GAME DEVELOPMENT

Godot Fundamentals

The Godot game engine and the game loop, GDScript basics, nodes and scenes, 2D fundamentals, input and player movement, physics and collision, signals and game events, UI and HUD, audio and polish, and a capstone building a complete small game.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: GODOT & THE GAME LOOP

Godot Fundamentals

Chapter 1 · Godot & the Game Loop

Godot is a free, open-source game engine, and the tool this entire course is built around. Before writing a single line of GDScript, this chapter covers the two ideas everything else depends on: how Godot organises a game as a tree of Nodes, and how the game loop actually runs, frame by frame.

What Is Godot?

Godot is released under the MIT licence — genuinely free, with no royalties, even for a commercial game you go on to sell. As of this writing the current stable release is Godot 4.7, and the engine ships as a single executable rather than a heavy installer.

Free & open source

MIT licensed. No subscription, no revenue share, no per-seat cost — for a hobby project or a commercial release alike.

GDScript

Godot's own built-in scripting language — deliberately close to Python's own syntax, covered fully in Chapter 2.

2D and 3D

One engine handles both. This course focuses on 2D, since it's the faster, more forgiving place to actually learn the fundamentals.

Installing Godot

1. Download the current stable build from `godotengine.org/download` for your platform.
2. Unlike many game engines, there's no installer to run — the download is a single executable file.
3. Run it directly. Godot opens straight into the Project Manager, where you create or open a project.

> **No install step, genuinely** — you can keep multiple Godot versions side by side as separate files with no conflict, which matters more than it sounds once you're following tutorials written against slightly different versions.

The Node & Scene Paradigm

Everything in Godot is a **Node**. A sprite is a node. A sound player is a node. A camera is a node. A collision shape is a node. Each node type is a small, focused component that handles exactly one job.

Nodes are arranged in a tree — a parent can have any number of children, but every node has exactly one parent (except the root). This tree of nodes is called a **scene**, and a scene is typically saved as its own file so it can be reused — a player character, an enemy, a whole level can each be their own scene.

> **Moving or deleting a parent affects everything beneath it** — exactly like a folder and the files inside it. This is a deliberate, useful property of the tree, not a gotcha, but it's worth having in mind from the very first scene you build.

A first scene, conceptually

A simple player character scene might look like this: a `Node2D` as the root (giving the whole thing a position, rotation, and scale), with a `Sprite2D` child (the visible image) and a `Camera2D` child (so the view follows the player). Three specialised nodes, each doing one job, combined into one reusable scene.

Scripts & the `extends` Keyword

A script attaches behaviour to a node. Every GDScript script starts by declaring which node type it `extends` — meaning the script's own code effectively becomes part of that node, with access to everything that node type already provides.

```
extends Node2D
```

```
# Called once when the node enters the scene tree for the first time.
func _ready() -> void:
    print("Hello, Godot!")
```

`_ready()` is one of Godot's own built-in lifecycle functions — it runs exactly once, the moment this node is fully set up in the scene tree. It's a natural place for one-time setup, but it isn't the game loop itself — that's covered next.

The Game Loop — `_process` vs. `_physics_process`

Godot calls two special functions repeatedly, for as long as the game runs, and the difference between them matters a great deal.

FUNCTION	RUNS	USE IT FOR
`_process(delta)`	Every rendered frame — rate can vary with performance	UI updates, animation, general logic, reading input for an immediate response
`_physics_process(delta)`	A fixed rate, independent of rendering framerate	Movement, collisions, anything using `move_and_slide()` — physics needs a consistent, predictable timestep

Both functions receive a `delta` parameter — a floating-point number representing how much time has passed since the last call, typically around `0.0167` seconds at 60 frames per second.

```
extends Node2D

var speed = 200.0

func _physics_process(delta):
    # Move 200 pixels per SECOND, not 200 pixels per FRAME.
    position.x += speed * delta
```

Always scale movement by `delta` — moving a fixed number of pixels per call, with no `delta` involved, would make the game run faster or slower depending purely on how fast the player's own

machine happens to render frames. Multiplying by `delta` is what makes movement frame-rate independent.

