what comes next

Why licenses matter

When you use someone else's work, a license tells you what you may do. Ignoring licenses can force you to take a game down, and crediting the people whose tools you used is simply good manners. Keep a list while you work, not at the end.

Godot

Godot is released under the **MIT license**. It allows commercial and free use; the one requirement is that the license text (with its copyright notice) is included somewhere in your game. The Godot documentation page "Complying with licenses" lists acceptable ways: a credits screen, a licenses menu, a file shipped with the game, a printed manual, printing it to the output log, or a link to godotengine.org/license. Godot also bundles third-party components that are licensed separately, and many of those licenses ask you to include their copyright notices too. GDScript can read all of this from the engine, so it never goes out of date: `Engine.get_license_text()` returns Godot's own license text (a `String`), `Engine.get_license_info()` returns a `Dictionary` of the licenses used by the components, and `Engine.get_copyright_info()` returns an `Array` of `Dictionary` entries listing each component and its copyright holders. A credits screen can display these in a `RichTextLabel` or `TextEdit`. (Exercise 5.)

Popochiu

Popochiu is also released under the **MIT License** (the `LICENSE` file at the root of the repository, which begins "Copyright (c) 2022 Mateo Robayo"). So the same rule applies: include Popochiu's copyright notice and MIT license text with your game. Copy the text from that `LICENSE` file into your credits file, together with the Popochiu name and version, and link to https://github.com/carenalgas/popochiu.

> **Warning:** The GUI templates use font files that ship inside the addon (`minecraftia-regular.ttf`, `monkeyisland_1991.ttf` and `open_sans_regular.ttf` in `addons/popochiu/engine/objects/gui/fonts/`). The repository has no license file beside them, and Popochiu's MIT license does not automatically cover third-party fonts. Before you give the game to others, look up each font's license (Open Sans, for example, is published under an open font license) or replace the fonts with ones whose license you know, and credit them.

Your assets

Asset type	Common licenses	What to credit
Art	CC0, CC-BY, a paid license	Author, source link, license (CC-BY requires attribution)
Music, sound	CC0, CC-BY, royalty-free licenses	Same
Fonts	SIL Open Font License, others	Font name and license
Your own work	You choose	Your name and the year

A **CC0** asset needs no credit but crediting is kind. A **CC-BY** asset requires credit. Anything "for personal use only" cannot go in a game that you give to others.

A credits file

Make a plain text file `CREDITS.txt` and ship it in the same folder as the exe:

```
WEIRD HOTEL v0.1
Made by <your name>, 2026

Made with Godot Engine 4.7.2 (MIT License), https://godotengine.org
  <paste the text of Engine.get_license_text() here, or ship GODOT_COPYRIGHT.txt>
Adventure framework: Popochiu 2.2.1 (MIT License), https://github.com/carenalgas/popochiu
  <paste the text of Popochiu's LICENSE file here>
```

```
Fonts: <font name>, <license>, <link>

Music:  "<title>" by <author>, <license>, <link>
Sounds: "<title>" by <author>, <license>, <link>
Art:    <your name>

Playtesters: <names, with permission>
```

What next?

You can now build and finish a point-and-click adventure. Where to go depends on what you want to make.

- **Your own adventure.** Sketch it on paper first (Chapter 12), then build one room at a time with this book as your reference. Keep it small: three rooms, one puzzle chain, one mini-game.
- **More dialogue power.** The Dialogue Manager addon (a community plugin for Godot, found on the Godot Asset Library and GitHub) lets writers author conversations in a text format. Popochiu has its own dialog trees (Chapter 16), which are enough for most adventures.
- **Other 2D genres.** Chapters 6 to 9 are the base for platformers, top-down games and shooters. Read the Godot docs on `TileMapLayer`, navigation and `AnimationTree`.
- **3D.** Godot has a full 3D engine. The nodes are similar (`Node3D`, `CharacterBody3D`, `MeshInstance3D`), and your knowledge of scenes, signals and GDScript transfers directly. The Godot docs have a 3D section starting from "Introduction to 3D".
- **C#.** Godot also supports C# (needs the .NET version of the editor). It is optional; GDScript is fully supported and well suited to small and medium games.
- **Community.** The official docs (docs.godotengine.org), the Godot forums and Discord, the Popochiu Discord server linked from its GitHub page, and game jams such as those on itch.io are the best places to ask questions and find players for your game. Game jams are a good way to finish something small in a few days.

Capstone assignment: Weird Hotel, release candidate

The capstone ties together all of Part III. You submit a finished, exported build with its project and a short report.

Requirements

Game (the Weird Hotel, as built in Chapters 12 to 17):

1. Two rooms (Bedroom, Hallway) with working walking, props and hotspots.
2. The key puzzle: Drawer gives the Key; the Key unlocks the Door (`key_taken`, `door_unlocked` in `Globals`).
3. The bone flute puzzle: combine Bone and Twine into BoneFlute.
4. The Frog dialogue (`ChatWithFrog`) with at least three options and one option that turns another on or off.
5. The Jar mini-game and the Ceiling boss fight, both following the contract.
6. Death returns the player to the Bedroom with items and state intact.

Chapter 18 additions:

7. Save and load work (F9 and F10, and/or the GUI menu), tested and documented.
8. At least one music track and three sound effects through the `A` singleton.

9. At least one use of `T` or `E.camera.shake`.
10. Idle lines for at least one object, using a counter in `Globals`.
11. At least one accessibility option stored in a settings file.
12. Exported Windows build, tested on a second computer, with `CREDITS.txt`.

Report (one to two pages):

13. A playtest summary: number of players, a table of at least eight observations, and the top three problems you fixed because of them.
14. A one-paragraph post-mortem: what went well, what you would change, what you learned.

Grading rubric (100 points)

Criterion	Excellent	Good	Fair	Missing or poor
Adventure puzzles (20) Key, Door, bone flute, Frog dialogue work and are saved	All work and are robust to odd click orders (18-20)	All work in the normal order (14-17)	One is broken or can be skipped (8-13)	Two or more broken (0-7)
Mini-games (20) Jar and boss follow the contract, block the GUI, free their scenes	Both follow the contract; no stuck states; difficulty feels tuned (18-20)	Both work; minor roughness (14-17)	One works, or the GUI is not blocked (8-13)	Neither works (0-7)
Saving (10)	Tested at three points; all state correct after load; settings kept separate (9-10)	Works at most points (7-8)	Works only sometimes (4-6)	Not working (0-3)
Audio and polish (10)	Music, effects, transition or shake, idle lines, all used with taste (9-10)	Most present (7-8)	Few present (4-6)	Almost none (0-3)
Accessibility (5)	Option works and persists; color is not the only signal (5)	Option works (4)	Option exists but does not persist (2-3)	None (0-1)
Code quality (10) Typed GDScript, clear names, comments where non-obvious, no duplicated logic, single exit in mini-games	Clean throughout (9-10)	Mostly clean (7-8)	Often unclear (4-6)	Hard to read (0-3)
Export and release (10) Working exe, tested on a second PC, credits and licenses	All items done (9-10)	Works; credits incomplete (7-8)	Works only on your machine (4-6)	No working export (0-3)
Playtest report (10)	Real players, concrete table, fixes tied to findings (9-10)	Real players, general findings (7-8)	Few players or vague (4-6)	No playtest (0-3)
Professional habits (5) Git history with meaningful commits, no secrets or huge files in the repo	Clean history (5)	Mostly clean (4)	Few commits (2-3)	None (0-1)

A grade of 90 to 100 is an A, 80 to 89 a B, 70 to 79 a C, and 60 to 69 a D. Below 60 is a resubmission.

Lab 18: Save, polish and ship Weird Hotel

Time: 90 minutes (plus a playtest session). **Needs:** your Weird Hotel project at the end of Chapter 17.

1. Open **Project Settings > InputMap** and add `quick_save` (F9) and `quick_load` (F10).
2. Open `res://game/popochiu_globals.gd` and add the `_unhandled_input` function from 18.1, and the `click_counts` dictionary with `next_line`, `on_save` and `on_load` from 18.4. Keep your existing variables.
3. Run the game. Take the Key, use it on the Door (this uses the Key up), walk to the Hallway, pick up the Bone and press F9. Close the game. Run it, press F10. Check that you are in the Hallway with the Bone, and that the bone pile is gone. Open **Project > Open User Data Folder** and find `save_1.json`.
4. Collect three short sounds and one music loop (or generate them). Name them `sfx_pickup`, `sfx_win`, `sfx_lose` and `mx_hallway` with the correct extensions and put them in `res://game/audio/`. In the Audio tab click **Scan audio files** so the cues are created.
5. Add the sound calls to the Jar and Ceiling (18.3), and the music with `_on_restore_from_savegame` to the Hallway room.
6. Create `res://settings.gd` (18.5) and register it as the autoload `GameSettings`. In the Ceiling's `_play_boss_fight`, add `game.shake_enabled = GameSettings.screen_shake`.
7. Add the camera shake and the white transition to the Ceiling (18.4).
8. Replace the Door's locked line with the `Globals.next_line("Door", [...])` version. Click it five times.
9. Press F5 and play the whole game once. Then repeat the save test: save before the boss fight, lose, load, and check the Bedroom messages do not repeat.
10. Install export templates (Editor > Manage Export Templates...), add a Windows Desktop preset, turn off Export With Debug, and export to an empty folder outside the project. (Popochiu's own `popochiu_data.cfg` is added to the export automatically; no filter is needed.)
11. Write `CREDITS.txt` and put it in the export folder. Zip the folder.
12. Recruit two people, run the protocol of 18.6, fill in the observation table, triage the problems, and fix the top two.
13. Commit to git and tag `v0.1`.

Expected result: The exported `WeirdHotel.exe` starts without Godot. F9 and F10 save and load; after a load, the Key, the Door state, the music and the click counters are restored. The Jar, Ceiling, Door and Bedroom play sounds and effects. Two people have played the game, and you have a triaged list of what to fix next.

Chapter summary

- `E.save_game(slot, description)` and `E.load_game(slot)` write and read `user://save_<slot>.json`. Popochiu saves the player's room, inventories, room, character, item and dialog state, characters' positions, and simple `Globals` variables (not dictionaries) automatically. Hidden props are not remembered; re-apply them with `_apply_state()`.
- Complex data needs `on_save()` and `on_load(data)` in `Globals` (no underscore) or `_on_save()` and `_on_load(data)` in a state script. A load does not call `_on_room_transition_finished()`, so use `_on_restore_from_savegame()` to rebuild anything that is not saved.
- Test every new piece of state with a real save-and-load cycle.
- Audio files inside `res://game/` named `mx_`, `sfx_`, `vo_`, `ui_` become cues on `A` when you click Scan audio files. Music and sound effects have different `play` arguments and different uses.
- `T.play_transition` and `E.camera.shake` add polish. Counters in `Globals` make idle lines that survive saves.
- Keep settings (volume, shake) in their own autoload and `ConfigFile`, separate from the save game.

- A playtest has a fixed protocol: do not help, record events, find patterns across players, and triage by severity.
- Export needs templates that match the editor, a preset, and an empty output folder. Test the exported build on another computer. Credit Godot, Popochiu and every asset.

Review questions

1. What are the names of the two functions that save and load a Popochiu game, and where is the save file written?
2. Name three kinds of data that Popochiu saves automatically.
3. Your `Globals` has `var lamp_color: Color`. Why is it not saved automatically, and which two functions do you add to fix that?
4. True or false: after `E.load_game`, the function `_on_room_transition_finished()` runs in the loaded room.
5. Why does the quick-save function return early when `G.is_blocked` is true?
6. A file is named `sfx_pickup.wav`. Which group does the Popochiu Audio tab put it in, and how do you play its cue?
7. Why is `GameSettings` a separate autoload and file, rather than a variable in `Globals`?
8. In `next_line`, what does `mini(n, lines.size() - 1)` prevent?
9. During a playtest a player is stuck for two minutes. What should you do, and what should you write down?
10. Multiple choice. Your exported game works on your PC but not on a friend's. Which is the *most likely* first check? (a) Reinstall Godot on the friend's PC. (b) Make sure the `.pck` file (if not embedded) is in the same folder as the `.exe`. (c) Delete `popochiu_data.cfg`. (d) Turn on Export With Debug.

Exercises

1. **Practice (easy): idle lines everywhere.** Add `next_line` idle lines to two more clickable objects (for example the Bed and the Frog). Each should have at least four lines, with the third one a joke. Verify with a save and load in the middle of a sequence.
2. **Practice (easy): a second sound.** Add a short sound for the Drawer opening and one for adding the BoneFlute. Create the cues, call them from the right scripts, and check the volume balance against the others.
3. **Apply (medium): a settings button.** Add a Checkbox ("Screen shake") to the menu you built in Chapter 8 (or a new small scene). Connect its `toggled` signal so that it sets `GameSettings.screen_shake` and calls `GameSettings.save_settings()`. At start, set its `button_pressed` from the saved value. Restart the game and check the choice is remembered.
4. **Apply (medium): a custom save value.** Add `var best_boss_time: float = 0.0` to `Globals`. Make the boss fight report how long the player survived (add a second value that the caller reads, such as `game.time_survived`, set before `finished` is emitted). In the Ceiling, after a loss, store the longest survived time in `Globals.best_boss_time` (update it only when the new value is larger). Save, quit, load, and show the value in a line of dialogue such as "I lasted 14 seconds last time."
5. **Challenge (hard): a credits screen.** Build a credits scene with a `RichTextLabel` that shows your own credits text, then Godot's license text from `Engine.get_license_text()`, and a scrolling effect with a `Tween`. Open it from a button on the main menu. Test that the full text appears in an exported build.

Common mistakes

- **Symptom:** after loading, the Key is back in the drawer, or puzzles reset. **Cause:** the flag lives in a script variable, not in `Globals` or a state script. **Fix:** move it to `Globals` as a simple typed variable (Chapter 15).
- **Symptom:** after loading, the music or the "waking up" lines do not play. **Cause:** they were in `_on_room_transition_finished()`, which a load does not call. **Fix:** also call them from `_on_restore_from_savegame()`.
- **Symptom:** the game cannot find `A.sfx_pickup` or similar ("Invalid access to property"). **Cause:** no cue was created (the file is outside `res://game/`, or has no known prefix and sits in Unassigned), or the file name has a different spelling. **Fix:** move the file under `res://game/`, click Scan audio files, and use the lower-case file name in code.
- **Symptom:** the idle-line counters start over after loading a game. **Cause:** `click_counts` is a `Dictionary`, which Popochiu does not save by itself. **Fix:** add the `on_save()` and `on_load()` functions from Section 18.4 to `Globals`.
- **Symptom:** "Failed to load" or a crash when loading an old save after adding variables. **Cause:** `on_load` assumes the key exists. **Fix:** always check `data.has("name")` and use a default.
- **Symptom:** the exported game starts to a black screen or errors, but F5 works. **Cause:** files not included in the export (for example data files), a missing `.pck` next to the `.exe`, or `res://` paths written with the wrong case (Windows ignores case, but exported packs do not). **Fix:** copy the whole export folder (or tick Embed PCK), add your own `*.json` data files to the export filter, use exact file name spelling, and read the error in a debug export.
- **Symptom:** the player's options reset when they load a game. **Cause:** options are stored in `Globals`, which is part of a save. **Fix:** keep settings in their own autoload and `user://settings.cfg`.

Further reading

- Godot docs, Exporting projects: https://docs.godotengine.org/en/stable/tutorials/export/exporting_projects.html
- Godot docs, Complying with licenses: https://docs.godotengine.org/en/stable/about/complying_with_licenses.html
- Godot docs, Saving games: https://docs.godotengine.org/en/stable/tutorials/io/saving_games.html
- Godot docs, ConfigFile class: https://docs.godotengine.org/en/stable/classes/class_configfile.html
- Godot docs, Input examples: https://docs.godotengine.org/en/stable/tutorials/inputs/input_examples.html
- Popochiu docs, working with game state: https://carenalgas.github.io/popochiu/the-engine-handbook/working-with-game-state/
- Popochiu docs, the Popochiu dock: https://carenalgas.github.io/popochiu/the-editor-handbook/popochiu-dock/
- Popochiu documentation home: https://carenalgas.github.io/popochiu/
- Popochiu on GitHub: https://github.com/carenalgas/popochiu

Appendices

Appendix A: GDScript 4 Quick Reference

This appendix is a lookup table, not a lesson. Each section gives the shortest correct form of a feature, with a small example. Everything here is Godot 4 GDScript (GDScript 2.0). The full reference is at https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html and the class reference at https://docs.godotengine.org/en/stable/classes/index.html.

> **Note:** All code blocks use tabs for indentation. GDScript requires every file to use tabs or spaces consistently; the editor default is tabs.

A.1 File structure and comments

A script is one file. The file itself is a class. A typical order of contents is:

```
@tool                          # optional annotation (see A.12)
class_name Hero                # optional: registers a global class name
extends CharacterBody2D        # the base class

## Documentation comment: shown in the editor help for this class.

signal health_changed(new_health: int)

enum State { IDLE, WALK }

const MAX_HEALTH: int = 10

@export var speed: float = 200.0
var health: int = MAX_HEALTH
@onready var sprite: Sprite2D = $Sprite2D

func _ready() -> void:
    pass
```

Recommended order inside a file: annotations, `class_name`, `extends`, doc comment, signals, enums, constants, exported variables, regular variables, `@onready` variables, then functions (built-in `_init`, `_ready`, `_process`, `_physics_process`, then your own).

Syntax	Meaning
`# text`	Comment to the end of the line
`## text`	Documentation comment (shown in the help panel)
`#region name` / `#endregion`	Foldable region in the script editor
`\` at the end of a line	Continue the statement on the next line
`;`	Optional statement separator (rarely used)
`pass`	Empty statement; a placeholder for a body

Parentheses, brackets, and braces let a statement span several lines without `\`.

A.2 Variables, constants, and types

```
var a = 5                     # dynamic: type can change
var b: int = 5                # explicit static type
var c := 5.0                  # inferred static type (float)
const PI_ISH: float = 3.14
var names: Array[String] = ["Ann", "Bo"]
var lookup: Dictionary = {"a": 1}
var maybe: Node = null
```

Keyword	Notes
`var`	Mutable variable. Without a type it is a `Variant`.
`const`	Compile-time constant. The value must be known when the script loads.
`static var`	One shared value for the whole class (Godot 4.1 and later).
`:=`	Infers the type from the right side. The type cannot change afterward.
`as`	Cast: `var s := node as Sprite2D` gives `null` if the cast fails.
`is`	Type test: `if node is Sprite2D:`

Built-in types

Type	Example literal	Notes
`bool`	`true`, `false`	
`int`	`42`, `0xFF`, `0b1010`, `1_000_000`	64-bit signed
`float`	`3.14`, `1e3`, `.5`	64-bit
`String`	`"hi"`, `'hi'`	Immutable text
`StringName`	`&"jump"`	Fast, interned name
`NodePath`	`^"Player/Sprite2D"`	Path to a node
`Vector2` / `Vector2i`	`Vector2(1.0, 2.0)`	Float or integer pair
`Vector3` / `Vector3i`	`Vector3(1, 2, 3)`	
`Rect2` / `Rect2i`	`Rect2(0, 0, 100, 50)`	Position and size
`Color`	`Color(1, 0, 0)`, `Color.RED`, `Color("ff0000")`	
`Transform2D`	`Transform2D.IDENTITY`	Position, rotation, scale
`Array`	`[1, "a"]`	Ordered list
`Dictionary`	`{"k": 1}`	Key-value map
`Callable`	`my_func`, `Callable(self, "my_func")`	A function reference
`Signal`	`button.pressed`	A signal reference
`Object`, `Node`, `Resource`		Engine classes

The special value `null` means "no object". `Variant` is the catch-all type that can hold any of the above.

Typed collections

```
var ids: Array[int] = [1, 2, 3]
var points: Array[Vector2] = []
var scores: Dictionary[String, int] = {"ann": 5}   # typed dictionary, Godot 4.4 and later
```

An `Array[Node2D]` cannot be assigned to an `Array[Node]` variable. Use `target.assign(source)` to copy with conversion.

Type conversion functions

Call	Result
`int(3.9)`	`3` (truncates toward zero)
`float(3)`	`3.0`
`str(42)`	`"42"`
`String.num(3.14159, 2)`	`"3.14"`
`"42".to_int()`	`42`
`"3.5".to_float()`	`3.5`
`"abc".is_valid_int()`	`false`
`var_to_str(v)` / `str_to_var(s)`	Text round-trip for any Variant

A.3 Operators

Group	Operators
Arithmetic	`+ - * / % **` (`**` is power)
Comparison	`== != < > <= >=`
Logic	`and or not` (also `&& || !`)
Assignment	`= += -= *= /= %= **=`
Bitwise	`& | ^ ~ << >>`
Membership	`in`, `not in`
Ternary	`a if cond else b`

Integer division truncates: `7 / 2` is `3`; `7 / 2.0` is `3.5`. `%` on integers gives the remainder; for floats use `fmod(a, b)`. Operator precedence, high to low: `**`, unary `-` / `~`, `* / %`, `+ -`, shifts, `&`, `^`, `|`, comparisons and `in` / `is`, `not`, `and`, `or`, ternary, assignment.

```
var label: String = "big" if health > 5 else "small"
if "a" in ["a", "b"]:
    print("found")
```

A.4 Strings

Feature	Syntax
Concatenate	`"a" + "b"`
Repeat	`"ab".repeat(3)`
Format with `%`	`"%s has %d coins" % [name, coins]`
Format with `format`	`"{n} wins".format({"n": 3})`
Raw string	`r"C:\temp"`
Multiline	triple quotes `"""text"""`
Convert to string	`str(value)`

Format codes for `%`: `%s` any value as text, `%d` integer, `%f` float, `%.2f` float with two decimals, `%05d` zero-padded integer, `%%` a literal percent sign. A single value needs no array: `"Score: %d" % score`.

Common String methods

Method	Result
`length()`	Number of characters
`is_empty()`	`true` for `""`
`to_upper()`, `to_lower()`	Case change
`capitalize()`	`"hello_world"` becomes `"Hello World"`
`strip_edges()`	Trim whitespace at both ends
`begins_with(s)`, `ends_with(s)`	`bool`
`contains(s)`	`bool`
`find(s)`	Index of first match or `-1`
`replace(a, b)`	New string with replacements
`substr(from, len)`	Part of the string
`split(sep)`	`PackedStringArray`
`join(array)`	`", ".join(["a","b"])` gives `"a, b"`
`to_int()`, `to_float()`	Number conversion
`pad_zeros(n)`	`"7".pad_zeros(3)` gives `"007"`
`to_snake_case()`	`"MyRoom"` becomes `"my_room"`
`get_extension()`, `get_file()`, `get_base_dir()`	Path helpers

Strings are indexed with `s[0]` and can be sliced only with `substr`.

A.5 Control flow

```
if hp <= 0:
    print("dead")
elif hp < 3:
    print("hurt")
```

```
else:
    print("fine")
```

```
match command:
    "go":
        print("going")
    "stop", "halt":          # several patterns
        print("stopping")
    var other:               # binds the value to a name
        print("unknown: ", other)
```

`match` also matches arrays (`[1, 2]`), dictionaries (`{"id": 1}`), and wildcards (`_`). There is no automatic fall-through; the first matching branch runs. Write `continue` inside a branch if you want Godot to try the next matching pattern.

```
for i in 5:                          # 0, 1, 2, 3, 4
    print(i)
for i in range(2, 10, 2):            # 2, 4, 6, 8
    print(i)
for item in ["a", "b"]:
    print(item)
for key in {"x": 1}:                 # iterates keys
    print(key)
for ch in "abc":                     # iterates characters
    print(ch)

var n: int = 3
while n > 0:
    n -= 1
    if n == 1:
        continue                     # next iteration
    if n == 0:
        break                        # leave the loop
```

Keyword	Effect
`break`	Leave the nearest loop
`continue`	Jump to the next iteration
`pass`	Do nothing
`return`	Leave the function (with an optional value)
`assert(cond, "message")`	Stops a debug run when `cond` is false (ignored in release builds)

A.6 Functions

```
func add(a: int, b: int = 1) -> int:
    return a + b

func log_all(first: String, ...rest: Array) -> void:   # variadic, Godot 4.5 and later
    print(first, rest)

static func double(x: float) -> float:
    return x * 2.0
```

Feature	Notes
Default arguments	`func f(x: int = 5)`; defaults must come last

Feature	Notes
Return type	`-> int`, `-> void`; omit for dynamic
Multiple results	Return an `Array` or a `Dictionary`
`static func`	Callable on the class without an instance: `MyClass.double(2.0)`
Lambdas	`var f := func(x: int) -> int: return x * 2` then `f.call(4)`
Callables	`var c: Callable = add` then `c.call(1, 2)`
Functions are not overloaded	One name, one function per class

A lambda used as a callback captures local variables by value at the time it is created.

```
var numbers: Array[int] = [1, 2, 3, 4]
var evens: Array = numbers.filter(func(n: int) -> bool: return n % 2 == 0)
var doubled: Array = numbers.map(func(n: int) -> int: return n * 2)
var total: int = numbers.reduce(func(acc: int, n: int) -> int: return acc + n, 0)
```

A.7 Classes, inheritance, and inner classes

```
# animal.gd
class_name Animal
extends Node

var sound: String = "..."

func speak() -> void:
    print(sound)
```

```
# dog.gd
class_name Dog
extends Animal

func _init() -> void:
    sound = "Woof"

func speak() -> void:
    super()                 # call the parent version
    print("(wags tail)")
```

Feature	Syntax
Base class	`extends Node2D`, or `extends "res://other.gd"`
Global class name	`class_name Hero` (must be unique and must not clash with an engine class)
Call parent method	`super()` or `super.method_name()`
Constructor	`func _init(args) -> void:`
Create a script instance	`Hero.new()` (not for scripts that extend Node and need a scene)
Inner class	`class Stats:` followed by an indented body
Check class	`if obj is Hero:`
Property getters and setters	see below

```
var health: int = 10:
    set(value):
```

```
        health = clampi(value, 0, 10)
        health_changed.emit(health)
    get:
        return health
```

Inside a setter or getter, assigning to the variable's own name does not call the setter again.

```
class Stats:
    var attack: int = 1
    var defense: int = 1

var stats := Stats.new()
```

Instance members are accessed with `self.name` or just `name`. A member variable and a function argument with the same name hide the member; use `self.name` to reach it.

A.8 Annotations

Annotations begin with `@`. They are placed on the line before a variable, function, or class, or on the same line before `var`.

Annotation	Purpose
`@export`	Shows the variable in the Inspector
`@export_range(min, max, step)`	Slider or number box; `@export_range(0, 100, 1, "or_greater")` allows larger values
`@export_enum("A", "B")`	Drop-down of fixed choices (works on `int` and `String`)
`@export_file("*.png")`	File picker that stores a path string
`@export_dir`	Directory picker
`@export_multiline`	Multi-line text box for a `String`
`@export_color_no_alpha`	Colour picker without alpha
`@export_flags("A", "B")`	Bit-flag checkboxes (`int`)
`@export_group("Name")`	Starts a named group of exports in the Inspector
`@export_subgroup("Name")`	Starts a sub-group
`@export_category("Name")`	Starts a top-level category
`@onready`	Delays initialization until the node is ready (see A.16)
`@tool`	Runs the script inside the editor (see A.12)
`@icon("res://icon.svg")`	Icon for a `class_name` class in the Create Node dialog
`@static_unload`	Frees static variables when the script is unloaded
`@warning_ignore("unused_variable")`	Silences a named warning for the next line

```
@export var speed: float = 200.0
@export_range(0.0, 1.0, 0.05) var volume: float = 0.8
@export_enum("Easy", "Normal", "Hard") var difficulty: int = 1
@export var scene_to_spawn: PackedScene
@export var items: Array[String] = []
@export_group("Combat")
@export var damage: int = 3
```

> **Warning:** `@onready` and `@export` must not be used on the same variable. The value would be overwritten when the node becomes ready, and Godot reports this as an error by default.

Exported values are set after `_init()` and before `_ready()`.

A.9 Signals

```
signal died
signal health_changed(new_health: int, old_health: int)

func take_damage(amount: int) -> void:
    var old: int = health
    health -= amount
    health_changed.emit(health, old)
    if health <= 0:
        died.emit()
```

Operation	Syntax
Declare	`signal name(arg: Type)`
Emit	`name.emit(args)`
Connect	`button.pressed.connect(_on_button_pressed)`
Connect with extra bound args	`button.pressed.connect(_on_pressed.bind("quit"))`
Connect once	`sig.connect(fn, CONNECT_ONE_SHOT)`
Disconnect	`button.pressed.disconnect(_on_button_pressed)`
Is connected?	`button.pressed.is_connected(_on_button_pressed)`
Wait for it	`await button.pressed`
Custom signal on an object	`obj.my_signal.connect(fn)`

Connecting the same callable to the same signal twice raises an error ("Signal already connected"); check `is_connected` first or use `CONNECT_REFERENCE_COUNTED` when intended.

```
func _ready() -> void:
    $Button.pressed.connect(_on_button_pressed)
    $Timer.timeout.connect(func() -> void: print("tick"))

func _on_button_pressed() -> void:
    print("pressed")
```

A.10 Coroutines and `await`

`await` pauses the current function until a signal is emitted or another coroutine finishes. The rest of the game keeps running.

```
func intro() -> void:
    print("start")
    await get_tree().create_timer(1.5).timeout
    print("1.5 seconds later")
```

```
    await $AnimationPlayer.animation_finished
    var result = await some_other_coroutine()
```

Case	What `await` returns
Signal with 0 arguments	`null`
Signal with 1 argument	That argument
Signal with 2 or more arguments	An `Array` of them
A function that itself awaits	That function's return value

A function that contains `await` is a coroutine. Calling it without `await` starts it and continues immediately. Calling it with `await` waits for it to finish. Tween example: `await create_tween().tween_interval(2.0).finished`.

A.11 Common built-in functions

Function	Result
`print(a, b)`	Prints values joined without spaces; `print("a", 1)` prints `a1`
`prints(a, b)`	Prints separated by a space
`print_rich("[b]bold[/b]")`	Prints with BBCode in the Output panel
`push_error("msg")`, `push_warning("msg")`	Add to the Debugger error list
`printerr("msg")`	Prints to the error stream
`abs(x)`, `absi(i)`, `absf(f)`	Absolute value
`sign(x)`, `signi(i)`, `signf(f)`	-1, 0, or 1
`min(a, b)`, `max(a, b)`	Smaller / larger (any numeric type)
`mini`, `maxi`, `minf`, `maxf`	Typed versions
`clamp(v, lo, hi)`, `clampi`, `clampf`	Limit a value to a range
`lerp(a, b, t)`, `lerpf`	Linear interpolation
`move_toward(from, to, delta)`	Step toward a value without overshooting
`floor(x)`, `ceil(x)`, `round(x)`	Rounding (float)
`floori`, `ceili`, `roundi`	Rounding that returns `int`
`snappedf(x, step)`	Round to the nearest step
`sqrt(x)`, `pow(x, y)`	Square root and power
`sin`, `cos`, `tan`, `atan2(y, x)`	Trigonometry (radians)
`deg_to_rad(d)`, `rad_to_deg(r)`	Angle conversion
`randi()`, `randf()`	Random 32-bit non-negative integer / random float from 0.0 to 1.0
`randi_range(a, b)`, `randf_range(a, b)`	Random in a range (inclusive for ints)
`randomize()`	Seed from the clock (automatic since Godot 4)
`range(n)`	Used in `for` loops
`len(x)`	Length of arrays, dictionaries, strings

Function	Result
`typeof(x)`	A `TYPE_*` constant
`is_instance_valid(obj)`	`false` if the node was freed
`load("res://a.tscn")`	Load a resource at runtime
`preload("res://a.tscn")`	Load a resource at script compile time (path must be a constant string)

A.12 `@tool` scripts

A script starting with `@tool` runs inside the editor as well as in the game. Use it for editor helpers and nodes that preview something. Guard game-only code with `Engine.is_editor_hint()`.

```
@tool
extends Sprite2D

@export var spin_speed: float = 1.0

func _process(delta: float) -> void:
    if Engine.is_editor_hint():
        return              # skip in the editor
    rotation += spin_speed * delta
```

> **Warning:** A `@tool` script that crashes or loops forever can freeze the editor. Test small, and save your work first. Popochiu's generated scripts for rooms, props, hotspots, regions, walkable areas, characters, and dialogs start with `@tool` (inventory item and `_state.gd` scripts do not); do not remove it where it is present.

A.13 Vector2 reference

`Vector2` holds `x` and `y` as floats. In Godot 2D, `+x` is right and `+y` is down.

Member	Meaning
`Vector2.ZERO`, `ONE`	(0, 0), (1, 1)
`Vector2.UP`, `DOWN`, `LEFT`, `RIGHT`	(0,-1), (0,1), (-1,0), (1,0)
`v.x`, `v.y`	Components
`v.length()`	Length
`v.length_squared()`	Length squared (faster; fine for comparing distances)
`v.normalized()`	Same direction, length 1
`v.distance_to(other)`	Distance between two points
`v.direction_to(other)`	Normalized vector pointing to `other`
`v.angle()`	Angle from the +x axis, in radians
`v.angle_to_point(other)`	Angle of the line from `v` to `other`
`v.rotated(radians)`	Rotated copy
`v.dot(other)`	Dot product

Member	Meaning
`v.lerp(to, t)`	Blend toward another vector
`v.move_toward(to, delta)`	Step toward a point
`v.limit_length(max)`	Shortens the vector if too long
`v.clamp(min_v, max_v)`	Clamp each component between two vectors
`v.clampf(min, max)`	Clamp each component between two floats
`v.abs()`, `v.floor()`, `v.round()`	Per-component
`v.snapped(step)`	Round to a grid
`v.is_zero_approx()`	Nearly zero
`v.is_equal_approx(other)`	Nearly equal
`v.bounce(normal)`, `v.slide(normal)`	Collision responses

```
var dir: Vector2 = global_position.direction_to(target.global_position)
velocity = dir * speed
var arrived: bool = global_position.distance_to(target.global_position) < 4.0
```

Operations with a float scale the vector (`v * 2.0`). Two vectors add and subtract component-wise. Never compare two float vectors with `==` after arithmetic; use `is_equal_approx`.

A.14 Array methods

```
var a: Array = [3, 1, 2]
a.append(4)             # [3, 1, 2, 4]
a.sort()                # [1, 2, 3, 4]
```

Method	Effect
`append(v)` / `push_back(v)`	Add to the end
`push_front(v)`	Add to the start
`insert(index, v)`	Insert at an index
`append_array(other)`	Add all of another array
`pop_back()` / `pop_front()`	Remove and return the last / first
`pop_at(index)`	Remove and return at index
`remove_at(index)`	Remove at index
`erase(v)`	Remove the first element equal to `v`
`clear()`	Empty the array
`size()`	Number of elements
`is_empty()`	`true` when empty
`has(v)`	`true` if present
`find(v)`	Index or `-1`
`count(v)`	Number of equal elements

Method	Effect
`front()`, `back()`	First / last element
`pick_random()`	Random element
`shuffle()`	Random order, in place
`sort()`	Ascending, in place
`sort_custom(callable)`	Custom order; the callable returns `true` if `a` goes before `b`
`reverse()`	Reverse in place
`slice(begin, end)`	New array of part
`duplicate()`	Copy (`duplicate(true)` is a deep copy)
`filter(callable)`	New array of elements where the callable is `true`
`map(callable)`	New array of results
`reduce(callable, initial)`	Fold to one value
`any(callable)`, `all(callable)`	Test elements
`min()`, `max()`	Smallest / largest element

Arrays are passed by reference. `var b = a` makes both names point at one array; use `a.duplicate()` for a copy. Negative indexes count from the end: `a[-1]` is the last element. Reading an out-of-range index is an error.

A.15 Dictionary methods

```
var d: Dictionary = {"name": "Frog", "hp": 3}
d["hp"] = 2
d.name = "Toad"        # dot access works for string keys written as identifiers
```

Method	Effect
`d[key]`	Value; error if the key is missing
`d.get(key, default)`	Value or the default (no error)
`d.has(key)`	`true` if the key exists
`d.has_all(keys)`	`true` if all keys exist
`d.keys()`	Array of keys
`d.values()`	Array of values
`d.size()`	Number of pairs
`d.is_empty()`	`true` when empty
`d.erase(key)`	Remove a pair
`d.clear()`	Remove all
`d.merge(other, overwrite)`	Add pairs from another dictionary
`d.duplicate()`	Copy (`duplicate(true)` is deep)
`d.find_key(value)`	First key with that value

Method	Effect
`d.get_or_add(key, default)`	Value; adds it if missing (Godot 4.3 and later)

Iterating with `for key in d:` visits keys in insertion order. To read keys and values together:

```
for key in d:
    print(key, " = ", d[key])
```

A.16 Node lifecycle order

When a scene loads, nodes are created and notified in this order:

Order	Callback	When
1	`_init()`	Object is created (before it is in the tree)
2	`_enter_tree()`	Node is added to the tree (parents first)
3	`_ready()`	Node and all its children have entered the tree (children first, then parent)
each frame	`_process(delta)`	Every rendered frame
each physics tick	`_physics_process(delta)`	Fixed rate (60 per second by default)
as events arrive	`_input(event)`	Every input event, first in line
	`_unhandled_input(event)`	Events not used by the interface
	`_unhandled_key_input(event)`	Key events not used by the interface
last	`_exit_tree()`	Node is removed from the tree (children first)

```
func _ready() -> void:
    print("ready")

func _process(delta: float) -> void:
    position.x += 100.0 * delta

func _physics_process(delta: float) -> void:
    move_and_slide()

func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("ui_accept"):
        print("accept")
        get_viewport().set_input_as_handled()
```

`delta` is the time in seconds since the previous call. Multiply speeds by it so the movement does not depend on frame rate.

Getting nodes

Syntax	Use
`$Sprite2D`	Child named `Sprite2D` (shorthand for `get_node("Sprite2D")`)
`$"Path/To Node"`	Path with spaces or deeper levels
`%PlayerName`	Node marked "Access as Unique Name" in the scene

Syntax	Use
`get_node("../Sibling")`	Relative path
`get_node_or_null("Maybe")`	`null` instead of an error
`get_parent()`	Parent node
`get_children()`	Array of direct children
`get_tree()`	The `SceneTree`
`owner`	The root of the scene the node was saved in
`has_node("Name")`	`bool`

A.17 Scene and tree operations

Operation	Code
Instantiate a scene	`var n := preload("res://enemy.tscn").instantiate()`
Add a child	`add_child(n)`
Remove from tree	`remove_child(n)`
Free safely at frame end	`n.queue_free()`
Change scene	`get_tree().change_scene_to_file("res://menu.tscn")`
Change scene from a packed scene	`get_tree().change_scene_to_packed(scene)`
Reload scene	`get_tree().reload_current_scene()`
Quit	`get_tree().quit()`
Pause	`get_tree().paused = true`
Wait	`await get_tree().create_timer(1.0).timeout`
Groups	`add_to_group("enemies")`, `is_in_group("enemies")`, `get_tree().get_nodes_in_group("enemies")`
Call on a group	`get_tree().call_group("enemies", "flash")`
Deferred call	`call_deferred("method")`
Change process behavior	`process_mode = Node.PROCESS_MODE_ALWAYS`

`add_child` during a physics callback sometimes errors; use `add_child.call_deferred(n)` or `call_deferred("add_child", n)`.

A.18 Input

Call	Meaning
`Input.is_action_pressed("jump")`	Held down now
`Input.is_action_just_pressed("jump")`	Pressed this frame
`Input.is_action_just_released("jump")`	Released this frame
`Input.get_axis("left", "right")`	-1 to 1

Call	Meaning
`Input.get_vector("left", "right", "up", "down")`	`Vector2`, length at most 1. Order: negative x, positive x, negative y, positive y
`Input.get_action_strength("jump")`	0 to 1
`event.is_action_pressed("jump")`	Inside `_input` / `_unhandled_input`
`event is InputEventMouseButton`	Mouse click events
`get_global_mouse_position()`	Mouse position in world coordinates (a `CanvasItem` method)
`Input.set_mouse_mode(Input.MOUSE_MODE_HIDDEN)`	Hide the system cursor

Define actions in **Project > Project Settings > Input Map**.

A.19 Common node types and key signals

Node	Purpose	Key signals or members
`Node`	Base of everything; logic-only nodes	`ready`, `tree_entered`, `tree_exiting`
`Node2D`	Anything with a 2D position	`position`, `rotation`, `scale`, `global_position`
`Sprite2D`	Draw one texture	`texture`, `flip_h`, `frame` (with `hframes` / `vframes`)
`AnimatedSprite2D`	Frame animation from `SpriteFrames`	`animation_finished`, `frame_changed`, `play("name")`
`AnimationPlayer`	Keyframe any property	`animation_finished(anim_name)`, `play("name")`
`Camera2D`	The view	`zoom`, `make_current()`, `position_smoothing_enabled`
`CharacterBody2D`	Player-driven physics body	`velocity`, `move_and_slide()`, `is_on_floor()`
`RigidBody2D`	Simulated physics body	`body_entered(body)`, `apply_impulse()`
`StaticBody2D`	Immovable collider	(none)
`Area2D`	Detects overlap	`body_entered(body)`, `body_exited(body)`, `area_entered(area)`, `area_exited(area)`, `input_event`
`CollisionShape2D`	Shape for a body or area	`shape`, `disabled`
`RayCast2D`	Ray test	`is_colliding()`, `get_collider()`
`TileMapLayer`	Tile grid (Godot 4.3 and later)	`set_cell()`, `get_cell_source_id()`
`Timer`	Countdown	`timeout`, `start()`, `stop()`, `wait_time`, `one_shot`
`AudioStreamPlayer`	Plays a sound	`finished`, `play()`, `stop()`, `volume_db`
`Control`	Base of all interface nodes	`gui_input(event)`, `resized`, `mouse_entered`, `mouse_exited`

Node	Purpose	Key signals or members
`Button`	Clickable button	`pressed`, `toggled(toggled_on)`
`Label`	Text	`text`
`LineEdit`	One-line text input	`text_changed(new_text)`, `text_submitted(new_text)`
`TextureRect`	Image in the interface	`texture`
`ProgressBar`	Bar	`value`, `max_value`
`HSlider`	Slider	`value_changed(value)`
`CheckBox`	Tick box	`toggled(toggled_on)`
`OptionButton`	Drop-down	`item_selected(index)`
`CanvasLayer`	Separate drawing layer, ignores camera	`layer`
`HBoxContainer`, `VBoxContainer`, `GridContainer`, `MarginContainer`, `CenterContainer`	Lay out children	(none needed)

A.20 Tweens

```
func fade_out() -> void:
    var tween: Tween = create_tween()
    tween.tween_property($Sprite2D, "modulate:a", 0.0, 0.5)
    tween.tween_callback($Sprite2D.queue_free)

func pop() -> void:
    var tween: Tween = create_tween()
    tween.set_trans(Tween.TRANS_BACK).set_ease(Tween.EASE_OUT)
    tween.tween_property(self, "scale", Vector2(1.2, 1.2), 0.15)
    tween.tween_property(self, "scale", Vector2.ONE, 0.1)
    await tween.finished
```

Steps run one after another. Call `tween.set_parallel(true)` to run them together. Never write `Tween.new()`; always use `create_tween()`.

A.21 Files and data

```
func save_text(text: String) -> void:
    var file := FileAccess.open("user://note.txt", FileAccess.WRITE)
    if file == null:
        push_error("Cannot open: %s" % FileAccess.get_open_error())
        return
    file.store_string(text)

func load_text() -> String:
    if not FileAccess.file_exists("user://note.txt"):
        return ""
    var file := FileAccess.open("user://note.txt", FileAccess.READ)
    return file.get_as_text()
```

```
var text: String = JSON.stringify({"coins": 5}, "\t")
var parsed: Variant = JSON.parse_string(text)
```

```
if parsed is Dictionary:
    var coins: int = int(parsed["coins"])   # JSON numbers load as float
```

```
var config := ConfigFile.new()
config.set_value("audio", "volume", 0.8)
config.save("user://settings.cfg")
if config.load("user://settings.cfg") == OK:
    var volume: float = config.get_value("audio", "volume", 1.0)
```

`res://` is the read-only project folder in an exported game; write only to `user://`.

A.22 Common patterns, side by side

Task	Pattern
Move toward the mouse	`global_position = global_position.move_toward(get_global_mouse_position(), speed * delta)`
Face a point	`look_at(target_position)`
Wrap a value	`wrapi(i, 0, 5)`
Run code once at start	`func _ready() -> void:`
Do something every second	A `Timer` node with `autostart`, connect `timeout`
Spawn a copy	`var e := enemy_scene.instantiate(); add_child(e)`
Safe null test	`if is_instance_valid(target):`
Check an input once	`if Input.is_action_just_pressed("jump"):`
Random item	`items.pick_random()`
Print with a label	`print("hp = ", hp)`

Appendix B: Godot Editor Shortcuts and Tips

Shortcuts here are the Windows and Linux defaults for Godot 4. The authoritative list is the official page "Default editor shortcuts" (for example https://docs.godotengine.org/en/4.6/getting_started/editor/default_key_mapping.html; in older versions the page sits at https://docs.godotengine.org/en/4.5/tutorials/editor/default_key_mapping.html). Shortcuts can change between versions, and you can view or change every one in **Editor > Editor Settings > Shortcuts**.

How this appendix was checked: every row was compared with the 4.6 version of that page. A row with a dagger (†) is **not** on that page: it is a common editor behavior that we could not confirm there, so treat it as "probably works" and check **Editor Settings > Shortcuts** or the editor's own menus. If a key here does not work in your version, look it up in that panel; searching by the action's name is quickest.

> **Note:** On Windows, "Ctrl" is the Control key. On macOS the same shortcuts normally use Cmd instead.