COMING FROM PYTHON

GDScript's own syntax was deliberately modeled on Python — indentation-based blocks, a similar `func` / `def` feel, and no semicolons. If you've worked through this site's own Python courses, GDScript will feel immediately familiar in a way most other scripting languages won't. The real differences show up in Chapter 2: GDScript is optionally statically typed, and it has Godot-specific building blocks — nodes, signals, scenes — that Python's own standard library has no equivalent for.

Coding Challenges

CHALLENGE 1

Install Godot and create a new project. Add a `Node2D` as the scene's root, save the scene, and run it (even with nothing visible yet — the goal is confirming your setup works end to end).

→ Solution

CHALLENGE 2

Build a small scene tree by hand: a `Node2D` root named "Player," with a `Sprite2D` child and a `Camera2D` child. Inspect the tree in Godot's own Scene panel and explain, in your own words, why each node is a separate child rather than one combined node.

→ Solution

CHALLENGE 3

Attach a script extending `Node2D` to your root node. Implement both `_process(delta)` and `_physics_process(delta)`, each printing its own `delta` value, and run the scene to compare how often each one actually fires.

→ Solution

QUICK REFERENCE — GODOT & THE GAME LOOP

- Godot: free, MIT licensed, single-executable install, current stable release 4.7
- Everything is a Node; a tree of Nodes is a scene
- A script `extends` a node type, becoming part of that node
- `_process(delta)`: every rendered frame, variable rate — UI, general logic
- `_physics_process(delta)`: fixed rate — movement, collisions
- Always scale movement by `delta` for frame-rate independence

CHAPTER 2: GDSCRIPT BASICS

Godot Fundamentals

Chapter 2 · GDScript Basics

Chapter 1 attached a first script with `extends` and touched `_ready()` without explaining GDScript itself. This chapter covers the language properly: variables and typing, the built-in types, functions, and control flow — the same territory a first Python chapter would cover, and for good reason, since GDScript's own syntax was deliberately modeled on Python's.

> **COMING FROM PYTHON**
>
> This whole chapter will feel like a dialect of something you already know rather than a new language. Blocks are still indentation-based, there are still no semicolons or curly braces, and `func` plays the exact role `def` does. The differences worth tracking as you go: GDScript adds **optional static typing** that Python doesn't have (without a separate tool), it has no `elif`-free equivalent quirks — it actually spells it `elif`, same as Python — and its `match` statement (covered below) predates Python's own `match` by several years, though the two look similar today.

Variables & Type Hints

A variable is declared with `var`. Like Python, GDScript doesn't require a type at all — but unlike Python, it lets you add one directly, and the editor will then catch a type mismatch before you ever run the game.

```
# No type hint - inferred at runtime, same as Python.
var health = 100
var player_name = "Sam"

# Explicit type hint - the editor enforces this.
var lives: int = 3
var speed: float = 150.0

# Inferred typing with := - GDScript figures out the type from the
# value on the right, and still enforces it afterward.
var is_alive := true
```

A `const` works the same way but can never be reassigned after it's set — used for values that are genuinely fixed, like a maximum speed or a gravity constant.

```
const MAX_SPEED = 300.0
const GRAVITY = 980.0
```

Type hints are optional, but worth the habit — a wrong-type mistake (assigning a String to a variable you meant to hold a number) is caught immediately in the editor instead of surfacing as a confusing runtime error mid-game. Later chapters use type hints throughout for exactly this reason.