B.1 Run and debug

Action	Shortcut
Run the project (main scene)	F5
Run the current scene	F6
Pause the running game	F7
Stop the running game	F8
Run a specific scene (choose one)	Ctrl+Shift+F5
Toggle a breakpoint (script editor)	F9
Step over (debugger, when stopped)	F10
Step into	F11
Continue after a breakpoint	F12

> **Try it:** Open a script, click in the left margin or press F9 on a line, and press F5. The game stops on that line, and the Debugger panel shows the variable values. Press F12 to continue.

B.2 Files and scenes

Action	Shortcut
Open scene	Ctrl+O
Quick open (any file)	Shift+Alt+O
Quick open scene	Ctrl+Shift+O
Quick open script	Ctrl+Alt+O

Action	Shortcut
Save the scene	Ctrl+S
Save the scene as	Ctrl+Shift+S
Save all scenes	Ctrl+Shift+Alt+S
Reopen the last closed scene	Ctrl+Shift+T
Close the scene	Ctrl+Shift+W
Next / previous scene tab	Ctrl+Tab / Ctrl+Shift+Tab
Filter files in the FileSystem dock	Ctrl+Alt+P
Distraction-free mode	Ctrl+Shift+F11
Toggle fullscreen	Shift+F11
Expand the bottom panel	Shift+F12
Quit to the project list	Ctrl+Shift+Q
Undo	Ctrl+Z
Redo	Ctrl+Shift+Z
Command palette (search any editor action)	Ctrl+Shift+P

B.3 Workspaces

The workspace buttons are at the top centre: **2D**, **3D**, **Script**, and (in newer versions) **Game** and **AssetLib**.

Action	Shortcut
2D workspace	Ctrl+F1
3D workspace	Ctrl+F2
Script workspace	Ctrl+F3
Search the built-in help	F1

Newer versions also have a **Game** workspace that shows the running game inside the editor. It has no shortcut listed in the default table; click its button.

B.4 Scene tree (nodes)

Action	Shortcut
Add a child node	Ctrl+A
Instantiate a scene as a child (†)	Ctrl+Shift+A
Rename the selected node (†)	F2
Batch rename	Ctrl+F2
Copy the node path	Ctrl+Shift+C
Delete without the confirmation dialog	Shift+Delete
Duplicate the selected node	Ctrl+D

Action	Shortcut
Delete the selected node	Delete
Copy / Paste / Cut nodes (†)	Ctrl+C / Ctrl+V / Ctrl+X
Move node up or down in the list	Ctrl+Up / Ctrl+Down
Search the tree	Click the filter box above the tree (no shortcut)
Attach a script	Right-click the node > Attach Script (no shortcut)

Tips for the Scene dock:

- Right-click a node and choose **Access as Unique Name** to make `%NodeName` work in scripts, no matter where the node moves in the tree.
- Right-click and choose **Save Branch as Scene** to turn part of a scene into its own reusable scene.
- Right-click a node and choose **Editable Children** to change inside an instanced scene. Use it sparingly; changes become overrides.
- The small **eye** icon hides a node in the editor only. It does not change `visible` at run time.
- Drag a script from the FileSystem dock onto a node to attach it.
- Drag a node from the Scene dock into a script editor to paste a `$NodePath` (hold Ctrl while dropping to get an `@onready var` line).

B.5 2D viewport

Action	Shortcut
Select tool	Q
Move tool	W
Rotate tool	E
Scale tool	S
Pan view	Space+drag; middle mouse button drag (†)
Zoom	Ctrl+= / Ctrl+-; mouse wheel (†)
Reset zoom to 100%	Ctrl+0
Center on the selection	F
Frame the selection	Shift+F
Ruler tool	R
Use grid snap	Shift+G
Use smart snap	Shift+S
Always show grid	G
Show helpers	H
Show guides	Y
Constrain movement while dragging (†)	hold Shift
Pick a node under others (†)	Alt+right click (a list of nodes under the cursor)

> **Note:** The tool shortcuts Q, W, E, S, and the framing keys act on the viewport that has the focus. Click the viewport once first, then press the key.

B.6 Script editor

Action	Shortcut
Find	Ctrl+F
Find next / previous	F3 / Shift+F3
Replace	Ctrl+R
Find in all files	Ctrl+Shift+F
Move the line up / down	Shift+Alt+Up / Shift+Alt+Down
Next / previous script tab	Ctrl+Shift+. / Ctrl+Shift+,
Reopen the last closed script	Ctrl+Shift+T
Save the script	Ctrl+Alt+S
Soft reload the script	Ctrl+Shift+R
Go back / forward in history	Alt+Left / Alt+Right
Close the script	Ctrl+W
Run the open tool script	Ctrl+Shift+X
Show or hide the scripts list	Ctrl+\
Zoom the text in / out / reset	Ctrl+= / Ctrl+- / Ctrl+0
Toggle a comment on the selected lines	Ctrl+K
Delete the current line	Ctrl+Shift+K
Show code completion	Ctrl+Space
Indent / unindent (†)	Tab / Shift+Tab
Jump to the definition of the word under the cursor (†)	Ctrl+Click
Add another text caret (†)	Alt+Click

The rows without a dagger come from the official default shortcut table. The three rows with a dagger are common editing behaviors that we could not confirm on that page; if one does not work, search for the action in Editor Settings.

The official page notes that it can lag behind the newest version. Open **Editor Settings > Shortcuts** to confirm any key.

Script editor tips

- Errors are underlined in red, warnings in yellow. Hover over the underline for the message.
- The **Search Help** button and the built-in class reference let you read the docs without leaving the editor.
- Keep static types on. The editor then offers better completion and finds typos before you run the game.
- The Output panel prints `print()` results. The Debugger panel shows errors and the stack.

- Use `#region` and `#endregion` comments to fold long scripts into sections.

B.7 Inspector tips

- The search box at the top of the Inspector filters properties by name.
- Click the small **reset arrow** next to a property to put back its default value.
- Right-click a property name and choose **Copy Property** and **Paste Property** to move values between nodes.
- Drag a file from the FileSystem dock onto a texture or resource slot to assign it.
- Click a resource (like a shape) to edit its properties inline. Use the **tool menu** (the wrench, top right of the Inspector) to save it as a file when you want to reuse it.
- Use the **history** arrows at the top to go back to the previously inspected object.
- If you change a property on a resource and it changes in several places, the resource is shared. Choose **Make Unique** from the resource drop-down to give this node its own copy.
- Exported variables from your script appear at the bottom of the Inspector under the script's name.

B.8 Popochiu dock

The Popochiu dock has four tabs: **Main**, **Room**, **Audio**, and **Tools**. The dock is mouse-driven, and Popochiu does not add editor keyboard shortcuts of its own. (One small exception: while editing a polygon in the 2D viewport, hovering a point and pressing Delete or Backspace removes that point.) Every row has a three-dots **Options** button; right-clicking the row opens the same menu. The actions you use most:

Action	How
Create a room, character, inventory item, or dialog tree	**Main** tab, click the + next to the group (its tooltip says, for example, "Create room")
Make a room the starting room	Room row's **Options** menu (three dots, or right-click) > **Set as Main scene**. The starting-room icon then shows on that row
Make a character the player	Character row's **Options** menu > **Set as Player-controlled Character (PC)**. The player icon then shows on that row
Make an inventory item start in the inventory	Item row's **Options** menu > **Start with it**
Open an object's scene, script, or state	The icons on its row (tooltips "Open in Editor", "Open in Script", "Open state", "Open state Script")
Play a room on its own	The play button on the room's row ("Play scene")
Add a prop, hotspot, region, marker, or walkable area	Open a room scene, switch to the **Room** tab, click + in the group
Add a character to the room	**Room** tab, **Characters in room** group, **Add character to room** button
Create audio cues	Put files with the prefixes `mx_`, `sfx_`, `vo_`, or `ui_` in the project, open the **Audio** tab, click **Scan audio files**
Register a dimmed object	Its **Options** menu > **Add to Popochiu**
Remove an object safely	Its **Options** menu > **Remove**, not the Windows file manager

Useful habits:

- Delete Popochiu objects from the dock only. The dock updates `popochiu_data.cfg` and the generated `R`, `C`, `I`, `D`, and `A` scripts for you. The dock has no rename option, so choose names carefully when you create objects; to change a name, it is safest to create a new object with the right name and move your code over.
- The **Room** tab only works while a room scene is the open scene in the editor. Until then it shows "Open a room in Main tab to see or create Props, Hotspots, Regions and Points."
- Rows shown dimmed in the dock are not yet registered in `res://game/popochiu_data.cfg`.
- The **Tools** tab holds translation helpers (**Sync translation sources**, **Generate POT file...**).
- After many changes, if names such as `R.Bedroom` are underlined as unknown, wait a moment for the autoload scripts to regenerate, or use **Project > Reload Current Project** (check your version's docs).

B.9 Handy settings

Open **Editor > Editor Settings**. Use the search box at the top.

Setting	Where and why
Interface theme and scale	**Interface > Editor > Display Scale** (set to Custom to make everything bigger; restart if asked)
Script text size	**Interface > Editor > Code Font Size** (the name can differ between versions; search for "font size")
Show line numbers	**Text Editor > Appearance > Line Numbers**
Indent type	**Text Editor > Behavior > Indent > Type** (leave as Tabs for GDScript)
Auto-save before running	**Run > Auto Save > Save Before Running**, so unsaved scripts do not run old code
Code completion	**Text Editor > Completion**
External script editor	**Text Editor > External > Use External Editor** with an **Exec Path** and **Exec Flags**
Shortcuts	**Shortcuts** tab, to see or change any key

> **Note:** Setting names move around between Godot versions. Type a keyword (for example "font size") into the search box instead of browsing.

Using an external editor

Godot can open scripts in Visual Studio Code or another editor. Turn on **Use External Editor**, then set **Exec Path** to the editor program and **Exec Flags** to the arguments the editor needs. For Visual Studio Code the flags are commonly `{project} --goto {file}:{line}:{col}`. Godot's own script editor still gives the best completion for beginners, so try the built-in one first.

Project settings worth knowing

Open **Project > Project Settings**.

- **Application > Run > Main Scene**: which scene F5 starts. (In a Popochiu project the start room is set from the dock with **Set as Main scene**, which writes this setting for you.)
- **Display > Window**: game window size, stretch mode, and fullscreen options.

- **InputMap** tab: defines your actions, such as `move_left`.
- **Layer Names > 2D Physics**: give names to collision layers so the checkboxes make sense.
- **Plugins** tab: enable or disable add-ons such as Popochiu.
- Use **Project > Open User Data Folder** to find your save files under `user://`.
- Use **Project > Reload Current Project** after enabling a plugin or if the editor looks stale.

B.10 Safety habits

- Save with Ctrl+S often. Editor crashes are rare but possible.
- Commit to git before large changes (Chapter 11 and Appendix F).
- Do not rename or move files with Windows Explorer; use the FileSystem dock so Godot updates references. In Popochiu projects use the dock for `res://game/` content.
- If the editor warns that files changed on disk, choose **Reload** unless you edited the same file in another program and want to keep that version.

Appendix C: Troubleshooting and Error-Message Dictionary

When something breaks, do not panic and do not guess. Follow this routine:

1. **Read the message.** Open the **Debugger** panel (bottom of the editor) and the **Output** panel. The first error is usually the real one; later errors are often fallout.
2. **Note the file and line number.** Godot prints them, for example `res://player.gd:12`. Double-click the line in the Debugger to jump to it.
3. **Look it up here.** Find the nearest row in the tables below.
4. **Change one thing at a time**, then run again.
5. If you are still stuck after 15 minutes, ask for a hint (Appendix F) or search the official docs and the Godot and Popochiu issue trackers.

> **Note:** Exact message wording changes slightly between Godot versions. Match on the key words, not the whole sentence. Messages shown in `code font` are typical examples, not exact copies; they were checked against Godot 4.x behavior, but your version may word them differently.

C.1 GDScript parser and syntax errors

Message or symptom	Meaning	Fix
`Parse Error: Expected indented block after "if" condition` (or `"func"`, `"for"`, `"while"`)	A line that ends in a colon needs an indented body, and there is none	Indent the next line one level (one Tab), or write `pass` as a placeholder
`Parse Error: Unindent doesn't match the previous indentation level`	Lines in one block use different indentation	Select the lines, make each level exactly one Tab
`Used tab character for indentation instead of space as used before in the file` (or the reverse)	You mixed tabs and spaces in one file	Pick tabs. Use **Edit > Convert Indent to Tabs** in the script editor (**Edit** menu), then fix any line left over
`Parse Error: Expected end of statement after expression, found "Identifier" instead`	Two things were written side by side, such as a missing operator or a stray word	Look at the end of the line and the line above; look for a missing `=`, `,`, or `:`
`Parse Error: Expected ":" after "if" condition`	A missing colon at the end of `if`, `elif`, `else`, `for`, `while`, `func`, or `match`	Add the colon
`Parse Error: Unexpected "Identifier" in class body`	A statement sits outside any function, such as `print("hi")` at the top level	Move the statement into `_ready()` or another function
`Parse Error: Expected closing ")"` (or `"]"`, `"}"`)	Brackets are not balanced	Count opening and closing brackets on the line and the lines before
`Parse Error: Unexpected "else" ...`	`else` is not aligned with its `if`, or follows a `match` branch	Align `else` with the matching `if`

Message or symptom	Meaning	Fix
An error on a line that uses `yield` (for example `Identifier "yield" not declared in the current scope`)	You used Godot 3 syntax	Use `await` instead of `yield`
A parse error on a line that starts with `onready var` or `export var` (for example `Unexpected "Identifier" in class body`)	Godot 3 syntax	Write `@onready var x = $Node` and `@export var x: int = 0`
`Could not find base class "KinematicBody2D"` or `Could not find type "Sprite" in the current scope`	Godot 3 class names	Use `Sprite2D` and `CharacterBody2D`
`Too many arguments for "move_and_slide()" call. Expected at most 0`	In Godot 4 `move_and_slide()` takes no arguments	Set `velocity = ...` first, then call `move_and_slide()`
`Cannot assign a new value to a constant.`	You tried to change a `const`	Change `const` to `var`, or stop assigning it
`Parse Error: Function "x" has the same name as a previously declared function`	Two functions have one name in the same script	Rename one. GDScript does not allow overloading
`There is already a variable named "x" declared in this scope.`	Two `var x` in one function	Remove the second `var` or rename it
An error about using `self` or an instance member in a static function	Static functions cannot use instance members	Remove `static`, or pass the data in as an argument
`Class "X" hides a global script class.` (or `Class "X" hides a native class.` for an engine class)	Your `class_name` matches another script's class name or an existing Godot class	Rename your class (for example `PlayerHero`, not `Node`, `Timer`, `Tween`, or `Input`)
`Parse Error: Could not find base class "X"`	`extends X` names a class that does not exist (typo, or a `class_name` script that has an error)	Fix the spelling; fix errors in the script that defines `X` first

C.2 GDScript analysis errors (found at load time)

Message or symptom	Meaning	Fix
`Identifier "x" not declared in the current scope`	You used a name that does not exist here: a typo, a variable declared in another function, or a missing `var`	Check the spelling (names are case-sensitive). Declare the variable before use, at the right level
`Invalid call. Nonexistent function 'foo' in base 'Node2D (script)'`	The function does not exist on that object	Check the spelling and the class docs (**F1** search help). It may be a method from a different class, or a Godot 3 name
`Invalid call. Nonexistent function 'foo' in base 'Nil'`	The object is `null` (it was never assigned)	Find why the variable is `null`; see the "Attempt to call function ... on a null instance" row below
`Invalid get index 'foo' (on base: 'Node2D')`	You read a property that does not exist on that object	Check the spelling; check you hold the right type of node

Message or symptom	Meaning	Fix
`Invalid set index 'foo' (on base: ...) with value of type ...`	You assigned to a property that does not exist or has another type	Check the property name and type
`Invalid get index 'x' (on base: 'Dictionary')`	The key is not in the dictionary	Use `d.has("x")` or `d.get("x", default)`
`Invalid get index '5' (on base: 'Array')` / `Out of bounds get index '5' (on base: 'Array')`	The index is larger than the last element	Check `size()` first; remember the last index is `size() - 1`
`Attempt to call function 'x' in base 'null instance' on a null instance`	The variable holds `null`: a node that was not found, freed, or never set	`print()` the variable. Check the node path, check `@onready`, check the node was not freed. Guard with `if node != null:`
`Trying to assign value of type 'X' to a variable of type 'Y'`	A static type does not match	Convert (`int()`, `float()`, `str()`) or change the declared type
`Cannot assign a value of type String to variable "x" with specified type int.`	Same idea, caught before the game runs, with a conversion missing	`int("5")` or `"5".to_int()`
`The function signature doesn't match the parent`	An overriding function has different arguments or return type	Copy the signature from the class docs exactly
`The variable type is being inferred from a Variant value, so it will be typed as Variant`	`:=` used with a value of unknown type	Write the type explicitly: `var x: int = ...`
`The local variable "x" is declared but never used in the block.` (warning)	A harmless note	Remove the variable, rename it with a leading underscore (`_x`), or ignore
`The parameter "delta" is never used in the function "_process()".` (warning)	Same, for a function argument	Rename it `_delta`
`Division by zero error in operator '/'`	An integer divided by 0	Check the divisor; use a float or guard against zero
`Integer division, decimal part will be discarded` (warning)	`5 / 2` is `2`	Use `5 / 2.0` when you want a fraction
`Cannot infer the type of "x" variable because the value doesn't have a set type`	`:=` on a dynamic value	Declare the type
`Signal "x" is already connected to given callable` / `Signal already connected`	The same connection was made twice	Connect once (for example in `_ready()`); or check `is_connected` first; remove duplicate connections in the Node dock's Signals tab
`Error calling from signal 'pressed' to callable: ... Method not found.` (a connection made in the editor)	The connected function was renamed or deleted	Re-create the function with the exact name, or reconnect the signal
`Attempt to connect nonexistent signal 'x' to callable`	Misspelled or wrong signal name	Look up the signal in the class docs

Message or symptom	Meaning	Fix
A parse error on an `await` that is outside any function	`await` at the top of a script	Put it inside a function

C.3 Node and scene errors

Message or symptom	Meaning	Fix
`Node not found: "Player/Sprite2D" (relative to "/root/Main")`	`get_node`, `$Name`, or `%Name` could not find that child	Compare with the Scene dock exactly (case, spaces). For `%Name`, enable **Access as Unique Name**
`Node not found` only after changing scenes	The path is right in the scene you wrote, but the script now runs in another scene	Use `get_node_or_null()` and check, or restructure so the script uses exports
`@onready` variable is `null` in `_init()`	`_init()` runs before the node is in the tree	Use `_ready()`
`Can't change this state while flushing queries. Use call_deferred() or set_deferred()`	You changed physics state (adding bodies, toggling `disabled`) inside a physics callback	Use `set_deferred("disabled", true)` or `call_deferred("add_child", node)`
`Parent node is busy setting up children, add_child() failed. Consider using add_child.call_deferred(child) instead.`	`add_child` called inside `_ready()` of the parent in some cases	Use `add_child.call_deferred(child)`
`Can't add child 'X' to 'Y', already has a parent 'Z'.`	Adding a node that is already in the tree	Call `remove_child` first or `reparent`; or instance a new copy
`Attempt to call function on a previously freed instance`	The node was freed with `queue_free()` and you still use the old variable	Check `is_instance_valid(node)` before use
`Cannot open file 'res://x.tscn'.` / `Failed loading resource: res://x.tscn`	Wrong path, or the file moved	Check the path in the FileSystem dock. Drag the file into the code to paste the correct path
Path works in the editor but not in the exported game	File names are case-sensitive in the exported game even though Windows ignores case in the editor	Use the exact same capitalization everywhere (`Player.png` is not `player.png`)
`Scene 'res://x.tscn' has missing dependencies` / the "Error loading scene... dependencies" dialog	A resource the scene uses (script, texture, sub-scene) was moved or deleted outside Godot	In the dialog choose **Fix Dependencies** and select the new file, or restore the file from git
An error about a cyclic reference or dependency when a scene preloads itself, directly or through another	Scene A contains scene B which contains scene A	Break the loop: build the child at runtime with `load()` inside a function instead of `preload()`, or redesign

Message or symptom	Meaning	Fix
An error about a cyclic dependency between two scripts that `preload` each other	Same issue between scripts	Use `load()` in one of them at run time, or use `class_name` references
`Scene instancing: missing script` / script icon with a red mark	The script file was moved, deleted, or has an error	Reattach the script, or fix the error
`Invalid owner. Owner must be an ancestor in the tree.`	A node added at runtime was set as the scene owner wrongly	Avoid setting `owner` unless you build editor tools
Changing a property in the Inspector does nothing at run time	A script sets the same property in `_ready()` or `_process()`, overriding it	Search the scripts for that property name
Nodes drawn in the wrong order	Draw order follows the scene tree (lower is on top) or `z_index`	Reorder in the tree or set **Z Index**
Child node moves strangely when parent scales	Scale is inherited	Avoid scaling parents of bodies and shapes; resize the texture or shape instead

C.4 Physics oddities

Message or symptom	Meaning	Fix
Player falls through the floor	The floor has no `CollisionShape2D` with a shape, the shape is not enabled, or layers and masks do not match	Add a shape and assign a resource. Check **Collision > Layer** on the floor and **Mask** on the player: the mask must include the floor's layer
`Node ... CollisionShape2D ... A shape must be provided for CollisionShape2D to function` (editor warning triangle)	The node has an empty **Shape**	Select the node and set Shape to a new `RectangleShape2D` or another shape
Bodies do not collide at all	Wrong layers or masks, or both are areas (areas only detect, they do not block)	Open the Inspector, **Collision**, compare layers and masks; use a body for solid things
`body_entered` never fires	The area's `monitoring` is off, mask does not include the body's layer, or you connected to the wrong node	Check **Monitoring**, **Collision > Mask**, and that the signal is connected to a function that exists
`body_entered` fires twice or for the wrong node	Multiple colliders on the same body, or the body is also another group member	Filter with `if body.is_in_group("player"):`
Player slides on a slope or will not stay still	Floor settings or small float values	Zero `velocity` when no input; check `floor_stop_on_slope` in the CharacterBody2D settings

Message or symptom	Meaning	Fix
`is_on_floor()` is always false	The body is not touching a floor, or `up_direction` is wrong, or you call it before `move_and_slide()`	Call `move_and_slide()` first, then check; check **Up Direction** is `(0, -1)` for side-view games
Player jitters near walls	Moving in `_process` instead of `_physics_process`	Move bodies in `_physics_process(delta)`
Player moves faster diagonally	The direction vector was not normalized	Use `Input.get_vector(...)` or `.normalized()`
Player walks through thin walls at high speed	Tunnelling: movement per tick is bigger than the wall	Make walls thicker, slow the body, or lower speed
RigidBody2D ignores `position = ...`	Physics overwrites the transform	Use `apply_impulse()` / `apply_force()`, or change `freeze` and move it, or use a `CharacterBody2D`
Click on an Area2D does nothing	`input_pickable` is off, or another Control covers it	Enable **Input > Pickable**, and make covering Controls `Mouse Filter: Ignore`