Built-in Types

TYPE	EXAMPLE	NOTES
`int`	`var lives: int = 3`	Whole numbers
`float`	`var speed: float = 150.0`	Decimal numbers
`bool`	`var is_alive := true`	`true` / `false`, lowercase
`String`	`var name := "Sam"`	Text — capital S, unlike Python's lowercase `str`
`Array`	`var items = [1, 2, 3]`	Like a Python list; can optionally be typed, e.g. `Array[int]`
`Dictionary`	`var d = {"hp": 100, "mp": 50}`	Like a Python dict, key/value pairs
`Vector2`	`var pos = Vector2(10, 20)`	Godot-specific — an (x, y) pair used constantly for position/movement

COMING FROM PYTHON

`Array` and `Dictionary` map directly onto Python's `list` and `dict` — same square-bracket and curly-brace literal syntax, same mixed-type-by-default flexibility. `Vector2` has no direct Python equivalent in the standard library (it's closer to a NumPy array of length 2) — it's Godot's

> own building block for anything with an x/y position, and it shows up everywhere starting in Chapter 4.

Functions

Functions are declared with `func`, and — like variables — parameters and return values can optionally be typed.

```
# No type hints - works, but nothing is enforced.
func add(a, b):
    return a + b

# Typed parameters and a typed return value.
func take_damage(amount: int, current_hp: int) -> int:
    return current_hp - amount

# A default parameter value - just like Python's own default arguments.
func greet(name: String = "Player") -> void:
    print("Hello, " + name + "!")
```

A function with no meaningful return value is typed `-> void`, GDScript's explicit way of saying "this function doesn't return anything" — Python has no direct equivalent (a bare Python function implicitly returns `None`, with nothing written to say so).

Control Flow

if / elif / else

```
if health <= 0:
    print("Game over")
elif health < 30:
    print("Low health!")
else:
    print("Doing fine")
```

for loops

```
# Same shape as Python's for-in-range().
for i in range(5):
    print(i)

# Iterating directly over an array, same as Python.
for enemy_name in ["Slime", "Goblin", "Dragon"]:
    print(enemy_name)
```

while loops

```
var countdown = 3
while countdown != 0:
    print(countdown)
    countdown -= 1
```

match — GDScript's switch statement

Python only gained `match` in version 3.10; GDScript has had one from early on. The shape is familiar if you've used Python's version: match a value against several patterns, with `_` as the catch-all default.

```
match weapon_type:
    0:
        print("Sword")
    1:
        print("Bow")
    _:
        print("Unknown weapon")
```

> **Indentation is significant, exactly like Python** — a stray tab/space mismatch produces a real parse error. Godot's own script editor defaults to tabs for GDScript files; mixing tabs and spaces in the same file is the single most common source of a confusing "unexpected indent" error for anyone new to the language.

Coding Challenges

CHALLENGE 1

Write a typed function `calculate_damage(base: int, multiplier: float) -> int` that returns the base damage multiplied by the multiplier, rounded down to a whole number. Call it with a few different values and print the results.

→ Solution

CHALLENGE 2

Build an `Array` of five enemy names as Strings. Using a `for` loop, print each name along with its position in the array (e.g. "1: Slime").

→ Solution

CHALLENGE 3

Write a script that starts a player at 100 health and, using a `while` loop, subtracts 15 health per iteration. Inside the loop, use `if` / `elif` / `else` to print "Critical!", "Hurt", or "Healthy" depending on the current health, and stop the loop once health reaches 0 or below.

→ Solution

QUICK REFERENCE — GDSCRIPT BASICS

- `var name: Type = value` — typed; `var name := value` — inferred; `var name = value` — untyped
- `const NAME = value` — never reassigned
- Core types: `int`, `float`, `bool`, `String`, `Array`, `Dictionary`, `Vector2`
- `func name(param: Type = default) -> ReturnType:`
- `if` / `elif` / `else`, `for x in range(n)`, `for x in array`, `while`, `match`

- Indentation-based blocks, no semicolons — same as Python

CHAPTER 3: NODES & SCENES IN DEPTH

Godot Fundamentals

Chapter 3 · Nodes & Scenes In Depth

Chapter 1 introduced the scene tree conceptually; Chapter 2 covered GDScript itself without touching nodes at all. This chapter puts both together: how to actually navigate a scene tree from code, how to create new nodes at runtime by instancing a whole scene, and a first look at **signals** — Godot's own way for one node to tell another that something happened, without the two needing to know much about each other.

Navigating the Scene Tree

Once a script is attached to a node, it can reach other nodes in the same tree in a few ways.

METHOD	GETS
`get_parent()`	This node's own parent
`get_children()`	An Array of every direct child
`get_node("Path")`	A specific descendant, by relative path
`$Path`	Shorthand for `get_node("Path")` — used constantly in real code

```
extends Node2D

func _ready() -> void:
    # $ is shorthand for get_node() - both reach the same child.
    var sprite = $Sprite2D
    var same_sprite = get_node("Sprite2D")

    # A deeper path works the same way a file path does.
    var nested = $UI/HealthBar

    print(get_parent().name)
    print(get_children())
```

Prefer `$Path` for anything set up in the editor — it's shorter, and Godot's own editor autocompletes real node names as you type it, catching a typo immediately rather than failing silently at runtime.

Instancing Scenes at Runtime

Every scene saved to a `.tscn` file can be loaded and created fresh from a script — this is how a game spawns things that don't exist yet when the game starts: a bullet, an enemy, a pickup.

```
extends Node2D

# preload() loads the scene once, at compile time - the fastest option
# when the path is known in advance and never changes.
const BulletScene: PackedScene = preload("res://bullet.tscn")

func shoot() -> void:
    var bullet = BulletScene.instantiate()
    add_child(bullet)
    bullet.position = position
```

CALL	WHEN IT RUNS	USE IT WHEN...
`preload("res://x.tscn")`	At compile time, once	The path is a fixed, known string
`load("res://x.tscn")`	At runtime, each call	The path is built dynamically (e.g. from a variable)
`.instantiate()`	Whenever called	Creates one real, independent copy of that scene as a node

Godot 3 vs. Godot 4 naming — a lot of tutorials still online use `.instance()`. That was the Godot 3 method name; Godot 4 renamed it to `.instantiate()`. If a tutorial's code throws an "Invalid call" error on that exact line, this rename is very often why.

An instanced node exists in memory but isn't part of the visible scene tree until `add_child()` actually attaches it — exactly like Chapter 2's typed `Array`: creating the object and placing it somewhere are two separate steps.

Removing a node

The reverse operation is `queue_free()` — it schedules the node for deletion at the end of the current frame, rather than deleting it immediately mid-execution (which can cause errors if other code is still using it that same frame).

```
bullet.queue_free()
```

A First Look at Signals

A signal is Godot's own way for a node to announce "something happened" without needing to know who, if anyone, is listening. Any node can declare a custom signal, **emit** it when the relevant thing occurs, and any other node can **connect** a function to run whenever that signal fires.

```
# --- enemy.gd (attached to the Enemy scene) ---
extends Node2D

signal died(enemy_name: String)

func take_damage(amount: int) -> void:
    health -= amount
    if health <= 0:
        died.emit(name)
        queue_free()
```

```
# --- game.gd (a parent node that spawned the enemy) ---
extends Node2D

func _ready() -> void:
    $Enemy.died.connect(_on_enemy_died)
```

```
func _on_enemy_died(enemy_name: String) -> void:
    print(enemy_name + " was defeated!")
```

Godot's built-in nodes already come with their own signals too — a `Button` has a `pressed` signal, an `Area2D` has a `body_entered` signal, and so on. Custom signals declared with `signal` work exactly the same way, connected and emitted with the identical syntax.

COMING FROM PYTHON

Signals are Godot's own built-in version of the **observer pattern** — if you've written a callback registered against an event (a button's `on_click` handler, a custom pub/sub system), the shape is the same idea: something emits an event, and zero or more listeners react to it, with neither side needing a direct reference to the other's internals. What's different is that this pattern is a first-class part of the language itself here, declared with `signal` and connected with `.connect()`, rather than something you'd build yourself with a list of callback functions the way you might in plain Python.

Why bother with signals instead of just calling a function directly? A signal keeps the Enemy scene fully independent of whatever happens to be listening — it doesn't need to know the parent scene exists at all. That decoupling is what lets the exact same Enemy scene be reused in a dozen different levels without editing its own script every time. Chapter 7 goes much deeper on this.