C.5 User interface oddities

Message or symptom	Meaning	Fix
Control node ignores its position in a container	Containers control their children's layout	Use **Layout > Container Sizing** or put a `Control` wrapper inside the container
Control is invisible or has zero size	No text, no minimum size, or anchors collapsed it	Set **Custom Minimum Size**, or set text, or use **Layout > Anchors Preset**
Button does not respond to clicks	Another Control sits on top and eats the mouse	Set the covering node's **Mouse > Filter** to Ignore (or Pass)
Interface stretches or blurs when the window resizes	Stretch settings	**Project Settings > Display > Window > Stretch Mode** (`canvas_items`) and **Aspect** (`keep`)
Text is cut off	The Label is too small or has no autowrap	Enable **Autowrap Mode**, or enlarge the node
HUD moves with the camera	The HUD is a child of a world node	Put the HUD in a `CanvasLayer`
`Control` nodes cannot be placed freely	Parent is a container	Reparent to a plain `Control`
Theme change does not show	A theme override on the node wins over the theme	Clear the node's **Theme Overrides**

C.6 Audio oddities

Message or symptom	Meaning	Fix
No sound plays	`stream` is empty, `playing` is never set, the bus is muted, or the volume is -80 dB	Assign a stream, call `play()`, check the Audio panel and `volume_db`

Message or symptom	Meaning	Fix
Sound plays only once	One-shot players need `play()` again; a looping sound needs loop enabled in the **Import** dock	Select the audio file, **Import** tab, set **Loop**, click **Reimport**
Sound cuts off when the node is freed	`queue_free()` ended the player	Wait for `finished` before freeing, or play from a persistent node
Sound clicks or is very loud	Clipping	Lower `volume_db` and check the **Audio** bottom panel meters
Music restarts on every room change	The player lives in the room scene	Put music in an autoload, or (in Popochiu) use `A.mx_name.play()`; it manages music across rooms
`Cannot open file ... .ogg`	The file is damaged or not imported	Reimport; try WAV or OGG Vorbis

C.7 Export errors

Message or symptom	Meaning	Fix
`No export template found at the expected path` / `Export templates are missing`	The templates for your Godot version are not installed	**Editor > Manage Export Templates...** then **Download and Install** (needs internet)
A message that the export templates do not match the editor version	The installed templates are for another version	Install templates that match the exact editor version
The Export window is empty, or an export from the command line says the preset was not found	No preset exists (or the name is wrong)	**Project > Export...**, **Add...**, pick **Windows Desktop**
A file loaded from code is missing after export	Non-resource files (`.json`, `.txt`, `.csv`) are only included if a filter lists them; check also the export mode	In the Export dialog, **Resources** tab, keep the mode on **Export all resources in the project** and add the file types to **Filters to export non-resource files/folders** (for example `*.json`)
Exported game starts then closes at once	A script error at startup, often a case-sensitive path	Run the exe from a terminal to see the log; check capitalization
The exported game opens a black or gray window	The main scene is not set	**Project Settings > Application > Run > Main Scene**
Windows says the file is unrecognized or blocks it	An unsigned exe	Choose **More info > Run anyway**; this is normal for hobby builds
The save file works in the editor but not in the build	Writing to `res://` (read-only after export)	Write to `user://`

C.8 Git

Message or symptom	Meaning	Fix
`fatal: not a git repository`	You are not inside a repository	`git init` in the project folder, or `cd` into the right folder
`nothing to commit, working tree clean`	No changes since the last commit	Nothing to do
`Author identity unknown`	Git does not know your name	`git config --global user.name "Your Name"` and `git config --global user.email "you@example.com"`
The `.godot/` folder is huge in the commit list	It holds generated cache files	Add `.godot/` to `.gitignore`. If it was already committed, also run `git rm -r --cached .godot` and commit
`error: failed to push some refs`	The remote has commits you do not	`git pull` first, then push again
Merge conflict markers (`<<<<<<<`) appear in a file	Both sides changed the same lines	Open the file, keep the correct lines, delete the markers, then `git add` and commit
You committed a password or key	It is now in history	Treat it as leaked: revoke it at the provider and get a new one. Never commit keys; keep them out of the project

C.9 Popochiu-specific problems

Message or symptom	Meaning	Fix
The plugin does not appear in the Plugins list	The `addons/popochiu` folder is in the wrong place	The path must be exactly `res://addons/popochiu/plugin.cfg`. Copy the `addons` folder from the release zip into the project folder (not the zip's parent folder)
Plugin will not enable, or errors on enable	A Godot version that the Popochiu version does not support	Use the pair that matches: Godot 4.7 with Popochiu 2.2.1 (4.6 with 2.1.0, 4.3 with 2.0.3). Check the README table
No Popochiu dock after enabling	Dock hidden or moved	Check **Project > Project Settings > Plugins** is ticked, then **Project > Reload Current Project**. Look at all dock tabs
Setup popup does not appear	It only shows on first install	Open it again with the **Setup** button at the bottom of the Popochiu dock
`Identifier "R" not declared in the current scope` (same for `C`, `I`, `D`, `E`, `A`, `G`, `T`, `Globals`, `Cursor`)	The autoload is missing: the plugin is off, or setup did not run	Enable the plugin; **Reload Current Project**; check **Project Settings > Globals (Autoload)** lists `Globals`, `Cursor`, `E`, `R`, `C`, `I`, `D`, `A`, `G`, `T`
`R.Bedroom` is reported as not found (for example `Cannot find member "Bedroom" in base "r.gd"`, or at run time an invalid-access error on `'Node (r.gd)'`; exact wording varies)	The room is not registered, was created outside the dock, or was renamed by hand	Create objects only from the dock. Check the room appears (not dimmed) in the dock list; for a dimmed row use its **Options** menu > **Add to Popochiu**
`R.goto_room("hallway")` does nothing or errors	Room names are the script names and are case-sensitive	Use the exact name shown in the dock, such as `"Hallway"`

Message or symptom	Meaning	Fix
`C.Frog` not found	The character does not exist or was named differently	Check the Characters list in the dock; use that name
`I.Key` not found	The inventory item does not exist, or has another name	Create it from the dock; use that name
`Nonexistent function 'is_in_inventory'` / `'has'` on an item	No such methods	Use `I.is_item_in_inventory("Key")` or `I.Key.quantity_owned > 0`
`Nonexistent function '_on_look'` or you cannot react to look	Popochiu has no `_on_look`	Use `_on_right_click()`; with verb GUIs add `on_look_at()`
Clicking a prop does nothing	The prop was created without "Will this prop have interaction?" ticked, `_on_click` is not defined, the node is hidden or `clickable` is off, the interaction polygon is missing, or the GUI is blocked	Recreate with interaction, check the prop's **Clickable** setting in the Inspector, and make sure it has an interaction polygon (select the prop, then use the **Interaction Polygon** button in the Popochiu toolbar above the 2D viewport); check the function is named exactly `_on_click()`; check nothing called `G.block()` without `G.unblock()`
Clicking a hotspot nothing happens but props work	The hotspot's polygon is empty or off-screen	Select the hotspot and edit its interaction polygon (**Interaction Polygon** button in the Popochiu toolbar above the 2D viewport) so it covers the area
Character does not walk when `walk_to_clicked` runs	No walkable area, or the click is outside it, or a missing `await`	Add a walkable area covering the floor with a polygon; always write `await C.player.walk_to_clicked()`
Character walks through walls or props	The walkable area polygon covers them	Edit the walkable area polygon, or tick the prop's **Obstacle** setting and draw its polygon with the **Obstacle Polygon** button in the Popochiu toolbar
Character stuck or not shown in a new room	No character in the room, no marker, or the player not placed	Add the player with **Add character to room**, or `C.player.teleport_to_marker("Entrance")` in `_on_room_entered()`
Actions all happen at once, in the wrong order	A missing `await`	Put `await` in front of each action: `await C.player.say("Hi")`
`C.player.queue_say("Hi")` does nothing	`queue_` functions only work inside `E.queue([...])`	Use the plain `await C.player.say("Hi")` outside a queue
The dialog never ends	The options do not call `stop()`, or the `_:` branch is missing	In `_option_selected`, add a branch calling `stop()` and a catch-all `_:` that calls `stop()`. End each pass with `_show_options()`
A dialog option never appears again after the first use	It was turned off	Use `turn_on_options(["Id"])`, but note it has no effect on a `disabled` option
`Function "turn_off_options()" not found in base self.` (typical wording)	Called without a dialog in front, outside the dialog script	Inside the dialog script call it directly; elsewhere use `D.MyDialog.turn_off_options(["Id"])`

Message or symptom	Meaning	Fix
The dialog behaves oddly when it starts (options missing or out of order)	`_on_start()` does not await anything; the template notes it "must await something to function correctly"	Keep an `await` in `_on_start()`, as the template does (`await PopochiuUtils.e.get_tree().process_frame` by default) (check your version's docs)
Item not appearing in the inventory	The `add()` call was never reached, the item name is wrong, or it is added to a different character	Check the function runs (use `print`), use `I.Key.add()`, check the inventory GUI shows
Item used on a prop but nothing happens	The prop lacks `_on_item_used`, or you compared the wrong item	`func _on_item_used(item: PopochiuInventoryItem) -> void:` and `if item == I.Key:`
Game forgets state after a room change (a door stays locked)	The state is held in a plain variable on a script that is re-created	Keep puzzle state in `Globals` (declare `var key_taken := false` in `res://game/popochiu_globals.gd`) or in the room's `state` script
`Globals.key_taken` is reported as not found (for example `Cannot find member "key_taken" in base "popochiu_globals.gd"`, or at run time an invalid-access error on `'Node (popochiu_globals.gd)'`; exact wording varies)	The variable was never declared in Globals	Declare it in `res://game/popochiu_globals.gd`
Saved game does not restore something	The value is a custom type that Popochiu cannot serialize automatically	Store only statically typed bool, int, float, String or Array variables (a `Dictionary` or an untyped variable is not saved), or save the rest yourself: `on_save()` and `on_load(data)` in Globals, `_on_save()` and `_on_load(data)` in a state script
After loading a save, room code from `_on_room_transition_finished` did not run	That hook is not called on load	Put restore logic in `_on_restore_from_savegame()`
The mini-game scene opens but clicks also hit the room	The room still receives input	Call `G.block()` before and `G.unblock()` after, give the mini-game's Controls **Mouse Filter: Stop**, and put it on a CanvasLayer above 10
Everything freezes when the mini-game pauses the tree	Popochiu movement stops when the tree is paused	Do not use `get_tree().paused`, or set the mini-game root's **Process > Mode** to Always
The GUI template change erased my edits	Changing the template overwrites `res://game/gui/`	Keep a copy of your edits before changing; decide the template early
Things break after hand-renaming a file in `res://game/`	State scripts and `popochiu_data.cfg` refer to paths	Undo the rename (git). The dock has no rename option: create a new object with the right name from the dock and move your code over

Message or symptom	Meaning	Fix
Warning about the deprecated property `in_inventory` (Popochiu prints one when you assign to it)	It is deprecated since 2.2 and will be removed in 2.3	Read `quantity_owned > 0` instead; use `add()` / `remove()` to change it
Upgrading Popochiu from 2.1.x fails	Migrations need a step	Upgrade to 2.2.0 first, then to 2.2.1, and read the release notes
Sound cue `A.sfx_door` not found	The file name lacks the right prefix or the audio was not scanned	Name files `mx_`, `sfx_`, `vo_`, or `ui_` plus a name; open the dock's **Audio** tab and click **Scan audio files**. Files left in the **Unassigned** group can be sorted with their row menu (for example **Add to Sound Effects**)

C.10 General editor and run problems

Message or symptom	Meaning	Fix
Nothing happens when you press F5	No main scene is set	The editor asks for one; choose the starting scene (in Popochiu, set the start room from the room row's **Options** menu > **Set as Main scene** in the dock)
The game shows old behavior after an edit	The script was not saved	Save with Ctrl+S, or turn on **Save Before Running** in Editor Settings
A script edit has no effect on an already-running game	The game window keeps the old code	Close the game window and press F5 again
`Parse Error` shows but the code looks fine	The previous line is the real problem	Read the line above the reported one
Editor slow or frozen	A `@tool` script loops, or the project is huge	Disable `@tool`, or restart Godot; commit often
Red dot on a scene tab	The scene has unsaved changes	Save
Everything is red after pasting code	Pasted spaces mix with tabs	Convert indent to tabs
Nodes missing after a git pull	The `.godot/` cache is stale	Close the editor, delete the `.godot/` folder, reopen (it rebuilds)
Fonts or textures look blurry in a pixel-art game	Filtering	**Project Settings > Rendering > Textures > Default Texture Filter**: Nearest
Strange behavior only in the build	Uses `res://` paths for writing, or different case	See C.7

C.11 Reading a stack trace

When an error happens in a running game the Debugger shows a **stack**: a list of function calls from newest to oldest.

```
E 0:00:02:0123   player.gd:14 @ _physics_process(): Invalid get index 'x' (on base: 'Nil').
  <GDScript Source>player.gd:14 @ _physics_process()
```

How to read it:

- The top line gives the time, file, line, function, and the message.
- Click the file link to open the script on that line.
- If the stack has several rows, the first is where it failed; the rows below show who called it.
- "on base: 'Nil'" means the thing before the dot was `null`.

Use `print()` to inspect values near the line, or set a breakpoint with F9 and run (see Appendix B).

Appendix D: Answers to Review Questions

Chapter 1

1. **c.** Deciding the story is design work done by the developer; engines provide drawing, input, and audio, not creative decisions.
2. **False.** The order is read input, update the world, then draw the frame.
3. Read input (keys, mouse, controller), update the world (move objects, run scripts and rules), and draw a frame to the screen. The loop repeats many times per second, each pass producing one frame.
4. **a.** The MIT license allows commercial use with no royalties; you only need to keep the license notice with the engine's code.
5. **False.** GDScript works in the Standard build; the .NET build is only needed for C#, and this book uses Standard.
6. **c.** `export var` is Godot 3 syntax; Godot 4 uses `@export var`.
7. `res://` means the folder that contains `project.godot` (the project root). Advantages: the project works wherever it is stored on disk, and paths stay valid after export.
8. **c.** Compatibility uses OpenGL, which runs on the widest range of older hardware.
9. **False.** `.godot/` holds only rebuildable cache and imported data; scenes live in your own `.tscn` files. It should be ignored by version control.
10. `.import` files store how each asset is imported (compression, filtering, and so on); the converted data is kept in `.godot/`. Yes, they should be saved in version control so settings stay consistent on every computer.

Chapter 2

1. **c.** The FileSystem dock lists project files starting at `res://`; the Scene dock shows nodes, not files.
2. **False.** Every node has exactly one parent (except the root, which has none).
3. A node is one building block with a single job (a label, a sprite, a timer). A scene is a tree of nodes saved in a `.tscn` file that can be reused and instanced.
4. **b.** `Node2D` adds position, rotation, and scale in 2D; `Control` is for interface layout.
5. **True.** A child's position is relative to its parent, so moving the parent moves all of its children.
6. **b.** F6 runs the current scene; F5 runs the main scene, F7 pauses, F8 stops.
7. All five lamps are instances of one `lamp.tscn` file. Changing and saving that one file updates every instance, so you do not edit five copies.
8. **b.** Files use `snake_case` and node names use `PascalCase`, so `hero_walk.tscn` with root `HeroWalk` follows the convention.
9. **False.** Remote tree edits affect only the running game and disappear when it stops.
10. Composition lets you build by combining small parts and swapping them freely. For example, an enemy made of Sprite, Collision, Health, and Shooter parts can drop the Shooter to make a harmless version, without rebuilding a class tree.

Chapter 3

1. The script's class builds on `Control`, so the node gets every property and function a Control has (such as `size` and `visible`) without any extra code, and the script can add its own on top.
2. False. Mixing tabs and spaces causes an indentation error; use tabs only (or run Edit > Indentation > Convert Indent to Tabs).
3. `print("A", 1)` joins its arguments with nothing between them and shows `A1`; `prints("A", 1)` puts a space between them and shows `A 1`.
4. When something is wrong but the game can continue, such as the player having no lives left. `push_warning` shows in color and in the Debugger's Errors tab with a file and line number, so it is not lost in the log.
5. `:=` infers the type from the value (here `int`) and locks it, so the variable is statically typed; `var c = 5` leaves `c` untyped, able to hold any value later.
6. `9 / 2` is `4` (integer division drops the remainder) and `9 / 2.0` is `4.5` (one float operand gives normal division).
7. Floats cannot store most decimals exactly, so `0.1 + 0.2` is not exactly `0.3` and `==` can be false. Use `is_equal_approx(a + b, 0.3)`.
8. `Mira has 12 coins`. The array supplies one value for each placeholder, in order: `%s` takes the text and `%d` takes the integer.
9. `Mood.HAPPY` is `2` (numbering starts at 0). `Mood.keys()[2]` returns `"HAPPY"`.
10. (c) `_physics_process`. It runs at a fixed rate that suits physics and movement of bodies; `_process` runs once per frame and the frame rate varies.

Chapter 4

1. One branch runs: the first one, from top to bottom, whose condition is true. The remaining branches are skipped even if their conditions are also true.
2. False. `match` has no fall-through, so only the matching branch runs and no `break` is needed. (`continue` inside a branch is how you ask to try the next pattern.)
3. `2, 5, 8`. The sequence starts at 2, steps by 3, and stops before 10.
4. `break` leaves the whole loop immediately; `continue` skips the rest of the current iteration and moves on to the next one.
5. It gives the keys, one at a time. Use `d[key]` inside the loop to get each value.
6. A parameter is the name in the function definition (`func add(a: int)`); an argument is the actual value you pass when calling it (`add(5)`).
7. It prints `[1, 2]`. Assigning an array to another variable shares the same array rather than copying it, so appending through `b` also changes `a`. Use `duplicate()` for a separate copy.
8. `d.get("mana", 0)` is safer. Reading a missing key with `d["mana"]` causes an error, while `get` returns the default value you supply.
9. No. Godot 4 seeds the random generator automatically at startup (calling `randomize()` is harmless but unnecessary). Call `seed(some_number)` once to get the same sequence every run.
10. (a) `randi_range(1, 20)`. It includes both ends; (b) can give 0, (c) gives a float, and (d) gives 0 to 19.

Chapter 5

1. It gives the script a global name, so you can use `Lamp` as a type and write `Lamp.new()` from any script. The name must not match a built-in Godot class like `Timer` (or an autoload name), because the names would clash.
2. To keep the parent's setup: if you override `_ready()` without calling `super()`, the parent's `_ready()` code is skipped. Usually you call `super()` first, at the top of your `_ready()`.
3. An exported variable appears in the Inspector, can be set differently per node, and the value is saved in the scene file; a plain `var` can only be changed in code.
4. Member variables are set when the object is created, before it is in the scene tree and before its children exist, so `$Output` cannot be found yet. `@onready` delays the assignment until just before `_ready`.
5. It finds the node in the same scene that was marked "Access as Unique Name" and named `Footer`, wherever it sits in the tree. It survives moving or reparenting nodes, whereas `$Footer` is a fixed path that breaks when the layout changes.
6. `button.pressed.connect(_on_play_pressed)`. Leaving out the parentheses passes the function itself as a Callable; with parentheses it would call the function immediately and connect its result.
7. An Array holding both values, `[who, amount]`, because the signal has more than one parameter. (With a single parameter it would return that value directly.)
8. False. A child should not reach up through its parents; it breaks as soon as the scene layout changes and ties the child to one scene. The child should emit a signal and let a node above decide what to do ("call down, signal up").
9. `get_nodes_in_group` returns an array of all nodes in the group so you can inspect them; `call_group` calls the named method (`flash`) on every member without returning the nodes.
10. Good: keeping game state such as score or settings across scenes; an audio or scene-switching manager (an event bus also qualifies). Bad: using one as a dumping ground for variables that belong to a single scene, which makes bugs hard to trace because any script can change them.

Chapter 6

1. **Down.** In Godot 2D the origin is the top-left and y grows downward, so a larger y puts the node lower on the screen. Up is negative y.
2. **Radians.** A quarter turn is `PI / 2.0` (about 1.571). Use `rotation_degrees` or `deg_to_rad()` if you prefer degrees.
3. **(100, 150).** Rotating the local offset (50, 0) by 90 degrees clockwise on screen gives (0, 50), and adding the parent's position (100, 100) gives (100, 150).
4. **False.** `delta` is the time since the previous frame, so it changes with frame rate and load. That is exactly why movement is multiplied by it.
5. **4 pixels.** Distance = speed x time = 200 x 0.02 = 4.
6. **Nearest.** Set `Texture > Filter` to `Nearest` on one node, or project-wide in Project Settings > Rendering > Textures > Canvas Textures > Default Texture Filter.
7. `hframes = 4`, `vframes = 2`; **frame 5 is row 2, column 2.** Frames count left to right then top to bottom from 0, so frame 4 starts the second row and frame 5 is its second cell.
8. **Held vs. one frame.** `is_action_pressed` is true on every frame the action is held (use it for walking); `is_action_just_pressed` is true only on the frame the press began (use it for jumping or shooting once).
9. **The UI gets the first chance.** `_unhandled_input` receives only events that no button or other `Control` used, so clicking a menu does not also trigger gameplay.

10. `5.0` **and** `(0.6, 0.8)`**; length 1.** The length is the square root of 3 squared plus 4 squared. `get_vector` returns a normalized vector for diagonals (length 1, not about 1.41), so diagonal movement is not faster.

Chapter 7

1. **Fixed time step.** `_physics_process` runs at a steady rate (60 ticks per second by default), so physics behaves the same on every computer and bodies do not tunnel through thin walls at low frame rates.
2. **(a) -> 4, (b) -> 3, (c) -> 1, (d) -> 2.** `StaticBody2D` is the fixed wall, `CharacterBody2D` is moved by your code, `RigidBody2D` is moved by the physics engine (the crate), and `Area2D` only detects overlaps (the pickup zone).
3. **Yes.** The coin's mask (1) contains the layer the player is on (1), so the coin detects the player. The player's own mask (2) does not matter for the coin's `body_entered`.
4. **False.** `move_and_slide()` takes no arguments; it reads the built-in `velocity` property.
5. **Negative.** Up is negative y, so a jump sets `velocity.y` to a negative value (for example -520). Gravity then adds positive values each tick until the body falls.
6. **Whether the body is standing on the floor.** `move_and_slide()` updates it, so the value reflects the result of the most recent call.
7. `body_entered(body)`**.** It passes the entering body as a `Node2D`, so you can check it with `is_in_group` or a cast.
8. **The physics engine owns its position.** Setting `position` each frame fights the simulation and causes jitter. Use `apply_central_impulse`, `apply_force`, or set `linear_velocity` instead.
9. `TileMapLayer`**.** The old `TileMap` node is deprecated; each `TileMapLayer` node holds one layer of tiles.
10. **Debug > Visible Collision Shapes.** Turn it on before running the game to see every collision shape drawn on screen.