Coding Challenges

CHALLENGE 1

Build a scene with a Node2D root and three Node2D children named "Head," "Body," and "Feet." Write a script on the root that uses `get_children()` to print the name of every child, and separately uses `$Body` to print that one child's name directly.

→ Solution

CHALLENGE 2

Save a simple scene (e.g. a single Node2D) as its own `.tscn` file. From a different script, `preload` it, instantiate three copies with `.instantiate()`, and `add_child()` each one at a different

position so all three appear in the running scene.

→ Solution

CHALLENGE 3

Declare a custom signal called `coin_collected(amount: int)` on one node. Emit it with a sample amount, and connect a function on a different node that prints a message using the amount it received.

→ Solution

QUICK REFERENCE — NODES & SCENES IN DEPTH

- `get_parent()`, `get_children()`, `get_node("Path")` / `$Path`
- `preload("res://x.tscn")` (fixed path) vs. `load(...)` (dynamic path)
- `.instantiate()` creates a node from a PackedScene — `.instance()` was the old Godot 3 name
- `add_child(node)` attaches it to the tree; `queue_free()` removes it safely at end of frame
- `signal name(params)` declares a signal; `signal_name.emit(values)` fires it; `node.signal_name.connect(func)` listens for it

CHAPTER 4: 2D FUNDAMENTALS

Godot Fundamentals

Chapter 4 · 2D Fundamentals

Every chapter so far has worked with an empty Node2D. This chapter makes something actually appear on screen: displaying an image with `Sprite2D`, understanding exactly how Godot measures position in 2D space, and moving something across the screen using nothing more than what's already been covered.

Godot's 2D Coordinate System

Godot's 2D origin, `(0, 0)`, sits at the **top-left** of the viewport. X increases to the right, as you'd expect — but Y increases **downward**, not upward.

> **Y grows downward — the single most common early surprise** — this is the opposite of a standard math graph (and of a NumPy/matplotlib plot), where Y usually increases upward. It matches how most 2D game engines and image formats work, since screen pixels are naturally numbered from the top-left corner down. A positive `y` value moves something *down* the screen, not up.

position.x

Increasing it moves an object right; decreasing it moves left.

position.y

Increasing it moves an object **down**; decreasing it moves up.

Vector2

An (x, y) pair. `position` on every Node2D is itself a Vector2.

Position, Rotation & Scale

Every `Node2D` (and everything that extends it, including `Sprite2D`) automatically has three built-in properties: `position`, `rotation`, and `scale`. These were mentioned briefly in Chapter 1 — here's what each one actually does.

```
extends Node2D

func _ready() -> void:
    position = Vector2(100, 50)  # 100px right, 50px down from origin
    rotation = PI / 4            # rotation is in RADIANS, not degrees
    scale = Vector2(2.0, 2.0)      # twice the original size
```

> `rotation` **is in radians** — if degrees are more natural to think in, use `rotation_degrees` instead; it's the same underlying value, just exposed in a different unit.

Sprite2D — Displaying an Image

`Sprite2D` is the node responsible for showing a 2D image (a "texture," in Godot's own terminology). Add one as a child, assign it an image via its `texture` property, and it renders at its parent's position.

```
extends Node2D

func _ready() -> void:
    $Sprite2D.texture = preload("res://player.png")
```

More commonly, a texture is assigned directly in the editor's Inspector panel rather than from a script — drag an image file onto the `Sprite2D`'s own `texture` slot, and it's ready to go with no code at all.

PROPERTY	DOES
`texture`	The image to display
`centered`	If true (the default), the image is centered on the node's own position; if false, its top-left corner sits at that position instead
`flip_h` / `flip_v`	Mirror the image horizontally or vertically — useful for a character facing left vs. right

Basic On-Screen Movement

Chapter 1 already showed the pattern for frame-rate-independent movement using `_physics_process` and `delta`. Applied here to an actual visible sprite, the same idea makes something glide across the screen.

```
extends Sprite2D

const SPEED = 150.0

func _physics_process(delta: float) -> void:
    # Moves steadily to the right, 150 pixels per second.
    position.x += SPEED * delta
```

Note this script `extends Sprite2D` directly, rather than a separate Node2D parent — since `Sprite2D` already inherits `position` from `Node2D`, a script can be attached straight to the sprite itself.

This isn't "real" movement yet — directly changing `position` works for a simple demo, but it has no awareness of walls, floors, or other objects; it will happily move straight through anything. Chapter 5 adds real input control, and Chapter 6 introduces `CharacterBody2D` and collision — the node types built specifically for movement that actually respects the rest of the game world.

COMING FROM PYTHON

`Vector2(x, y)` behaves a lot like a lightweight NumPy array of length 2 — you can add, subtract, and scale two Vector2 values directly with `+`, `-`, and `*`, the same way NumPy overloads those operators for arrays, rather than needing separate `.x` / `.y` arithmetic every time. The Y-axis direction is the real trap to remember: a `matplotlib` plot or a plain Cartesian mental model has Y increasing upward; Godot's 2D space does not.

Coding Challenges

CHALLENGE 1

Add a Sprite2D to a scene with any image assigned to its texture. Write a script that sets its `position` to `Vector2(200, 300)` and its `scale` to `Vector2(1.5, 1.5)` in `_ready()`, and print the resulting position and scale.

→ Solution

CHALLENGE 2

Write a script attached to a Sprite2D that continuously moves it downward at 80 pixels per second using `_physics_process`. Once its `position.y` passes 400, print "Reached the bottom" exactly once (not on every frame after).

→ Solution

CHALLENGE 3

Write a script that continuously increases a Sprite2D's `rotation` in `_physics_process`, making it spin steadily in place. Use `rotation_degrees` instead of `rotation` and confirm the sprite still spins identically.

→ Solution

QUICK REFERENCE — 2D FUNDAMENTALS

- Origin (0, 0) is top-left; X increases right; **Y increases down**
- `position`, `rotation` (radians; `rotation_degrees` for degrees), `scale` — all on Node2D and everything that extends it
- `Sprite2D.texture` — the image to display; `centered`, `flip_h` / `flip_v`
- Direct movement: `position.x += SPEED * delta` inside `_physics_process`
- Directly setting `position` ignores walls/collisions — real movement comes in Chapters 5-6

CHAPTER 5: INPUT & PLAYER MOVEMENT

Godot Fundamentals

Chapter 5 · Input & Player Movement

Everything so far has moved on its own, automatically, in `_physics_process`. This chapter connects that movement to the keyboard for the first time — genuinely the first moment in this course where something responds to you pressing a key. Building a real, player-controlled character is the payoff for every chapter that came before it.

The Input Map

Godot doesn't ask you to check for a raw key like "the right arrow" directly in most real code. Instead, you define a named **action** — like `"move_right"` — and map one or more physical keys/buttons to it. Your script then only ever asks about the action, never the specific key.

1. Open **Project > Project Settings > Input Map**.
2. Type a new action name (e.g. `move_right`) and click "Add."
3. Click the "+" next to your new action and press the key or button you want mapped to it (e.g. the right arrow, or `D`).
4. Repeat for `move_left`, `move_up`, and `move_down`.

> **Why name actions instead of checking raw keys?** The same decoupling idea from Chapter 3's signals shows up again here — a script that checks `Input.is_action_pressed("move_right")` doesn't care whether that's currently bound to the right arrow, `D`, or a gamepad's right stick. Rebinding controls, or adding gamepad support later, needs zero script changes — only the Input Map itself changes.

Godot also ships with several built-in actions already defined, prefixed `ui_` — like `ui_left`, `ui_accept` — meant mainly for menu navigation. Gameplay input generally gets its own custom action names instead, exactly as above.