Chapter 8

1. A `Node2D` has a position, rotation and scale and is used for objects in the game world; a `Control` is a rectangle with a size, anchors, offsets, a theme and focus, used for interface elements. Controls adapt to the window size, Node2Ds do not.
2. The left edge stays 120 pixels to the left of the parent's right edge. The anchor is at the parent's right side (1.0), so it moves with the window, and the negative offset keeps a fixed distance from it.
3. `Fill + Expand` (`Control.SIZE_EXPAND_FILL`) on `size_flags_horizontal`. Expand asks for the spare space and Fill makes the child use it.
4. False. A container controls the position and size of its children, so any manual position is overwritten. Use size flags and minimum sizes to influence the layout.
5. `toggled(toggled_on: bool)`. The parameter is `true` when the box is now ticked and `false` when it is now cleared.
6. Set the `ColorRect`'s `mouse_filter` to `Ignore`. The default `Stop` makes the rectangle swallow all mouse clicks before the buttons behind it can see them.
7. Theme override on the node itself, then the node's own `theme`, then the theme of the nearest ancestor that has one, then the project's custom theme, and finally the built-in default theme.
8. Without focus no control receives the "accept" and arrow actions, so keyboard and gamepad players cannot see or use the menu. `grab_focus()` on the first button gives them a starting point.
9. The pause menu's `process_mode` is still `Inherit`/`Pausable`, so it freezes with the rest of the tree. Set it to `Always` (or `When Paused`).

10. (b) `canvas_items`. It redraws the 2D scene and interface at the larger size, so text and vector shapes stay sharp; `viewport` scales a fixed-size image, which suits pixel art.

Chapter 9

1. Any three of: a node's property (such as `position`, `modulate`, `visible`), a method call (Call Method track), an audio playback, another animation, or a Bezier curve value. Property tracks and Call Method tracks are the most common.
2. When you scrub an animation in the editor the properties really change, so the scene may be saved in the wrong pose. The RESET animation stores the default values, so a node always loads and saves in its resting pose.
3. False. A looping animation never finishes, so `animation_finished` does not fire; turn looping off if you need the signal.
4. `AnimatedSprite2D`, because a walk cycle is a list of pictures and this node plays a `SpriteFrames` resource with no extra setup. Use `AnimationPlayer` only when you also need to animate other properties or trigger events on the same timeline.
5. No. A tween plays once and is then finished and invalid. Call `create_tween()` again to get a new tween each time.
6. `set_trans` chooses the shape of the curve (linear, quad, bounce, elastic and so on). `set_ease` chooses where the curve is applied: at the start (`EASE_IN`), at the end (`EASE_OUT`) or both.
7. When the timer is part of the design and repeats, needs to be adjusted in the Inspector, or must be started and stopped, such as a spawner. `create_timer` is better for a single one-off wait.
8. `.ogg` for long, looping music because it is compressed and loops well; `.wav` for very short sound effects because it plays instantly without decoding.
9. It sets the volume of the bus to half of the original loudness. Decibels are a logarithmic scale, so `linear_to_db` converts a human-friendly 0 to 1 value (such as from a slider) into the decibel value the audio system needs.
10. If no stream is assigned, `play()` has nothing to play and may print errors. The guard lets the game run silently when sounds are missing, so it never crashes because of a missing file.

Chapter 10

1. A Node lives in the scene tree and can run code each frame, while a Resource is a data object outside the tree with no `_process` or `_ready`. A Resource can also be saved to a `.tres` file and shared by many users, whereas a Node exists at one place in the tree.
2. The change shows up in the other sprite too, because both hold a reference to the same Resource object. To avoid it, call `duplicate()` on the resource (or use Make Unique / Local to Scene) to get a private copy.
3. `class_name MyName` and `extends Resource`. Together they register the new type and make it a resource class that can be created from the New Resource dialog.
4. When the path is not known until the game runs (for example a level number), because `preload` needs a constant string path. `load` also lets you check with `ResourceLoader.exists` first.
5. `res://` is packed into a read-only file in an exported game, so writes fail there. `user://` is always writable and is meant for saves and settings.
6. It returns `null`. Your code must check for `null` (and may read `FileAccess.get_open_error()`), report the problem and stop, instead of using the missing file object.
7. It is a `float` (`12.0`), because JSON has one number type. Convert with `int(value)`.

8. False. JSON only has numbers, strings, booleans, arrays, dictionaries and `null`. Store a `Vector2` as `{"x":..,"y":..}` or an array and rebuild it on load.
9. Loading a `.tres` file can run scripts that are embedded in it, so a modified save file could run harmful code. JSON is only data and is safe to parse.
10. It lets newer versions of the game recognize old saves and upgrade them (migrate), and lets the game refuse saves from a newer version it does not understand.

Chapter 11

1. If you change several things at once you cannot tell which change fixed (or caused) the behavior. One change per test keeps cause and effect clear.
2. The top entry (newest) shows where the error happened. The entries below it show the chain of calls that led there, which is often where the real cause lies.
3. A red error means something failed and the current function usually stops; a yellow warning means something is suspicious but the game continues.
4. For example: Stack Trace shows the chain of function calls at a pause, and its Variables list shows the current values of locals, members and globals; Errors lists errors with lines; Monitors graphs FPS, memory and node counts; Profiler measures time per function. Any two with a correct description are right.
5. Assertions are removed in release exports, so any work inside them would silently not happen in the shipped game. Use them only to check conditions, never to do something.
6. `.godot/`, because it is a generated cache that Godot rebuilds, it is large, and it differs per machine. (`/android/` and your export folder are also usually ignored.)
7. False. The `.import` files record how assets are imported and must be committed (as must `.uid` files in Godot 4.4 and newer). Only the `.godot/` cache is ignored.
8. A parent may call methods and set properties of its children, but a child should not reach up to its parent; it emits a signal and lets the owner decide what to do. This keeps scenes independent and reusable.
9. A vertical slice is one small part of the game finished completely (art, sound, UI, gameplay, polish). Building it early proves the whole process works and gives a realistic estimate of the time needed for the rest.
10. Export again with Export With Debug on and start the `.exe` from a terminal to read the error messages. The usual cause is a file that was not included in the export.

Chapter 12

1. A parser is the part of an early adventure game that read typed sentences. Players found it frustrating because the game might understand "take lamp" but not "pick up the lamp", so guessing the words became part of the puzzle.
2. Sierra and Lucasfilm Games (LucasArts). Sierra's worlds were dangerous, with deaths and mistakes that could ruin the game later; LucasArts avoided accidental death and unwinnable states.
3. Left click performs the natural action on the object; right click looks at it.
4. It needs much less writing and scripting (about two responses per object plus item use), and it is faster to learn and works on touch screens.
5. The lock is the locked Door; the key is the Key found in the Drawer.
6. True. With no arrow between them, neither depends on the other, so the player may do them in either order.

7. A dead end is a state from which the game can no longer be finished. Example: using up a key that is needed for a second door.
8. A setback costs the player some time but the game can still be won; a dead end makes the game unwinnable.
9. If the player sees the lock first they already have a goal, so finding the key feels like a solution, not a confusing object to carry.
10. Nothing but time. They wake up in the Bedroom and keep all their items.

Chapter 13

1. A plugin (addon) is a folder of scenes, scripts and editor tools that adds features to Godot. It lives in `res://addons/`.
2. Popochiu 2.2.1 belongs to Godot 4.7. Check the version table in the Popochiu README on GitHub for the current pair.
3. False. You download the release zip named like `popochiu-v2.2.1.zip` under Assets; "Source code (zip)" is the whole repository, a different file.
4. In Project > Project Settings > Plugins, by ticking Enable next to Popochiu.
5. Simple Click (chosen in step 3 of the Setup wizard; after choosing Modern / Hi-Res, the wizard shows the high-resolution version automatically). Left click interacts and right click looks.
6. Main, Room, Audio and Tools.
7. The Rooms group (Main tab) lists the rooms of the whole game; Walkable areas (Room tab) lists the walkable polygons inside the one room that is open.
8. (a) `R` (`R.goto_room`), (b) `C` (`C.player.say`), (c) `I` (`I.Key.add()`), (d) `E` (`E.wait(0.5)`).
9. The dock registers the item in `popochiu_data.cfg`, creates its state script and regenerates `i.gd` so `I.Key` exists. Copying a folder skips these steps and gives errors.
10. `PopochiuProp` extends `PopochiuClickable`, which extends `Area2D`. Clicks are detected by Godot's normal input handling on an area, which Popochiu turns into calls to `_on_click()` and similar functions.

Chapter 14

1. A scene (`room_bedroom.tscn`, what is in the room), a script (`room_bedroom.gd`, hooks such as `_on_room_entered()`) and a state script (`room_bedroom_state.gd`, saved variables). A `.tres` registration resource is also created.
2. Tick the box in the create-room popup that offers to make it the main scene (it is shown only while the project has no main scene), or later use the **Options** button on its row in the Main tab and choose **Set as Main scene**. A heart icon marks the main scene, which is Godot's `application/run/main_scene` setting.
3. A prop is a visible picture object that can be clicked (Drawer, Jar); a hotspot is an invisible clickable zone over the background (Door, Ceiling).
4. False. Characters can only walk inside walkable areas; without one nothing can walk.
5. False. By default characters can only walk inside walkable areas; without one nothing can walk. (A character's **Ignore Walkable Areas** property changes this, but the player keeps the default.)
6. `C.player` always refers to whichever character is currently the player; `C.Frog` refers to that one character by name.
7. `say` takes time, and `await` makes the script wait until the line is finished. Without it the next action starts at once and lines overlap.

8. `_on_right_click()` runs on a right click; `_on_item_used(item)` runs when an inventory item is used on it (in Simple Click: select the item in the inventory bar, then click the prop).
9. They do not exist in 2.2.1. The correct handlers are `_on_right_click()`, `_on_item_used(item)` and `_on_room_entered()`.
10. `await R.goto_room("Hallway")`.

Chapter 15

1. `add_as_active()`. `I.Key.add_as_active()` adds the item and selects it, so the cursor becomes the item at once. Plain `add()` only puts it in the inventory.
2. **False.** There is no `has()` in Popochiu 2.2.1. Use `I.Key.quantity_owned > 0` or `I.is_item_in_inventory("Key")`.
3. `I.Key.quantity_owned > 0` **and** `I.is_item_in_inventory("Key")`**.** Prefer the second when the item name arrives as a string (from data, a dialog option, or a function parameter), because the first needs `I.Key` to be written in the code.
4. `_on_item_used(item)`**, on the clicked object.** The engine calls it on the prop, hotspot, character, or inventory item that was clicked, passing the active item as the argument.
5. **Its variable has no type, and it lives in a node that is rebuilt.** `key_taken` belonged to the Drawer node inside the Bedroom scene; leaving unloads the room, and returning creates a new Drawer. Popochiu copies back only statically typed simple variables (`bool`, `int`, `float`, `String`, `Array`), and Lab 14 removed the `: bool`, so the new Drawer starts with `false` again. Even typed, the flag is better in `Globals`: other rooms and objects (the Door, the Frog, cutscenes) can read it, it is in one place you can check, it is saved with the game automatically, and it cannot be lost by a missing type.
6. **A runtime error.** Globals only has the variables you declare in `popochiu_globals.gd`; assigning an undeclared one fails. Declare `var stone_found: bool = false` first.
7. **Example:** `door_unlocked` **in Globals,** `drawer_searches` **in the Bedroom's state.** Globals holds facts that several rooms or dialogs read; a state script holds facts that belong to one object and are mostly read only there.
8. **Because the player can combine in either direction.** Using the Twine on the Bone calls the Bone's handler, using the Bone on the Twine calls the Twine's handler. Both call one shared function, so the recipe exists in one place.
9. **So a second Key is not wasted.** If the door is already unlocked, the branch replies and returns before `remove()`, which keeps the Key (such as the Frog's spare) and prevents consuming an item for no result.
10. `_on_room_entered` **and** `_on_restore_from_savegame`**.** The second is needed because `_on_room_transition_finished` does not run when a saved game is loaded, so the loaded room must re-apply its flags itself.

Chapter 16

1. **Hub, one-shot option, gated option, ending.** The ending (an always-available exit such as "Bye") is what stops the player getting trapped; the hub keeps the conversation going and the others shape the menu.
2. **In** `_option_selected(opt)`**.** You `match` on `opt.id`. The `_:` branch catches any id you did not handle and calls `stop()` so the dialog ends instead of freezing.
3. **False.** `turn_off_options` takes an array of ids: `turn_off_options(["Who"])`.

4. **It makes the player speak the selected option's text.** The menu shows the text, and `await D.say_selected()` turns it into a spoken line before the NPC replies.
5. **So the menu reappears after any branch that did not stop.** One call at the end covers all branches. In a per-option handler the engine already shows the menu after you finish, so calling it too shows the menu twice.
6. **Flags are the truth, the menu is a view.** Popochiu does save option states, but the dialog Resource stays loaded for the whole run, so its menu can drift from the real facts (for example a new game started without quitting). Rebuilding from the Globals flags in `_on_start()` keeps one source of truth and always gives the right menu.
7. **Nothing happens, silently.** `queue_say` only returns a `Callable`; it works only inside `E.queue([...])` or `E.cutscene([...])`. On its own line it never runs. Use `await C.Frog.say(...)` instead.
8. **ESC (the `popochiu-skip` action).** The rule is that a cutscene never holds the only copy of a game fact: change flags and items before or after the cutscene, outside its instruction list.
9. **The player should see the intro, and it must not replay on load.** `_on_room_transition_finished` runs after the fade, and it is not called when a save is loaded, so loading does not replay the scene. `_on_room_entered` is for set-up.
10. `flute_given` **changes;** `key_taken` **and** `door_unlocked` **do not.** Those two describe the drawer and the door, not what the player carries, so the drawer still says "old socks" and the Door tolerates the extra Key.

Chapter 17

1. **Any two of: simplicity, reuse, easy testing.** A plain scene needs no knowledge of rooms or inventories, can be copied into another game, and can be run alone with F6. Popochiu is built for clicking and walking, not real-time dodging.
2. **Exactly one (once); the caller.** The mini-game guards its `_end()` with a `done` flag so it emits a single time, and the caller (the Jar or Ceiling script) frees the scene after `await`.
3. **The** `won` **value itself (a** `bool`**).** A signal with one argument makes `await` return that argument. With no arguments it returns `null`, with two or more it returns an `Array`.
4. **Layer 50 (any number from 16 to 99).** It must be above the GUI (10) and transitions (15) so it covers them, and below the cursor (100) so the player's cursor stays visible.
5. **True.** The interface was blocked before the mini-game started, so it must be unblocked whatever the result; otherwise the player cannot click anything. (Exception: do not call it by hand inside `E.queue`, which unblocks for you.)
6. **Key changes need no code changes.** The code asks for intent (`move_left`), and you or the player can rebind the keys, add arrow keys or a gamepad, in the Input Map.
7. **Removing an element shifts the later ones down.** Looping forwards while calling `remove_at` would skip the next block; counting backwards keeps every remaining index valid.
8. **(b).** A `CanvasLayer` does not follow the Popochiu camera, so `E.camera.shake` does not move it. The boss fight tweens its own layer `offset` to shake.
9. **Only the position is lost.** Inventory items belong to the player and `Globals.door_unlocked` is a saved global, so both stay. The player is moved to the Bedroom marker.
10. **Because the Hallway, and this script's node, may be unloaded once the room changes.** `goto_room` fades out and requests the scene change; code after it would run in a room that is being removed. Setting every fact before the room change is the safe order, and the Bedroom reads the flag in its own hooks.

Chapter 18

1. `E.save_game(slot, description)` **and** `E.load_game(slot)`**.** They write and read `user://save_<slot>.json`; on Windows you can find the folder with Project > Open User Data Folder (in the editor).
2. **Any three of: the player's room, inventories, room/character/item/dialog state (including dialog options), characters' positions and visibility, simple** `Globals` **variables. (Not saved: whether a prop is hidden, and** `Dictionary` **variables in Globals.)** These are restored without extra code.
3. **A** `Color` **is not one of the simple types (only** `bool`**,** `int`**,** `float`**,** `String` **and arrays are saved automatically).** Add `on_save()` returning a Dictionary (for example the color as HTML text) and `on_load(data)` to rebuild it, both in `Globals` without an underscore.
4. **False.** A load does not call `_on_room_transition_finished()`; use `_on_restore_from_savegame()` for things that must also happen after a load.
5. **Saving in the middle of a mini-game or cutscene would store an inconsistent state.** The mini-game is not part of the save, so loading it would leave the room in a half-finished situation; refusing while the interface is blocked avoids that.
6. **Sound effects (**`sfx_` **prefix);** `A.sfx_pickup.play()`**.** Pass `true` as the first argument and use `await` if you want to wait for it to finish.
7. **Anything simple in** `Globals` **is saved with the game.** Loading an old save would silently reset the player's options, so settings go in their own autoload and a `ConfigFile` at `user://settings.cfg`.
8. **It stops the index from going past the last line.** After the last line, the same final line repeats instead of causing an out-of-range error.
9. **Do not help yet; wait, and write down the time, place and what they tried.** If they are clearly upset, give the smallest hint and note that you did. The stuck point is the finding.
10. **(b).** An exported game made of an `.exe` and a separate `.pck` needs both files together; copy the whole export folder (or embed the pack). Reinstalling Godot is unnecessary, and the other options do not address a missing file.

Appendix E: Glossary

Terms are listed alphabetically. Chapter-specific names of Popochiu objects (for example `R`, `C`, `I`) are listed under their letters. Code words are shown in `code font`.

A

Action (input). A named input such as `jump` that you map to keys, buttons, or stick movements in the InputMap. Scripts ask about the action, not the key.

Adventure game. A game genre built on exploring places, collecting items, solving puzzles, and talking to characters. Also called a point-and-click adventure when mouse input drives it.

Anchor. A value on a Control node that sets which parts of its parent rectangle it stays attached to when the parent is resized.

Animation. A set of changes to node properties over time. In Godot it is stored in an `AnimationPlayer` or in `SpriteFrames`.

AnimatedSprite2D. A 2D node that plays frame-by-frame animations from a `SpriteFrames` resource.

AnimationPlayer. A node that stores animations and can change almost any property over time with keyframes.

Annotation. A word that begins with `@` and changes how GDScript treats the next line, such as `@export` and `@onready`.

Area2D. A 2D node that detects when bodies or other areas overlap it and sends signals. It does not block movement.

Array. An ordered list of values. Written `[1, 2, 3]`.

Asset. Any file a game uses that is not code: images, sounds, fonts, models.

Asset Library (AssetLib). The in-editor store of community add-ons and templates.

Atlas. One large image that holds many smaller pictures, such as sprite frames.

Audio bus. A channel that audio flows through, with volume and effects. Buses can be Master, Music, SFX, and others.

AudioStreamPlayer. A node that plays a sound file without any position in space.

Autoload. A script or scene that Godot loads at startup and keeps alive for the whole game. Also called a singleton.

Await. A GDScript keyword that pauses a function until a signal fires or another coroutine finishes.

B

Background (prop). In Popochiu, a prop without interaction that holds a room's backdrop picture.

Base class. The class that another class extends. All nodes have `Node` as an ancestor.

Bitmask. A number whose bits are on or off to represent choices. Collision layers are stored this way.

Body (physics). A node that takes part in physics collisions: `StaticBody2D`, `RigidBody2D`, `CharacterBody2D`.

Bool. A value that is `true` or `false`.

Boss fight. A hard encounter against one strong enemy, usually near the end of a section of a game.

Branch (git). A separate line of commits. The default is usually `main`.

Breakpoint. A marker on a code line that stops a running game so you can inspect values.

Bug. A mistake in a program that causes wrong behavior.

Build. The process of preparing a program to run; also the finished result.

Bus layout. The saved list of audio buses and their settings.

Button. A Control that sends a `pressed` signal when clicked.

C

C (Popochiu). The autoload used to reach characters, for example `C.player` or `C.Frog`.

CallDeferred. Running a function at the end of the current frame, not at once. Written `call_deferred("name")`.

Callable. A value that refers to a function and can be called later.

Camera2D. A node that decides which part of a 2D world is shown on screen.

CanvasLayer. A node that draws its children on a separate layer, unaffected by the camera. Used for interfaces.

Cast. Telling the language to treat a value as a certain type, using `as`.

CharacterBody2D. A body you move with code. Typical for players and enemies. It uses `velocity` and `move_and_slide()`.

Character (Popochiu). An object that can walk, talk, and own an inventory. The player is a character.

Child node. A node placed inside another node in the scene tree.

Class. A blueprint that defines data and functions. Each script is a class.

`class_name`. A keyword that gives a script a global name so other scripts can use it as a type.

Click handler. A function that runs when the player clicks something, such as `_on_click()` in Popochiu.

Clone. A copy of a git repository on your computer.

Collision. Two physics shapes touching each other.

Collision layer. The "I am on these layers" setting of a physics object.

Collision mask. The "I look for objects on these layers" setting of a physics object.

CollisionShape2D. A node that holds the shape used for collisions.

Color. A value with red, green, blue, and alpha parts, each from 0 to 1.

Comment. Text in code that the computer ignores, starting with `#`.

Commit (git). A saved snapshot of your project, with a message.

Compile. To translate code into something a computer can run. GDScript is compiled when loaded.

Console. The text area where messages are printed. In Godot it is the Output panel.

Constant. A named value that never changes. Declared with `const`.

Container. A Control that arranges its children automatically, such as `VBoxContainer`.

Control. The base class of all interface nodes.

Coroutine. A function that can pause (with `await`) and continue later.

Cursor. The pointer on screen. In Popochiu it is controlled by an autoload called `Cursor`.

Cutscene. A scripted scene where the player has no control, such as a story moment. Popochiu has `E.cutscene()`.

D

D (Popochiu). The autoload used to start and control dialogs, for example `D.ChatWithFrog.start()`.

Debugger. The tool that shows errors, variables, and the call stack while a game runs.

Decoupling. Designing parts of a game so they depend on each other as little as possible. Signals help.

Delta. The time in seconds since the previous frame, given to `_process` and `_physics_process`.

Dependency. A file that another file needs, such as a texture used by a scene.

Dialog (Popochiu). A conversation made of options the player picks. It is a Resource, not a scene.

Dialog option. One selectable line in a Popochiu dialog, with an id and text.

Dictionary. A collection of key-value pairs. Written `{"a": 1}`.

Dock. A panel in the editor, such as Scene, FileSystem, Inspector, or the Popochiu dock.

Dynamic typing. Letting a variable hold any type of value.

E

E (Popochiu). The main Popochiu engine autoload. It provides `wait`, `queue`, `cutscene`, and saving.

Editor. The Godot program where you build your game.

Enum. A named list of constant integers. `enum State { IDLE, WALK }`.

Engine. Software that supplies the common parts of games, such as drawing, sound, and physics.

Event (input). An object that describes something the player did, such as a key press or a click.

Export (annotation). Using `@export` to show a script variable in the Inspector.

Export (build). Turning your project into a program players can run without Godot.

Export preset. Saved settings for exporting to one platform, such as Windows Desktop.

Export template. A prebuilt game runner for one platform, installed through the editor and used during export.

Extends. The keyword that sets a script's base class.

F

Fade. A gradual change to or from a solid colour; used for transitions between rooms.

File system. The folders and files of your project, shown in the FileSystem dock.

Float. A number with a decimal part.

Frame. One picture drawn on the screen; also one image in an animation.

Frame rate. How many frames are drawn per second.

Function. A named block of code you can run by calling it.

G

G (Popochiu). The autoload for the graphic interface; `G.block()` and `G.unblock()` stop and allow input.

Game loop. The repeating cycle of reading input, updating the world, and drawing.

Game state. All information that describes where the player is in a game, such as items held and doors opened.

GDScript. Godot's own programming language, similar to Python.

Gizmo. The handles shown in the viewport for moving, rotating, and scaling.

Git. A tool that records versions of your files so you can go back.

Globals (Popochiu). The autoload script `res://game/popochiu_globals.gd` where you declare game-wide variables like `var key_taken := false`, then use them as `Globals.key_taken`.

Godot. A free, open-source game engine.

Group. A tag you give to nodes so you can find or call them together.

GUI. Graphical user interface. In Popochiu, the template that shows verbs, inventory, and text.

GUI template (Popochiu). A ready-made interface chosen at setup: 9-Verbs, Action Bar (the Sierra-style one, stored in the `sierra` folder), or Simple Click, each with a high-resolution variant. It is copied into `res://game/gui/`.

H

Hotspot (Popochiu). An invisible clickable area in a room, used for things that are part of the picture.

HUD. Heads-up display; information drawn over the game such as health or score.

HBoxContainer. A container that places its children in a row.

I

I (Popochiu). The autoload for inventory items, for example `I.Key.add()`.

Identifier. The name of a variable, function, or class.

Import. Godot's process of converting a file such as a PNG into an internal form.

Indentation. Spaces or tabs at the start of a line. In GDScript they define blocks.

Inheritance. Making a new class from an existing one, which keeps its features.

Input Map. The project list that connects actions to keys and buttons.

Inspector. The dock that shows and edits properties of the selected node or resource.

Instance. One actual object made from a class, or a copy of a scene placed in another scene.

Instantiate. To create an instance. For scenes: `scene.instantiate()`.

Int. A whole number.

Interaction (Popochiu). A way for the player to act on a prop, hotspot, or character.

Inventory. The items a character carries.

Inventory item (Popochiu). An object the player can pick up, carry, and use on other things.

J

JSON. A text format for data using braces and lists, readable by humans and many programs.

Jump. An upward movement. In a platformer it is velocity set against gravity.

K

Key (dictionary). The label used to look up a value in a dictionary.

Keyframe. A point in an animation that sets a value at a specific time.

L

Label. A Control that shows text.

Lambda. A small unnamed function written in place with `func`.

Layer (collision). See Collision layer.

Layer (CanvasLayer). An integer that sets which canvas draws on top of others.

Lerp. Short for linear interpolation: finding a value part of the way between two others.

Local space. Coordinates measured from a node's parent, not the world.

Loop. Code that repeats: `for` and `while`.

Lock-and-key puzzle. A puzzle where an item opens a blocked path.

M

Main scene. The scene Godot runs first when the game starts.

Marker (Popochiu). A position in a room with a name, such as `Entrance`, used for placing and walking characters.

Marker2D. A Godot node that marks a position.

Material. An object that tells Godot how to draw a surface, often with a shader.

Menu. A screen of choices such as Play, Options, and Quit.

Method. A function that belongs to a class.

Mini-game. A short, self-contained game inside a bigger game.

Modulate. A property that multiplies a node's colour; used for tints and fading.

MCP (Model Context Protocol). A standard for connecting AI tools to programs through small servers.

Mouse filter. A Control setting that decides whether it receives, passes on, or ignores mouse events.

`move_and_slide()`**.** The `CharacterBody2D` method that moves the body by its `velocity` and slides along surfaces.

N

Navigation. Finding a path through a space. In Popochiu it relies on walkable areas.

Node. The basic building block of a Godot scene. Every node has a name, a type, and a parent.

NodePath. A text path to a node, such as `Player/Sprite2D`.

Normalize. To scale a vector to length 1 while keeping its direction.

Null. The value meaning "no object".

O

Object. The base of all Godot classes. In general, a thing with data and functions.

`@onready`**.** Runs a variable's setup when the node is ready.

Origin. The point (0, 0) of a coordinate system.

Output panel. The panel that shows `print()` text and engine messages.

Override. To replace an inherited function with your own version.

P

Parent node. The node above another node in the tree.

Parser. The part of the engine that reads code and checks it is well formed.

Parallax. An effect where far layers move slower than near ones.

PackedScene. A scene file loaded as a resource, ready to be instanced.

Particles. Many small graphics that make effects like smoke and sparks.

Path (file). The text that names a file, for example `res://player.gd`.

Physics body. See Body.

Physics process. The function `_physics_process(delta)`, which runs at a fixed rate.

Pixel art. Pictures drawn with large, visible pixels.

Platformer. A game about running and jumping across platforms.

Plugin. An add-on that extends the editor. Popochiu is a plugin.

Point-and-click. A way of playing by clicking the mouse on things.

Polygon. A shape made of straight edges. Used for hotspots and walkable areas.

Popochiu. A free plugin for Godot 4 that supplies the parts of a point-and-click adventure game.

Position. Where a node is, as a `Vector2`, relative to its parent.

Preload. Loading a resource when the script loads, using `preload("path")`.

Print. Writing text to the Output panel.

Process. The function `_process(delta)`, which runs once per frame.

Project. A folder with a `project.godot` file that holds one game.

Project Manager. The first window of Godot where you create and open projects.

Project Settings. The window for settings that apply to the whole game.

Prop (Popochiu). A clickable or decorative object inside a room.

Pull request. A request to merge a branch, made on a hosting site like GitHub.

Push (git). To send your commits to a remote copy.

Q

Queue (Popochiu). `E.queue([...])`: a list of actions run in order while the interface is blocked.

`queue_free()`. Removes a node safely at the end of the frame.

Quaternion. A way to store 3D rotation. Not used in this course.

R

R (Popochiu). The autoload for rooms, for example `await R.goto_room("Hallway")`.

Rect2. A rectangle value with position and size.

RectangleShape2D. A box-shaped collision shape.

Region (Popochiu). An area of a room that reacts when a character enters or leaves.

Release build. A finished build without debugging tools, made for players.

Remote (git). A copy of your repository stored elsewhere, such as GitHub.

Repository. A project folder that git tracks.

Resource. A file or object that holds data, such as a texture, a sound, or a custom data class.

`res://`. The path prefix for files inside the project.

RichTextLabel. A Label that can show formatted text with BBCode.

RigidBody2D. A body controlled by the physics engine, which pushes and bounces.

Room (Popochiu). One location in an adventure game, made of a scene and a script.

Root node. The top node of a scene.

Rotation. How much a node is turned, in radians in code.

S

Save file. A file that stores game progress, usually in `user://`.

Scale. The size multiplier of a node.

Scene. A saved tree of nodes, stored in a `.tscn` file. Scenes can contain other scenes.

Scene tree. The running arrangement of all active nodes.

Script. A file of code attached to a node.

Seed (random). The starting number of a random sequence.

Shader. A small program that runs on the graphics card to change how things are drawn.

Signal. A message that a node sends when something happens. Other nodes can connect to it.

Singleton. A script available everywhere. In Godot, an autoload.

Snapping. Making movement jump to a grid or to other objects.

Sprite. A picture drawn in a game.

Sprite2D. A node that draws one texture.

SpriteFrames. A resource holding named animations of frames for an `AnimatedSprite2D`.

Spritesheet. One image holding all the frames of an animation.

Stack trace. A list of function calls that led to an error.

State. The current condition of something, such as "door unlocked".

State machine. A design where an object is in exactly one state at a time and switches by rules.

Static typing. Declaring types for variables so errors show before running.

StaticBody2D. A body that never moves, such as a wall.

String. Text in a program.

StringName. An efficient string for names, written `&"name"`.

Subscene. A scene used inside another scene.

T

Tab. The character used for indentation in GDScript; also an editor panel title.

Texture. An image used for drawing.

TextureRect. A Control that shows a texture.

Theme. A set of colours, fonts, and styles for Controls.

Tile. One small repeatable picture in a grid.

TileMapLayer. A node that draws a grid of tiles.

Timer. A node that counts down and emits `timeout`.

Tool script. A script with `@tool` that runs in the editor.

Transform. Position, rotation, and scale of a node together.

Transition (Popochiu). An effect, such as a fade, that plays when changing rooms. Reached through `T`.

Tween. An object that changes a property smoothly over time. Created with `create_tween()`.

Type. The kind of data a value is: `int`, `float`, `String`, and so on.

U

Unique name. A node marked so scripts can reach it with `%Name`.

UI. User interface; the buttons, text, and bars.

`user://`**.** The path prefix for a folder where games may save data.

V

Variable. A named place that holds a value.

Variant. The type that can hold any value.

Vector2. A pair of numbers for position or direction.

Velocity. Speed with direction, as a vector.

Verb (adventure game). An action word the player can choose, such as Look at or Use.

Version control. A system that records changes to files over time. Git is one.

Viewport. The area where the game is drawn; also the 2D/3D view in the editor.

Visible. A node property that says whether it is drawn.

W

Walkable area (Popochiu). A polygon in a room that marks where characters can walk.

Window (display). The OS window that shows the game.

World space. Coordinates measured from the origin of the whole game world.

X, Y, Z

X axis. Horizontal; positive is right.

Y axis. Vertical; in Godot 2D positive is down.

Y-sort. Drawing nodes in order of their Y position so lower ones are in front.

Z index. A number that sets drawing order; higher numbers are drawn on top.

Additional terms

Alpha. The transparency part of a colour.

Async. Short for asynchronous; work that takes time without stopping everything else.

Blocking (Popochiu). A state where the player's clicks are ignored, such as during a cutscene.

Cooldown. A waiting time before something can happen again.

Cue (audio, Popochiu). A sound or music entry that Popochiu makes available through `A`.

Debug build. A build with extra checks and tools for development.

Frame-rate independent. Behaviour that does not change with frame rate, done by multiplying by `delta`.

Hitbox. The area that decides where an attack can hurt something.

Hurtbox. The area where a character can be hurt.

Idle. An animation or state where a character does nothing.

Interpolation. Finding values between two known values.

Sandbox. A safe place to try ideas without hurting the real project.

Spawn. To create a new object in the game.

Stretch mode. The project setting that controls how the game scales when the window is resized.

Terminal. A text window where you type commands.

Tile set. A resource that holds the tiles that a tile map layer can use.

Wizard (Popochiu). The guided mode of the first-time setup popup.

Appendix F: Working with Claude Code as a Tutor

An AI assistant can be a patient tutor or a machine that does your homework and leaves you knowing nothing. The difference is how you use it. This appendix shows how to set Claude Code up as a tutor for this course, how to ask questions that teach you, and how to stay safe and honest while doing it.

The rule from the first page of this book applies here too: **use the assistant to explain and check, not to do the exercises for you.** You learn by typing code, breaking it, and fixing it.

> **Note:** AI tools change quickly. Menu names, commands, and even product names may differ from what is written here. Treat this appendix as a method, and check the current documentation of the tool you use. Claude Code's overview is at https://code.claude.com/docs/en/overview.

F.1 What Claude Code is

Claude Code is an AI coding assistant that you run in a terminal inside your project folder. It can read your files, explain them, suggest changes, and, if you allow it, edit files and run commands. Because it can edit, it needs rules. You give those rules once in a file called `CLAUDE.md`.

F.2 The CLAUDE.md teaching-rules template

Claude Code reads a file named `CLAUDE.md` in your project folder at the start of a session. Use it to make the assistant behave like a teacher.

1. Create a file named `CLAUDE.md` in the project folder (next to `project.godot`).
2. Paste in the template below.
3. Change the course details to match where you are.

```
# How to help me on this project

- I am a beginner learning Godot 4 and the Popochiu plugin, from the textbook "Game Development with
Godot 4".
- Act like a patient teacher. Use short, plain sentences.
- Explain why something works or fails, not just the fix.
- Every piece of code must have clear comments on what each part does.
- Before you edit a file, tell me what you plan to change and why, and wait for me to say yes.
- When I ask for a hint, give a hint only. Do not give the full answer unless I say "show me the
answer".
- Use Godot 4 GDScript only: tabs for indentation, static types, `@export`, `@onready`, `await`, and
`signal.connect(callable)`. Never use Godot 3 syntax.
- Use the Godot MCP server (if it is available) to run the game and read real errors. Do not guess an
error message.
- For Popochiu, read the addon code and its docs before you answer. Its function names change between
versions. Say which version you checked.
- If you are not sure a function exists, say so and check, instead of inventing it.
- Keep changes small. One thing at a time.
- Do not run `git restore`, `git reset`, or delete files without asking me first.
```

> **Why it works:** The assistant treats `CLAUDE.md` as standing instructions. A rule like "tell me before you edit" turns surprise changes into a short conversation. A rule like "say if you are not sure" reduces invented function names, which is the most common mistake AI tools make in a fast-moving engine.

You can add a short line at the top that says which chapter you are on, for example "Current chapter: 7, collisions". Update it as you go.

F.3 Good prompts and bad prompts

A good prompt gives context, says what you tried, and asks for understanding.

Bad prompt	Why it fails	Better prompt
"Make my player move."	You get code you do not understand	"I wrote the script below for a CharacterBody2D. The player does not move. Do not rewrite it. Tell me what is wrong and why, and give me a hint first."
"Fix this." (pasting a whole project)	No error, no goal	"Running the game gives this error: `Invalid call. Nonexistent function 'move_and_slide'...` What does it mean, and where should I look?"
"Do Lab 7 for me."	You skip the learning, and may break academic rules	"I am stuck on step 4 of Lab 7. Here is my scene tree and script. What should I check?"
"Write a full adventure game."	Too big; you learn nothing	"Explain what `await C.player.walk_to_clicked()` does, line by line, and what would happen without `await`."
"Why is it broken?"	No details	"I expected the door to open when I use the key on it. Nothing happens. Here is the door script. Which Popochiu function runs when an item is used?"

Prompts that teach

- "Explain this script line by line, in plain words."
- "Run the game with the MCP. What is the error, and what does it mean?"
- "I think this bug happens because of X. Am I right?"
- "Give me a hint, not the answer."
- "What would break if I deleted this line?"
- "What is the difference between `_process` and `_physics_process`? Give a tiny example."
- "Quiz me on this chapter with five questions. Do not show the answers until I reply."
- "Here is my solution to the exercise. Review it and tell me what could be better, but do not rewrite it."
- "Which docs page explains this? Give me the link and tell me what to look for."

Prompts to avoid

- Anything that asks it to finish a lab or exercise that you are supposed to hand in.
- Anything that includes passwords, API keys, or personal data. Never paste them into a prompt.
- "Fix everything" with no description. Small questions get better answers.

F.4 The explain-then-try loop

Use this loop for every new idea.

1. **Try first.** Attempt the step yourself with the book in front of you. Give it a real try, about 15 minutes at most.
2. **Predict.** Before you run, write one sentence about what you think will happen.
3. **Run and read.** Run the game. Read any error in the Debugger carefully (see Appendix C).
4. **Ask for a hint.** If you are still stuck, ask for a hint only. Then try again.
5. **Ask for an explanation.** When it works, ask the assistant to explain it back, then explain it in your own words and compare. If your version differs, ask why.
6. **Change it.** Alter one thing on purpose (a number, a name) and predict the result. This proves you understand.
7. **Commit.** Save a git commit so you can come back to this point.

> **Try it:** Pick any script you wrote in this course. Ask Claude Code: "Explain each line of this script to me." Before reading the answer, write your own explanation. Then compare the two. Mark any line you got wrong and study it.

F.5 Using the Godot MCP server honestly

MCP (Model Context Protocol) is a standard way to let an AI tool use other programs. A Godot MCP server is a small helper that can let Claude Code launch your project, run it, and read its output, so answers come from real errors, not guesses.

The command from the original course is below. The package's public page (https://github.com/Coding-Solo/godot-mcp) shows the same `claude mcp add godot ... -- npx @coding-solo/godot-mcp` form and a `GODOT_PATH` setting, but check that page for the current instructions:

```
# Register the Godot MCP server with Claude Code
# GODOT_PATH tells it where the Godot program is
claude mcp add godot -e GODOT_PATH="C:/Godot/Godot.exe" -- npx @coding-solo/godot-mcp
```

```
# Shows every MCP server Claude Code knows about
claude mcp list
```

> **Warning:** Be honest about what this is. The server is a **community tool**. The author of this course has **not run** this exact command, and the package name, its options, and its features may have changed or may not work with your Godot version. Before you rely on it, check its page, read what it can do, and decide whether you trust it. An MCP server runs on your computer, so only install ones you trust.

How to check it works:

1. Open a terminal in your project folder and start Claude Code.
2. Ask: "Use the Godot MCP to run my project and tell me if there are errors."
3. A Godot window should open and Claude Code should report back what it saw.

If it fails:

- Make sure Node.js is installed (the command uses `npx`).
- Make sure the path to the Godot program is correct and uses forward slashes.
- Ask Claude Code to help diagnose the failure, or try another Godot MCP server.
- If no MCP works, you can still use Claude Code well: run the game yourself, copy the error text from the Debugger, and paste it into the prompt.

Be careful of one trap: if the MCP is not working, the assistant may still answer confidently, as if it had run the game. Ask it plainly, "Did you actually run the project, and what did you see?"

F.6 Git as your undo button

An AI can change many files at once. Git lets you see what changed and go back. Commit before you ask for any large change.

```
# See which files changed since your last commit
git status

# See the exact lines that changed
git diff

# Save a snapshot of the project
git add -A
git commit -m "Lab 7 working"

# Throw away all changes since your last commit
git restore .
```

> **Warning:** `git restore .` throws away your unsaved-to-git edits for tracked files, and it cannot be undone. Use it only when you want everything since the last commit gone. Run `git status` and `git diff` first to see what you would lose. New files that were never added to git are not removed by this command.

Habits that save you:

- Commit when something works, before trying something new.
- Read `git diff` before you accept an AI-made change.
- Keep `.godot/` out of git (add it to `.gitignore`).
- Never commit passwords or keys.

F.7 Academic honesty

If you use this book in a course, your instructor decides what help is allowed. Follow these principles.

- **Know the rules.** Read your course policy on AI tools. If it is unclear, ask your instructor before you use an assistant on graded work.
- **Do your own work.** The goal is for you to be able to write the code yourself. An exercise you did not understand teaches nothing, and the test will show it.
- **Say what you used.** If AI helped, say so. A short note is enough, for example: "I used Claude Code to explain the error in step 4 and to check my explanation of signals. I wrote the code myself."
- **Cite it when required.** Many institutions want AI help cited like any other source. Include the tool name, the date, and a short description of what you asked. Follow your institution's format.

- **Check what it says.** AI tools can be confidently wrong. Compare claims with the official Godot documentation at https://docs.godotengine.org and the Popochiu documentation at https://carenalgas.github.io/popochiu/.
- **Never submit AI output you cannot explain.** If you cannot explain a line, you do not own it.

F.8 One-page checklist

Print this page and keep it by your computer.

Setup (once)

- [ ] `CLAUDE.md` is in the project folder with my teaching rules.
- [ ] The project is in git, with at least one commit.
- [ ] `.godot/` is in `.gitignore`.
- [ ] I know my course's AI policy.
- [ ] (Optional) The Godot MCP server is installed and tested, and I know it is a community tool.

For every step of the course

- [ ] I tried the step myself first.
- [ ] I wrote down what I expected to happen.
- [ ] I read the real error in the Debugger.
- [ ] If stuck 15 minutes, I asked for a hint, not the answer.
- [ ] I can explain why the fix works, in my own words.
- [ ] I changed one thing on purpose to test my understanding.
- [ ] I read `git diff` before accepting an AI edit.
- [ ] I committed when it worked.

Before I ask

- [ ] My prompt says what I wanted, what happened, and what I tried.
- [ ] I pasted the exact error text.
- [ ] I did not paste any password, key, or private data.
- [ ] I asked for an explanation, not a finished lab.

When the assistant is wrong

- [ ] I asked it to check the official docs for my Godot and Popochiu versions.
- [ ] I looked up the function in the docs (F1 in the editor).
- [ ] I used `git restore .` only after reading `git diff` and only when I wanted to lose everything since the last commit.

Honesty

- [ ] I can explain every line I hand in.
- [ ] I said which AI help I used, if my course requires it.

F.9 After the course

You now know the pieces that a game like Weird Hotel uses: rooms, puzzles, items, talking, mini-games, bosses, and death. Next, sketch your own game on paper. Then build it one room at a time, with this book as your reference and your assistant as a tutor, not a replacement for you.

Appendix G: Weird Hotel Complete Script Listings

This appendix collects every script of the finished Weird Hotel, as it stands at the **end of Chapter 18**, after all the refactors. The chapters build these files a piece at a time: a Drawer with a local variable in Chapter 14 that moves to `Globals` in Chapter 15, a Hallway room script that grows in Chapters 14, 15, 16 and 18, a Ceiling that Chapter 18 decorates with sound, a shake and a transition. Here each file is shown once, complete, with every fragment merged in the place the chapters say.

Use it in three ways:

- **To check your work.** If something in your game behaves differently from the book, compare your file with the listing, line by line.
- **To catch up.** If you skipped a lab, these are the files you would have had at the end of it (but do the labs: the dock steps, polygons and markers are not in any script).
- **As a reference for your own game.** Each file is a small, working example of one Popochiu idea.

How to read the listings

- **Path** is where the file lives in the project, as the chapters give it. Files under `res://game/` are created by the Popochiu dock; never create, rename or move them by hand (Chapter 13).
- **Last changed** names the chapter (and section) that last changed the file.
- **Generated lines.** Where the chapters tell you to keep the lines the dock generated, the listing shows them: `@tool` (not for inventory items), the `extends` line, and for rooms, characters and items the two state lines `const Data := preload(...)` and `var state: Data = load(...)`. The dock's template also writes comment lines (such as `# @popochiu-docs-ignore-class`, `#region Virtual` and the explanation above each function). The listings leave those comments out to save space; if you kept them, that is fine, they change nothing.
- **Kept template functions.** Where a chapter says "keep the other functions your template wrote" (rooms and inventory items), the listing includes those functions, so the file is complete. A function that still calls `E.command_fallback()` shows the GUI's default reply; one with `pass` does nothing.
- **Indentation** is one tab per level, as in the editor. If you copy a listing, check that your editor did not turn the tabs into spaces (Appendix C).
- Every Popochiu call in these listings was checked against the Popochiu 2.2.1 source code, and every Godot call against the Godot 4.7 documentation (docs.godotengine.org/en/stable).

> **Note:** While merging the chapters into these listings, a few small differences between chapters were found and fixed in the chapters themselves, so the chapters and this appendix now agree. The list is at the end of the appendix (Section G.12).