Reading Input

METHOD	RETURNS	USE IT FOR
`Input.is_action_pressed("name")`	`bool`	A single held button — true for as long as it's down
`Input.is_action_just_pressed("name")`	`bool`	True for exactly one frame, the moment the button goes down — a jump or a menu confirm, not continuous movement
`Input.get_axis("neg", "pos")`	`float`, -1 to 1	One axis of movement — e.g. left/right only
`Input.get_vector("l", "r", "u", "d")`	`Vector2`	Full 2D movement — the usual choice for a top-down or platformer controller

```
func _physics_process(delta: float) -> void:
    if Input.is_action_pressed("jump"):
        print("Jumping!")

    var horizontal := Input.get_axis("move_left", "move_right")
    print(horizontal)  # -1.0, 0.0, or 1.0
```

> `get_vector()` **normalizes diagonal movement automatically** — pressing both "right" and "down" at once returns a Vector2 with a length of at most 1, not one with a length of `sqrt(2)`. Without this, a character moving diagonally would move noticeably faster than one moving in a straight line — a classic, easy-to-miss movement bug that `get_vector()` avoids automatically.

Building a Real Player Controller

Putting the pieces together: read input every physics frame, turn it into a direction, and apply it to `position` scaled by speed and `delta` — exactly the movement pattern from Chapter 4, now driven by the keyboard instead of running on its own.

```
extends Sprite2D
```

```
const SPEED = 200.0

func _physics_process(delta: float) -> void:
    var direction := Input.get_vector("move_left", "move_right", "move_up", "move_down")
    position += direction * SPEED * delta
```

Run this attached to a visible sprite, with the four `move_*` actions bound to arrow keys or

thanks to `get_vector()`'s own automatic normalization.

Still not "real" movement — this script moves the sprite straight through walls and other objects, exactly like Chapter 4's own automatic movement. Chapter 6 replaces this pattern with `CharacterBody2D` and `move_and_slide()`, which respects collisions — but everything about reading input stays exactly the same from here on.

COMING FROM PYTHON

`Input.get_vector()` doing its own normalization under the hood is the kind of thing you'd otherwise write by hand in Python with something like `direction / direction.length()` (careful to guard against dividing by zero when nothing is pressed) — Godot bakes that safety directly into the built-in function, including handling the "nothing is pressed" case cleanly by returning a zero vector rather than raising an error.

Coding Challenges

CHALLENGE 1

In the Input Map, create a `"jump"` action bound to the spacebar. Write a script that prints "Jump!" the moment the action is pressed — but only once per press, not repeatedly while held.

→ Solution

CHALLENGE 2

Create `move_left` / `move_right` actions and write a script using `Input.get_axis()` that moves a Sprite2D horizontally at 250 pixels per second in whichever direction is held, with no movement

when neither is pressed.

→ Solution

CHALLENGE 3

Build the full four-directional player controller from this chapter. Extend it so that whenever `direction.x` is negative, the sprite's `flip_h` (from Chapter 4) is set to true, and false whenever it's positive — so the character visually faces the direction it's moving.

→ Solution

QUICK REFERENCE — INPUT & PLAYER MOVEMENT

- Project > Project Settings > Input Map — define named actions, bind keys/buttons to them
- Scripts check the named action, never a raw key — decoupled, remappable
- `Input.is_action_pressed("name")` — bool, held state
- `Input.get_axis("neg", "pos")` — float, -1 to 1
- `Input.get_vector("l", "r", "u", "d")` — Vector2, normalized diagonal movement
- `position += direction * SPEED * delta` in `_physics_process`

CHAPTER 6: PHYSICS & COLLISION

Godot Fundamentals

Chapter 6 · Physics & Collision

Every movement script so far has moved straight through walls, floors, and anything else in its path — both Chapter 4 and Chapter 5 flagged this openly as unfinished business. This chapter replaces raw `position` changes with Godot's own real physics nodes, which know about the rest of the game world and actually respond to it.

Three Physics Node Types

Godot splits "something that participates in physics" into three different node types, each suited to a different job.