G.1 Overview of the finished project

Files you write or change

File	Kind	Last changed
`res://game/popochiu_globals.gd`	Globals autoload (Popochiu)	18
`res://game/rooms/bedroom/room_bedroom.gd`	Room script	17
`res://game/rooms/hallway/room_hallway.gd`	Room script	18
`res://game/rooms/bedroom/props/drawer/prop_drawer.gd`	Prop	15
`res://game/rooms/bedroom/hotspots/door/hotspot_door.gd`	Hotspot	18
`res://game/rooms/hallway/hotspots/back_door/hotspot_back_door.gd`	Hotspot	14
`res://game/rooms/hallway/props/bone_pile/prop_bone_pile.gd`	Prop	15
`res://game/rooms/hallway/props/twine_ball/prop_twine_ball.gd`	Prop	15
`res://game/rooms/hallway/props/jar/prop_jar.gd`	Prop	18
`res://game/rooms/hallway/hotspots/ceiling/hotspot_ceiling.gd`	Hotspot	18
`res://game/characters/frog/character_frog.gd`	Character	16
`res://game/inventory_items/key/inventory_item_key.gd`	Inventory item	15
`res://game/inventory_items/bone/inventory_item_bone.gd`	Inventory item	15
`res://game/inventory_items/twine/inventory_item_twine.gd`	Inventory item	15
`res://game/inventory_items/bone_flute/inventory_item_bone_flute.gd`	Inventory item	15
`res://game/dialogs/chat_with_frog/dialog_chat_with_frog.gd`	Dialog	16
`res://minigames/catch_bugs.tscn` and `catch_bugs.gd`	Plain Godot scene	17
`res://minigames/boss_fight.tscn` and `boss_fight.gd`	Plain Godot scene	17
`res://settings.gd`	Your own autoload, `GameSettings`	18

Files the dock generated that you do not change

- `res://game/characters/player/character_player.gd`: the Player keeps its generated script. Everything the player does is driven from other scripts through `C.player`.
- `res://game/inventory_items/bug/inventory_item_bug.gd`: the Bug keeps its generated script (it is a keepsake; the Frog and the Ceiling react to it by name).
- The non-interactive `Background` props of both rooms keep the template script the dock wrote (`prop_background.gd`); they are not clickable, so nothing in it runs.
- The walkable areas (`Floor` in each room) keep their generated `walkable_area_floor.gd`.
- All the `*_state.gd` files are unchanged. The book keeps every fact in `Globals` (Chapter 15 explains why). Exercise 3 of Chapter 15 adds `drawer_searches` to `room_bedroom_state.gd`; exercises are not part of these listings.
- `res://game/autoloads/` (`r.gd`, `c.gd`, `i.gd`, `d.gd`, `a.gd`) and `res://game/popochiu_data.cfg` are regenerated by the dock. Never edit them.

G.2 Globals

Path: `res://game/popochiu_globals.gd` (an autoload named `Globals`, created by Popochiu with the single line `extends Node`). **Last changed:** Chapter 18 (18.1 quick save, 18.4 idle lines). Built up in Chapters 15 (15.4 to 15.6), 16 (16.4) and 17 (17.6).

```gdscript
# res://game/popochiu_globals.gd
extends Node

# Bedroom puzzle (Chapter 15)
var key_taken: bool = false
var door_unlocked: bool = false

# Hallway pickups (Chapter 15)
var bone_taken: bool = false
var twine_taken: bool = false

# Hallway story (Chapter 16)
var hallway_intro_seen: bool = false   # the entry cutscene has played
var frog_met: bool = false             # the player has spoken to the Frog before
var frog_who_asked: bool = false       # "Who are you?" has been asked
var flute_given: bool = false          # the flute has been traded for the key

# Mini-games, death and respawn (Chapter 17)
var boss_defeated: bool = false
var just_died: bool = false
var deaths: int = 0
var bug_caught: bool = false

# Idle lines (Chapter 18)
# A Dictionary is NOT saved automatically, so on_save/on_load below store it.
var click_counts: Dictionary = {}


# Craft the flute from the Bone and the Twine.
# Called by both Bone and Twine when they are combined.
func make_flute() -> void:
    if I.Bone.quantity_owned < 1 or I.Twine.quantity_owned < 1:
        return
    # State first.
    I.Bone.remove()
    I.Twine.remove()
    I.deselect_active()
    I.BoneFlute.add()
    await C.player.say("I made a flute out of a bone. Normal.")


# Returns the next line in the list. After the last line it repeats the last one.
func next_line(id: String, lines: Array) -> String:
    var n: int = int(click_counts.get(id, 0))
    click_counts[id] = n + 1
    return str(lines[mini(n, lines.size() - 1)])


# Called by Popochiu when the game is saved. Return simple data only.
func on_save() -> Dictionary:
    return {"click_counts": click_counts}


# Called by Popochiu when a game is loaded, with what on_save returned.
func on_load(data: Dictionary) -> void:
    if data.has("click_counts"):
        click_counts = data["click_counts"]


# Quick save (F9) and quick load (F10), Chapter 18.
func _unhandled_input(event: InputEvent) -> void:
```

```
    if G.is_blocked:
        return                          # no saving during mini-games or cutscenes
    if event.is_action_pressed("quick_save"):
        E.save_game(1, "Quick save")
        print("Saved to slot 1")
    elif event.is_action_pressed("quick_load"):
        if E.has_save():
            E.load_game(1)
        else:
            print("No save to load")
```

Notes:

- The order of the blocks does not matter to Godot; the order above follows the chapters. Chapter 17 (17.6) shows a short Globals block with only `key_taken`, `door_unlocked` and its four new variables: that block is a reminder, not the whole file, and the chapter says to keep the other lines.
- `lamp_color` from Section 18.2 is only an example of a complex type and is not part of the game. If you added it, merge its key into the same `on_save()` and `on_load()` (a script cannot have two functions with the same name).
- Every variable has a written type. Popochiu saves `Globals` variables only when they are statically typed `bool`, `int`, `float`, `String` or `Array`; that is why `click_counts` (a `Dictionary`) needs `on_save()` and `on_load()`.
- `Globals` uses no underscore for `on_save` and `on_load`; the state scripts of rooms, characters and items use `_on_save` and `_on_load` (Chapter 18).

G.3 Room scripts

Bedroom

Path: `res://game/rooms/bedroom/room_bedroom.gd` **Last changed:** Chapter 17 (17.6, the Bed marker and the waking-up lines). Chapter 14 (14.7) wrote the first `_on_room_entered()`; Chapter 17 keeps its `FromHallway` branch as an `elif`.

```
# res://game/rooms/bedroom/room_bedroom.gd
@tool
extends PopochiuRoom

const Data := preload('room_bedroom_state.gd')

var state: Data = load("res://game/rooms/bedroom/room_bedroom.tres")


func _on_room_entered() -> void:
    if Globals.just_died:
        await C.player.teleport_to_marker("Bed")
    elif C.player.last_room == "Hallway":
        await C.player.teleport_to_marker("FromHallway")


func _on_room_transition_finished() -> void:
    if Globals.just_died:
        Globals.just_died = false
        await C.player.say("Ow. Everything went white.")
        await C.player.say("I am in bed again. The ceiling can wait.")


func _on_room_exited() -> void:
    pass
```

```gdscript
func _on_restore_from_savegame() -> void:
    pass
```

Notes:

- On a new game `last_room` is an empty string, so neither branch runs and the player starts where you placed `CharacterPlayer` in Lab 14 (step 11).
- The temporary test cheat from Section 15.7 also used `_on_room_transition_finished()` in this script. It must be gone: the finished game has only the function above (two functions with the same name are an error).

Hallway

Path: `res://game/rooms/hallway/room_hallway.gd` **Last changed:** Chapter 18 (18.2, the music). Built in Chapter 14 (Lab 14 step 25, the `FromBedroom` arrival), Chapter 15 (15.5, `_apply_state()`) and Chapter 16 (16.6, the intro).

```gdscript
# res://game/rooms/hallway/room_hallway.gd
@tool
extends PopochiuRoom

const Data := preload('room_hallway_state.gd')

var state: Data = load("res://game/rooms/hallway/room_hallway.tres")


func _on_room_entered() -> void:
    if C.player.last_room == "Bedroom":
        await C.player.teleport_to_marker("FromBedroom")
    _apply_state()
    _start_music()


func _on_room_transition_finished() -> void:
    if Globals.hallway_intro_seen:
        return
    # Set the flag first: a skip or a save cannot make the scene play twice.
    Globals.hallway_intro_seen = true
    # The player stands left of the Frog, so they face right and he faces left.
    # Swap the two face calls if your layout is the other way round.
    await E.cutscene([
        E.camera.queue_change_zoom(Vector2(1.15, 1.15), 0.8),
        C.player.queue_face_right(),
        C.Frog.queue_face_left(),
        C.Frog.queue_say("Ah. The new guest."),
        C.Frog.queue_say("You are eleven minutes late."),
        E.camera.queue_shake(8.0, 0.5),
        C.Frog.queue_say("Also, the carpet bites. Mind your ankles."),
        C.player.queue_say("Noted."),
        E.camera.queue_change_zoom(Vector2.ONE, 0.6),
    ])


func _on_room_exited() -> void:
    pass


func _on_restore_from_savegame() -> void:
    _apply_state()
    _start_music()
```

```
# Makes the room look like the flags say (Chapter 15). Safe to run twice.
func _apply_state() -> void:
    if Globals.bone_taken:
        get_prop("BonePile").disable()
    if Globals.twine_taken:
        get_prop("TwineBall").disable()


func _start_music() -> void:
    if not A.mx_hallway.is_playing():
        A.mx_hallway.play()
```

Notes:

- `A.mx_hallway` exists only after you put `mx_hallway.ogg` (or another supported format) under `res://game/` and click **Scan audio files** in the Audio tab (Chapter 18, 18.3).
- The intro does not replay after a load, because `_on_room_transition_finished()` is not called on load (Chapter 16).

G.4 Props and hotspots in the Bedroom

Drawer (prop)

Path: `res://game/rooms/bedroom/props/drawer/prop_drawer.gd` **Last changed:** Chapter 15 (15.4, the refactor to `Globals`). The Chapter 14 version with a local `key_taken` is gone.

```
# res://game/rooms/bedroom/props/drawer/prop_drawer.gd
@tool
extends PopochiuProp

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.key_taken:
        await C.player.say("Just old socks now.")
        return
    # State first, speech second.
    I.Key.add()
    Globals.key_taken = true
    await C.player.say("A key! It is warm. Why is it warm?")

func _on_right_click() -> void:
    await C.player.say("A drawer that hums faintly.")
```

Door (hotspot)

Path: `res://game/rooms/bedroom/hotspots/door/hotspot_door.gd` **Last changed:** Chapter 18 (18.4, the idle lines replace the locked line). The rest is the Chapter 15 version (15.3).

```
# res://game/rooms/bedroom/hotspots/door/hotspot_door.gd
@tool
extends PopochiuHotspot

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.door_unlocked:
        await R.goto_room("Hallway")
    else:
        await C.player.say(Globals.next_line("Door", [
            "It is locked.",
```

```
            "Still locked.",
            "I keep checking. It keeps being locked.",
            "The door and I have reached an understanding. It stays locked.",
        ]))

func _on_right_click() -> void:
    if Globals.door_unlocked:
        await C.player.say("An unlocked door. The keyhole looks relieved.")
    else:
        await C.player.say("A locked door. It would like a key.")

func _on_item_used(item: PopochiuInventoryItem) -> void:
    await C.player.walk_to_clicked()
    if item == I.Key:
        if Globals.door_unlocked:
            # Already open: do NOT consume the key.
            await C.player.say("It is already unlocked. The door looks smug.")
            return
        # Change state first, then talk.
        I.Key.remove()
        Globals.door_unlocked = true
        await C.player.say("It fits!")
    else:
        await C.player.say("That won't open it.")
```

G.5 Props and hotspots in the Hallway

BackDoor (hotspot)

Path: `res://game/rooms/hallway/hotspots/back_door/hotspot_back_door.gd` **Last changed:** Chapter 14 (Lab 14, step 24). Its job is to take the player back to the Bedroom, where the Bedroom's `_on_room_entered()` places them at `FromHallway`.

```
# res://game/rooms/hallway/hotspots/back_door/hotspot_back_door.gd
@tool
extends PopochiuHotspot

func _on_click() -> void:
    await C.player.walk_to_clicked()
    await R.goto_room("Bedroom")

func _on_right_click() -> void:
    await C.player.say("The door I came from. It waved.")
```

BonePile (prop)

Path: `res://game/rooms/hallway/props/bone_pile/prop_bone_pile.gd` **Last changed:** Chapter 15 (15.5).

```
# res://game/rooms/hallway/props/bone_pile/prop_bone_pile.gd
@tool
extends PopochiuProp

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.bone_taken:
        # A hidden prop should not be clickable, but this makes sure.
        return
    I.Bone.add()
    Globals.bone_taken = true
    disable()
    await C.player.say("A bone. It was just lying there, looking available.")
```

```
func _on_right_click() -> void:
    if Globals.bone_taken:
        return
    await C.player.say("A bone. Someone chewed it politely.")
```

TwineBall (prop)

Path: `res://game/rooms/hallway/props/twine_ball/prop_twine_ball.gd` **Last changed:** Chapter 15 (15.5).

```
# res://game/rooms/hallway/props/twine_ball/prop_twine_ball.gd
@tool
extends PopochiuProp

func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.twine_taken:
        return
    I.Twine.add()
    Globals.twine_taken = true
    disable()
    await C.player.say("A ball of twine. It is slightly damp. I will not ask.")

func _on_right_click() -> void:
    if Globals.twine_taken:
        return
    await C.player.say("Twine. Good for tying things together.")
```

Jar (prop)

Path: `res://game/rooms/hallway/props/jar/prop_jar.gd` **Last changed:** Chapter 18 (18.3, `A.sfx_pickup.play()` in the `if won:` branch). The rest is Chapter 17 (17.4).

```
# res://game/rooms/hallway/props/jar/prop_jar.gd
# The Jar prop in the Hallway: starts the catch-the-bug mini-game.
@tool
extends PopochiuProp

const CATCH_BUGS := preload("res://minigames/catch_bugs.tscn")


func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.bug_caught:
        await C.player.say("The jar is empty now. The bug is in my pocket.")
        return
    await C.player.say("Something in the jar wants out.")

    var won: bool = await _play_catch_bugs()

    if won:
        # State first, then talk.
        Globals.bug_caught = true
        I.Bug.add()
        A.sfx_pickup.play()
        await C.player.say("Got it. It is wiggling in my pocket.")
    else:
        await C.player.say("Too slow. I can click the jar to try again.")


func _on_right_click() -> void:
    if Globals.bug_caught:
        await C.player.say("An empty jar. It looks proud of itself.")
    else:
```

```
        await C.player.say("Something in the jar wants out.")


func _on_item_used(_item: PopochiuInventoryItem) -> void:
    await C.player.say("The jar does not want that. It wants a bug.")


# Starts the mini-game, waits for the result, cleans up, and returns the result.
func _play_catch_bugs() -> bool:
    var game: CatchBugsGame = CATCH_BUGS.instantiate() as CatchBugsGame
    G.block()                           # room and GUI ignore the mouse
    Cursor.unblock()                    # but the pointer must still work...
    Cursor.show_cursor("normal")        # ...and look normal, not "wait"
    get_tree().root.add_child(game)     # _ready() of the mini-game runs now
    var won: bool = await game.finished # pause here until the result arrives
    G.unblock()                         # give the player the interface back
    game.queue_free()                   # the caller frees the mini-game
    return won
```

Ceiling (hotspot)

Path: `res://game/rooms/hallway/hotspots/ceiling/hotspot_ceiling.gd` **Last changed:** Chapter 18: 18.3 (`sfx_win` and `sfx_lose`), 18.4 (the camera shake after the first line, the white `OUT` transition before the fight, `T.hide_curtain()` after the mini-game is added) and 18.5 (`game.shake_enabled`). The rest is Chapter 17 (17.6).

```
# res://game/rooms/hallway/hotspots/ceiling/hotspot_ceiling.gd
# The Ceiling hotspot in the Hallway: starts the boss fight.
@tool
extends PopochiuHotspot

const BOSS_FIGHT := preload("res://minigames/boss_fight.tscn")


func _on_click() -> void:
    await C.player.walk_to_clicked()
    if Globals.boss_defeated:
        await C.player.say("The ceiling is quiet now.")
        return
    await C.player.say("The ceiling is looking at me.")
    # Chapter 18: a rumble, unless the player switched shaking off.
    if GameSettings.screen_shake:
        await E.camera.shake(8.0, 0.8)
    else:
        await E.wait(0.8)
    # Chapter 18: cover the room in white; _play_boss_fight uncovers it.
    await T.play_transition("fade", 0.4, T.PLAY_MODE.OUT, Color.WHITE)

    var won: bool = await _play_boss_fight()

    if won:
        # State first, then the ending.
        Globals.boss_defeated = true
        A.sfx_win.play()
        await C.player.say("It stopped. I think I won?")
        await _play_ending()
    else:
        Globals.deaths += 1
        Globals.just_died = true
        A.sfx_lose.play()
        await R.goto_room("Bedroom")


func _on_right_click() -> void:
    if Globals.boss_defeated:
        await C.player.say("A ceiling. It has gone back to being a ceiling.")
    else:
```

```
        await C.player.say("The ceiling is looking at me.")


func _on_item_used(_item: PopochiuInventoryItem) -> void:
    await C.player.say("I am not throwing that at the ceiling. It would only be flattered.")


# The end of the game. The player can keep walking around afterwards.
func _play_ending() -> void:
    await C.Frog.say("Forty years that ceiling has been like that.")
    if I.is_item_in_inventory("Bug"):
        await C.Frog.say("And you kept the bug. Good. Bugs are tenants too.")
    await C.Frog.say("Your rent is waived. Mind the carpet on the way out.")
    await G.show_system_text("THE END. Thank you for staying at the Weird Hotel.")


func _play_boss_fight() -> bool:
    var game: BossFightGame = BOSS_FIGHT.instantiate() as BossFightGame
    game.shake_enabled = GameSettings.screen_shake
    G.block()
    Cursor.unblock()
    Cursor.show_cursor("normal")
    get_tree().root.add_child(game)
    T.hide_curtain()
    var won: bool = await game.finished
    G.unblock()
    game.queue_free()
    return won
```

Notes:

- On a loss, nothing runs after `await R.goto_room("Bedroom")`: every fact (`deaths`, `just_died`) is set before it, and the Bedroom does the waking up (Chapter 17).
- The sound calls do not use `await`, so the speech and the room change start while the sound plays.

G.6 Characters

Player

Path: `res://game/characters/player/character_player.gd` **Last changed:** never; it is the script the dock generated in Chapter 14 (Lab 14, step 3). Only the scene changed: the `Sprite2D` texture and its **Offset** `(0, -80)` (Lab 14, step 5).

Frog

Path: `res://game/characters/frog/character_frog.gd` **Last changed:** Chapter 16 (16.4, "The Frog's click, with flags", including `_on_item_used`). The chapter says to replace the script and keep the generated header lines; the other template functions (`_on_room_set`, `_play_idle` and so on) are not needed, because the base class already does what they did.

```
# res://game/characters/frog/character_frog.gd
@tool
extends PopochiuCharacter

const Data := preload('character_frog_state.gd')

var state: Data = load("res://game/characters/frog/character_frog.tres")

func _on_click() -> void:
    await C.player.walk_to_clicked()
```

```
    await C.player.face_clicked()
    if Globals.flute_given:
        await C.Frog.say("The rent is paid in music. You paid.")
    elif Globals.frog_met:
        await C.Frog.say("You again. Rent.")
    else:
        Globals.frog_met = true
        await C.Frog.say("Yes? I am busy. Being a frog.")
    await D.ChatWithFrog.start()

func _on_right_click() -> void:
    await C.player.say("A frog in a cardigan. He does not blink at the right times.")

# An item used on the Frog. We compare script names (strings) so this script
# also works before the Bug item exists (Chapter 17 creates it).
func _on_item_used(item: PopochiuInventoryItem) -> void:
    await C.player.walk_to_clicked()
    await C.player.face_clicked()
    match item.script_name:
        "Bug":
            # The hint promised in Chapter 12: the Bug is kept, the Frog points at the flute.
            await C.Frog.say("A bug. Nice. But I want music, not snacks.")
        "BoneFlute":
            await C.Frog.say("Ask me properly.")
        _:
            await C.Frog.say("I do not accept gifts. Except flutes.")
```

G.7 Inventory items

Inventory item scripts do **not** start with `@tool`. Chapter 15 tells you to keep the other functions your template wrote, so each listing below is the full file: the template's functions plus the ones the chapter adds or replaces.

Key

Path: `res://game/inventory_items/key/inventory_item_key.gd` **Last changed:** Chapter 15 (15.2, the right-click look line).

```
# res://game/inventory_items/key/inventory_item_key.gd
extends PopochiuInventoryItem

const Data := preload('inventory_item_key_state.gd')

var state: Data = load("res://game/inventory_items/key/inventory_item_key.tres")


# Left as the template wrote it: in Simple Click this selects the item.
func _on_click() -> void:
    E.command_fallback()


func _on_right_click() -> void:
    await C.player.say("It is warm. Keys should not be warm.")


func _on_middle_click() -> void:
    E.command_fallback()


func _on_item_used(_item: PopochiuInventoryItem) -> void:
    E.command_fallback()


func _on_added_to_inventory() -> void:
    super()
```

```
func _on_discard() -> void:
    super()
```

Bone

Path: `res://game/inventory_items/bone/inventory_item_bone.gd` **Last changed:** Chapter 15 (15.6, the combination).

```
# res://game/inventory_items/bone/inventory_item_bone.gd
extends PopochiuInventoryItem

const Data := preload('inventory_item_bone_state.gd')

var state: Data = load("res://game/inventory_items/bone/inventory_item_bone.tres")


func _on_click() -> void:
    E.command_fallback()


func _on_right_click() -> void:
    E.command_fallback()


func _on_middle_click() -> void:
    E.command_fallback()


func _on_item_used(item: PopochiuInventoryItem) -> void:
    if item == I.Twine:
        await Globals.make_flute()
    else:
        await C.player.say("Those don't go together.")


func _on_added_to_inventory() -> void:
    super()


func _on_discard() -> void:
    super()
```

Twine

Path: `res://game/inventory_items/twine/inventory_item_twine.gd` **Last changed:** Chapter 15 (15.6, the combination). The item is called Twine, not String, because `String` is a GDScript type and `I.String` would clash with it (Chapter 15, 15.5).

```
# res://game/inventory_items/twine/inventory_item_twine.gd
extends PopochiuInventoryItem

const Data := preload('inventory_item_twine_state.gd')

var state: Data = load("res://game/inventory_items/twine/inventory_item_twine.tres")


func _on_click() -> void:
    E.command_fallback()


func _on_right_click() -> void:
```

```
    E.command_fallback()


func _on_middle_click() -> void:
    E.command_fallback()


func _on_item_used(item: PopochiuInventoryItem) -> void:
    if item == I.Bone:
        await Globals.make_flute()
    else:
        await C.player.say("Those don't go together.")


func _on_added_to_inventory() -> void:
    super()


func _on_discard() -> void:
    super()
```

BoneFlute

Path: `res://game/inventory_items/bone_flute/inventory_item_bone_flute.gd` **Last changed:** Chapter 15 (15.6).

```
# res://game/inventory_items/bone_flute/inventory_item_bone_flute.gd
extends PopochiuInventoryItem

const Data := preload('inventory_item_bone_flute_state.gd')

var state: Data = load("res://game/inventory_items/bone_flute/inventory_item_bone_flute.tres")


func _on_click() -> void:
    E.command_fallback()


func _on_right_click() -> void:
    await C.player.say("It plays one note. The note is sad.")


func _on_middle_click() -> void:
    E.command_fallback()


func _on_item_used(_item: PopochiuInventoryItem) -> void:
    await C.player.say("I should not do that to a flute.")


func _on_added_to_inventory() -> void:
    super()


func _on_discard() -> void:
    super()
```

Bug

Path: `res://game/inventory_items/bug/inventory_item_bug.gd` **Last changed:** never; the generated script (Chapter 17, Lab 17 step 1). Its right-click shows Simple Click's default "Nothing to see in this item". Giving it a look line is a good extra exercise.

G.8 The dialog

ChatWithFrog

Path: `res://game/dialogs/chat_with_frog/dialog_chat_with_frog.gd` (the options live in `dialog_chat_with_frog.tres`) **Last changed:** Chapter 16 (16.4, "The complete dialog script"). The dialog template has no state lines, so the header is only `@tool` and `extends PopochiuDialog`.

Options, set in the Inspector of `dialog_chat_with_frog.tres` (Chapter 16, 16.3):

id	text	visible
`Who`	Who are you?	yes
`Flute`	Do you need a flute?	no
`Bye`	Goodbye.	yes

```gdscript
# res://game/dialogs/chat_with_frog/dialog_chat_with_frog.gd
@tool
extends PopochiuDialog

func _on_start() -> void:
    if Globals.frog_who_asked:
        turn_off_options(["Who"])
        turn_on_options(["Flute"])
    if Globals.flute_given:
        turn_off_options(["Flute"])
    await E.get_tree().process_frame

func _option_selected(opt: PopochiuDialogOption) -> void:
    match opt.id:
        "Who":
            await D.say_selected()
            # State first, then talk.
            Globals.frog_who_asked = true
            turn_off_options(["Who"])
            turn_on_options(["Flute"])
            await C.Frog.say("I am the landlord. Mind the carpet. It bites.")
            await C.Frog.say("I used to have a band. Now I have rent.")
        "Flute":
            await D.say_selected()
            await _flute_talk()
            if Globals.flute_given:
                stop()
        "Bye":
            await D.say_selected()
            await C.Frog.say("Rent is due. Always.")
            stop()
        _:
            stop()
    _show_options()

func _flute_talk() -> void:
    if not I.is_item_in_inventory("BoneFlute"):
        await C.Frog.say("No flute. Sad.")
        return
    if I.is_item_in_inventory("Key"):
        # Defensive: never take the flute unless the reward can be given.
        await C.Frog.say("Your hands are full of key. Come back with room.")
        return
    # Record it, remove the flute, give the key, all together, before any speech.
    # (remove() and add() change the inventory at once; only their animation is awaited.)
    Globals.flute_given = true
    turn_off_options(["Flute"])
```

```
    I.BoneFlute.remove()
    I.Key.add()
    await C.Frog.say("A bone flute. Hm.")
    await C.player.say("It plays one sad note.")
    await C.Frog.say("Perfect. Take my spare key.")
```

G.9 The mini-games

Both mini-games are ordinary Godot scenes in `res://minigames/`, outside `res://game/`, with no Popochiu code inside (Chapter 17). Build the scene trees with the exact node names; the scripts find the nodes by path.

Catch the bug

Scene: `res://minigames/catch_bugs.tscn`. **Script:** `res://minigames/catch_bugs.gd`. **Last changed:** Chapter 17 (17.4).

```
CatchBugs (CanvasLayer)          script: catch_bugs.gd, Layer = 50
└── Blocker (ColorRect)          Full Rect, color black with alpha 0.6, Mouse Filter = Stop
    ├── Target (Button)          text "BUG", Custom Minimum Size 64 x 64
    └── Label (Label)            top-left, font size about 28, white
```

```gdscript
# res://minigames/catch_bugs.gd
# Attached to the root node (CanvasLayer) of catch_bugs.tscn.
# Contract: emits finished(won) exactly once. The caller frees this scene.
class_name CatchBugsGame
extends CanvasLayer

signal finished(won: bool)

const NEEDED: int = 5            # hits needed to win
const TIME_LIMIT: float = 10.0   # seconds
const START_HOP: float = 1.2     # seconds before the bug hops by itself
const MIN_HOP: float = 0.5       # the hop gets this fast at the end
const TOP_MARGIN: float = 48.0   # keeps the bug away from the label
const END_PAUSE: float = 0.7     # show the result for a moment

var hits: int = 0
var time_left: float = TIME_LIMIT
var hop_left: float = START_HOP
var done: bool = false           # stops the game from ending twice

@onready var target: Button = $Blocker/Target
@onready var label: Label = $Blocker/Label


func _ready() -> void:
    target.pressed.connect(_on_target_pressed)
    _move_target()
    _update_label()


func _process(delta: float) -> void:
    if done:
        return
    time_left -= delta
    hop_left -= delta
    if time_left <= 0.0:
        time_left = 0.0
        _update_label()
        _end(false)
        return
    if hop_left <= 0.0:
        _move_target()
```

```
    _update_label()


func _on_target_pressed() -> void:
    if done:
        return
    hits += 1
    if hits >= NEEDED:
        _update_label()
        _end(true)
    else:
        _move_target()
        _update_label()


# Jump the bug to a random spot and decide when it hops next.
func _move_target() -> void:
    var area: Vector2 = get_viewport().get_visible_rect().size
    var max_x: float = maxf(area.x - target.size.x, 0.0)
    var max_y: float = maxf(area.y - target.size.y, TOP_MARGIN)
    target.position = Vector2(randf_range(0.0, max_x), randf_range(TOP_MARGIN, max_y))
    # The more bugs caught, the faster it hops on its own.
    hop_left = lerpf(START_HOP, MIN_HOP, float(hits) / float(NEEDED))


func _update_label() -> void:
    label.text = "Bugs: %d/%d    Time: %.1f" % [hits, NEEDED, time_left]


# The ONLY place that ends the game, and the only place that emits.
func _end(won: bool) -> void:
    if done:
        return
    done = true
    target.disabled = true
    label.text = "Got it!" if won else "Too slow!"
    await get_tree().create_timer(END_PAUSE).timeout
    finished.emit(won)
    if get_tree().current_scene == self:
        print("CatchBugs finished, won = ", won)   # only when run alone with F6
```

The boss fight

Scene: `res://minigames/boss_fight.tscn`. **Script:** `res://minigames/boss_fight.gd`. **Last changed:** Chapter 17 (17.5). Chapter 18 does not edit the script; it sets `shake_enabled` from the Ceiling before adding the scene (18.5).

```
BossFight (CanvasLayer)          script: boss_fight.gd, Layer = 50
├── Backdrop (ColorRect)         Full Rect, then offsets -32/-32/32/32; dark, alpha 0.85, Mouse Filter
= Stop
└── Arena (Control)              Full Rect, Mouse Filter = Ignore
    ├── Hero (ColorRect)         size 24 x 24, color yellow
    └── Label (Label)            top-left, white, font size 24
```

```
# res://minigames/boss_fight.gd
# Attached to the root node (CanvasLayer) of boss_fight.tscn.
# Contract: emits finished(won) exactly once. The caller frees this scene.
class_name BossFightGame
extends CanvasLayer

signal finished(won: bool)

const SURVIVE_TIME: float = 20.0
const START_LIVES: int = 3
const HERO_SIZE: Vector2 = Vector2(24, 24)
```

```
const BLOCK_SIZE: Vector2 = Vector2(20, 20)
const HERO_SPEED: float = 320.0       # pixels per second
const TOP_LIMIT: float = 0.35         # hero cannot go above 35% of the screen height
const FALL_START: float = 200.0       # block speed at the start
const FALL_END: float = 520.0         # block speed at the end
const SPAWN_START: float = 0.55       # seconds between blocks at the start
const SPAWN_END: float = 0.25         # seconds between blocks at the end
const INVULN_TIME: float = 1.0        # safe time after a hit
const END_PAUSE: float = 0.8
const BLOCK_COLOR: Color = Color(0.9, 0.15, 0.2)
const ACTION_KEYS: Dictionary = {
    "move_left": [KEY_A, KEY_LEFT],
    "move_right": [KEY_D, KEY_RIGHT],
    "move_up": [KEY_W, KEY_UP],
    "move_down": [KEY_S, KEY_DOWN],
}

# The caller may switch this off (Chapter 18, accessibility) before adding the scene.
var shake_enabled: bool = true

var lives: int = START_LIVES
var time_left: float = SURVIVE_TIME
var spawn_left: float = 0.0
var invuln_left: float = 0.0
var done: bool = false
var blocks: Array[ColorRect] = []
var _flash_tween: Tween
var _shake_tween: Tween

@onready var arena: Control = $Arena
@onready var hero: ColorRect = $Arena/Hero
@onready var label: Label = $Arena/Label


func _ready() -> void:
    _ensure_input_actions()
    hero.size = HERO_SIZE
    var area: Vector2 = _arena_size()
    hero.position = Vector2((area.x - HERO_SIZE.x) / 2.0, area.y - HERO_SIZE.y - 24.0)
    _update_label()


func _process(delta: float) -> void:
    if done:
        return
    # progress goes from 0.0 at the start to 1.0 when the time is up
    var progress: float = 1.0 - time_left / SURVIVE_TIME
    invuln_left = maxf(invuln_left - delta, 0.0)
    _move_hero(delta)
    _spawn_blocks(delta, progress)
    _move_blocks(delta, progress)
    if done:                          # a hit on this frame may have ended the game
        return
    time_left -= delta
    if time_left <= 0.0:
        time_left = 0.0
        _update_label()
        _end(true)
        return
    _update_label()


func _move_hero(delta: float) -> void:
    # get_vector turns the four actions into one direction, such as (1, 0) for right
    var dir: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    var area: Vector2 = _arena_size()
    var min_pos: Vector2 = Vector2(0.0, area.y * TOP_LIMIT)
    var max_pos: Vector2 = area - HERO_SIZE
    hero.position = (hero.position + dir * HERO_SPEED * delta).clamp(min_pos, max_pos)
```

```
func _spawn_blocks(delta: float, progress: float) -> void:
    spawn_left -= delta
    if spawn_left > 0.0:
        return
    spawn_left = lerpf(SPAWN_START, SPAWN_END, progress)
    var b: ColorRect = ColorRect.new()
    b.color = BLOCK_COLOR
    b.size = BLOCK_SIZE
    b.mouse_filter = Control.MOUSE_FILTER_IGNORE
    b.position = Vector2(randf_range(0.0, _arena_size().x - BLOCK_SIZE.x), 0.0)
    arena.add_child(b)
    blocks.append(b)


func _move_blocks(delta: float, progress: float) -> void:
    var fall_speed: float = lerpf(FALL_START, FALL_END, progress)
    var arena_h: float = _arena_size().y
    var hero_rect: Rect2 = hero.get_global_rect()
    # Walk the array backwards so removing an entry does not skip the next one.
    for i in range(blocks.size() - 1, -1, -1):
        var entry: Variant = blocks[i]
        if not is_instance_valid(entry):
            blocks.remove_at(i)
            continue
        var b: ColorRect = entry
        b.position.y += fall_speed * delta
        if b.position.y > arena_h:
            blocks.remove_at(i)
            b.queue_free()
        elif invuln_left <= 0.0 and b.get_global_rect().intersects(hero_rect):
            blocks.remove_at(i)
            b.queue_free()
            _hit()
            if done:
                return


func _hit() -> void:
    lives -= 1
    invuln_left = INVULN_TIME
    _flash_hero()
    _shake(10.0, 0.3)
    _update_label()
    if lives <= 0:
        _end(false)


# Game feel: the hero turns red and fades back to normal while it is safe.
func _flash_hero() -> void:
    if _flash_tween != null and _flash_tween.is_valid():
        _flash_tween.kill()
    hero.modulate = Color(1.0, 0.25, 0.25)
    _flash_tween = create_tween()
    _flash_tween.tween_property(hero, "modulate", Color.WHITE, INVULN_TIME)


# Game feel: shake the whole layer by moving its offset to random spots.
func _shake(strength: float, duration: float) -> void:
    if not shake_enabled:
        return
    if _shake_tween != null and _shake_tween.is_valid():
        _shake_tween.kill()
    offset = Vector2.ZERO
    var steps: int = 6
    var step_time: float = duration / float(steps)
    _shake_tween = create_tween()
    for i in range(steps):
        var amount: float = strength * (1.0 - float(i) / float(steps))
        var jitter: Vector2 = Vector2(randf_range(-amount, amount), randf_range(-amount, amount))
```

```
        _shake_tween.tween_property(self, "offset", jitter, step_time)
    _shake_tween.tween_property(self, "offset", Vector2.ZERO, step_time)


func _update_label() -> void:
    label.text = "Lives: %d    Survive: %.1f" % [lives, time_left]


func _arena_size() -> Vector2:
    return get_viewport().get_visible_rect().size


# The ONLY place that ends the fight, and the only place that emits.
func _end(won: bool) -> void:
    if done:
        return
    done = true
    for b in blocks:
        if is_instance_valid(b):
            b.queue_free()
    blocks.clear()
    label.text = "You survived!" if won else "Knocked out..."
    await get_tree().create_timer(END_PAUSE).timeout
    finished.emit(won)
    if get_tree().current_scene == self:
        print("BossFight finished, won = ", won)   # only when run alone with F6


# Safety net: add the movement actions if the project does not have them.
func _ensure_input_actions() -> void:
    for action in ACTION_KEYS:
        if InputMap.has_action(action):
            continue
        push_warning("Input action '%s' is missing from the Input Map. Adding it for this run." %
action)
        InputMap.add_action(action)
        for key in ACTION_KEYS[action]:
            var ev: InputEventKey = InputEventKey.new()
            ev.physical_keycode = key
            InputMap.action_add_event(action, ev)
```

G.10 Your own autoload: GameSettings

Path: `res://settings.gd`, registered in **Project > Project Settings > Globals > Autoload** with the name `GameSettings`. **Last changed:** Chapter 18 (18.5). It stores player options in `user://settings.cfg`, outside the save games.

```
# res://settings.gd
# Autoload named GameSettings. Player options, stored outside the save games.
extends Node

const PATH: String = "user://settings.cfg"

var screen_shake: bool = true


func _ready() -> void:
    load_settings()


func load_settings() -> void:
    var cfg: ConfigFile = ConfigFile.new()
    if cfg.load(PATH) != OK:
        return                          # no file yet: keep the defaults
    screen_shake = bool(cfg.get_value("accessibility", "screen_shake", true))
```

```
func save_settings() -> void:
    var cfg: ConfigFile = ConfigFile.new()
    cfg.set_value("accessibility", "screen_shake", screen_shake)
    cfg.save(PATH)
```

G.11 Project settings, names and the final test

Input Map actions

Set in **Project > Project Settings > InputMap**.

Action	Keys	Added in	Used by
`move_left`	A, Left arrow	17 (17.5)	`boss_fight.gd`
`move_right`	D, Right arrow	17 (17.5)	`boss_fight.gd`
`move_up`	W, Up arrow	17 (17.5)	`boss_fight.gd`
`move_down`	S, Down arrow	17 (17.5)	`boss_fight.gd`
`quick_save`	F9	18 (18.1)	`Globals._unhandled_input`
`quick_load`	F10	18 (18.1)	`Globals._unhandled_input`
`popochiu-skip`	Esc	Popochiu (already there)	Skipping `E.cutscene`

Every Globals variable

Variable	Type	Added in	Meaning	Saved
`key_taken`	`bool`	15	The drawer has been emptied	automatically
`door_unlocked`	`bool`	15	The Bedroom door is open	automatically
`bone_taken`	`bool`	15	The bone pile has been picked up	automatically
`twine_taken`	`bool`	15	The ball of twine has been picked up	automatically
`hallway_intro_seen`	`bool`	16	The Hallway intro has played	automatically
`frog_met`	`bool`	16	The player has greeted the Frog	automatically
`frog_who_asked`	`bool`	16	"Who are you?" has been asked	automatically
`flute_given`	`bool`	16	The flute has been traded for the spare Key	automatically
`bug_caught`	`bool`	17	The jar is empty; the Bug was won	automatically
`boss_defeated`	`bool`	17	The ceiling fight was won (the ending has played)	automatically
`just_died`	`bool`	17	The Bedroom should play the waking-up lines	automatically
`deaths`	`int`	17	How many times the player lost the boss fight	automatically

Variable	Type	Added in	Meaning	Saved
`click_counts`	`Dictionary`	18	Idle-line counters	through `on_save()` / `on_load()`

Functions in `Globals`: `make_flute()` (15), `next_line()` (18), `on_save()` and `on_load()` (18), `_unhandled_input()` (18).

Rooms, objects and markers

Names are exact; scripts use them as strings (`goto_room("Hallway")`, `get_prop("BonePile")`, `teleport_to_marker("Bed")`).

Room	Kind	Name	Notes
Bedroom (main scene)	Prop, no interaction	`Background`	`bedroom_bg.png`
	Walkable area	`Floor`	Bottom third of the picture
	Prop	`Drawer`	Gives the Key once
	Hotspot	`Door`	Locked until the Key is used; leads to the Hallway
	Marker	`FromHallway`	Arrival from the Hallway, in front of the Door
	Marker	`Bed`	Respawn point after losing the boss fight (Chapter 17)
	Character	`CharacterPlayer`	The Player's start position on a new game
Hallway	Prop, no interaction	`Background`	`hallway_bg.png`
	Walkable area	`Floor`	
	Hotspot	`BackDoor`	Leads back to the Bedroom
	Marker	`FromBedroom`	Arrival from the Bedroom
	Prop	`BonePile`	Gives the Bone once
	Prop	`TwineBall`	Gives the Twine once
	Prop	`Jar`	Catch-the-bug mini-game; gives the Bug once
	Hotspot	`Ceiling`	Boss fight; win plays the ending, loss respawns at `Bed`
	Character	`Frog`	The landlord; `ChatWithFrog`; trades the spare Key for the BoneFlute

Other names: characters `Player` (the PC) and `Frog`; items `Key`, `Bone`, `Twine`, `BoneFlute`, `Bug`; dialog `ChatWithFrog`. Audio cues (Chapter 18, files in `res://game/audio/`): `mx_hallway`, `sfx_pickup`, `sfx_win`, `sfx_lose`. Optional objects from exercises (`Wall`, `FrogSpot`, `FrontDoor`, `Closet` and so on) are not part of the finished game in these listings.

End-to-end playthrough checklist

Run this on a **fresh game** (quit and start again; do not load a save). Tick each line. It covers every script above.

1. [ ] The game starts in the Bedroom, with the Player at its placed start position.
2. [ ] Click the Door: the first idle line, "It is locked." Click it again: "Still locked." Right-click it: "A locked door. It would like a key."
3. [ ] Right-click the Drawer: "A drawer that hums faintly." Click it: the Key is added and "A key! It is warm. Why is it warm?". Click again: "Just old socks now."
4. [ ] Right-click the Key in the inventory bar: "It is warm. Keys should not be warm."
5. [ ] Select the Key and click the Door: "It fits!" and the Key leaves the inventory. Click the Door: fade to the Hallway; the Player appears at `FromBedroom`; music starts.
6. [ ] The intro plays once: zoom, both face each other, three Frog lines, a shake, "Noted.", zoom back. (On a second fresh game, press Esc during it: it skips, the zoom ends normal.)
7. [ ] Click the BackDoor: back to the Bedroom at `FromHallway`. Click the Drawer: still "Just old socks now." Click the Door: straight to the Hallway; the intro does not replay.
8. [ ] Take the Bone from the BonePile and the Twine from the TwineBall: each prop disappears. Leave and come back: both are still gone, one Bone and one Twine in the inventory.
9. [ ] Use the Twine on the Bone: BoneFlute, both gone. (On another run, Bone on Twine does the same.) Bone on the Door: "That won't open it."
10. [ ] Click the Frog: first greeting, menu Who / Goodbye. Who: two lines; the menu now shows Flute / Goodbye. Choose Flute: the trade happens, the BoneFlute goes, a Key (the spare) arrives and the conversation ends. Click the Frog again: "The rent is paid in music. You paid."
11. [ ] Click the Jar and win the mini-game: "Got it!", the pickup sound, the Bug in the inventory. Click the Jar: "The jar is empty now." (Losing first is fine: click it again to retry.)
12. [ ] Use the Bug on the Frog: "A bug. Nice. But I want music, not snacks." The Bug stays in the inventory.
13. [ ] Press F9 in the Hallway ("Saved to slot 1"). Quit, run, press F10: you are in the Hallway with the same items, the piles gone, the music playing and no intro.
14. [ ] Click the Ceiling and lose: the rumble, a white cover, the fight, the lose sound, then the Bedroom at `Bed` with "Ow. Everything went white." All items are still there. Walk back to the Hallway: no intro.
15. [ ] Click the Ceiling and win: the win sound, "It stopped. I think I won?", the Frog's ending lines (including the Bug line) and the "THE END" box. Click the Ceiling again: "The ceiling is quiet now."
16. [ ] Use the spare Key on the Bedroom Door: "It is already unlocked. The door looks smug." The Key stays.
17. [ ] Click the locked Door five times on a new game, save, load, click again: the idle lines continue where they were.
18. [ ] Switch screen shake off in `user://settings.cfg` (or your options button), restart: the rumble and the boss-fight shake are gone.
19. [ ] Run the exported `.exe` and repeat steps 1, 5, 13 and 15 (Chapter 18, 18.7).

G.12 Corrections made to the chapters while merging

Merging the fragments showed these small differences, which were fixed in the chapters so that every chapter now agrees with this appendix:

- The inventory item formerly called `String` is now `Twine` in Chapters 12 to 18 (prop `TwineBall`, flag `twine_taken`, script `inventory_item_twine.gd`), because `I.String` would clash with GDScript's `String` type.
- `teleport_to_marker()` is awaited everywhere, as in Chapter 14 (Chapters 15, 16, 17 and 18 had dropped the `await`).
- The Frog's Bug line in Chapter 16 now uses the wording promised in Chapter 12: "A bug. Nice. But I want music, not snacks."

- Chapter 15 explains that the Chapter 14 Drawer forgets because its variable is untyped (typed prop variables are restored by Popochiu), and why `Globals` is still the right home for facts.
- Chapters 13 and 14 now show the generated lines as the dock actually writes them: `E.command_fallback()` (the dock shortens the template's `PopochiuUtils.e`) and `var state: Data = load("...tres")`.
- Chapter 18's Lab step 3 no longer expects the Key in the inventory after the Door has used it up.