NODE	WHO'S IN CONTROL	TYPICAL USE
CharacterBody2D	Your script, fully — you set velocity and call a move function every frame	A player character, an enemy with hand-scripted movement
RigidBody2D	Godot's own physics engine — gravity, forces, and collisions are simulated automatically	A ball, a crate, debris — anything that should behave like a real physical object
Area2D	Neither — it detects overlap but applies no physical collision response at all	A pickup, a damage zone, a trigger that opens a door

CollisionShape2D — Required for All Three

None of the three node types above detect anything on their own — each needs a `CollisionShape2D` child, with an actual **Shape** resource assigned to its own `shape` property (a `RectangleShape2D`, a `CircleShape2D`, etc.).

> **An empty CollisionShape2D detects nothing** — adding the node alone isn't enough. In the Inspector, its `Shape` property must be set to a real shape resource, or the whole node is invisible to physics regardless of how it's positioned.

CharacterBody2D — Scripted Movement That Respects the World

A `CharacterBody2D` has a built-in `velocity` property (a Vector2), and a built-in `move_and_slide()` method that actually moves the node by that velocity, sliding smoothly along any wall or floor it collides with instead of passing through it.

```
extends CharacterBody2D

const SPEED = 200.0

func _physics_process(delta: float) -> void:
    var direction := Input.get_vector("move_left", "move_right", "move_up", "move_down")
    velocity = direction * SPEED
    move_and_slide()
```

> **Godot 3 vs. Godot 4 — `move_and_slide()` takes no arguments now** — in Godot 3, it took a velocity parameter and returned the resulting velocity, which you had to store back yourself. Godot 4 simplified this: set the node's own built-in `velocity` property first, then call `move_and_slide()` with no arguments at all — it reads and updates `velocity` automatically. A lot of tutorials still online use the old Godot 3 call shape; if code from one of those throws an "too many arguments" error, this is almost always why.

For a platformer, gravity is applied by hand each frame, and `is_on_floor()` reports whether the body is currently resting on the ground:

```
extends CharacterBody2D

const GRAVITY = 900.0
const JUMP_VELOCITY = -400.0

func _physics_process(delta: float) -> void:
    if not is_on_floor():
        velocity.y += GRAVITY * delta

    if Input.is_action_just_pressed("jump") and is_on_floor():
```

```
        velocity.y = JUMP_VELOCITY

    move_and_slide()
```

Area2D — Detecting Overlap Without Physical Collision

Unlike `CharacterBody2D` or `RigidBody2D`, an `Area2D` doesn't stop anything from passing through it — it simply notices when something does, and emits a signal.

```
# --- coin.gd, attached to an Area2D ---
extends Area2D

func _ready() -> void:
    body_entered.connect(_on_body_entered)

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        print("Coin collected!")
        queue_free()
```

The connection pattern here is exactly Chapter 3's own custom-signal approach — `body_entered` is simply a signal Godot's built-in `Area2D` already declares for you, connected and used the same way as the custom `coin_collected` signal from that chapter.

> **Groups, briefly** — `is_in_group("player")` checks whether a node was tagged into a named group (set in the editor's Node panel, under "Groups"). It's a lightweight way to ask "is this specifically the player?" without the Area2D needing a direct reference to the player scene at all — another small instance of the same decoupling idea from earlier chapters.

> **If `body_entered` never fires** — the most common cause is a **collision layer/mask mismatch**: the Area2D's own `collision_mask` has to include whatever layer the other body is actually on. Both nodes also need a real Shape assigned (see above), and the Area2D's own `monitoring` property has to be left enabled.

RigidBody2D — Letting the Engine Take Over

A `RigidBody2D` is deliberately not driven by setting `position` or `velocity` directly each frame — Godot's own physics engine already applies gravity and resolves collisions on its own. Instead, you nudge it with forces or impulses.

```
extends RigidBody2D

func launch() -> void:
    # A one-time push, like a kick or an explosion.
    apply_central_impulse(Vector2(0, -400))
```

This course focuses mainly on `CharacterBody2D`, since a hand-controlled player is this course's own central goal — `RigidBody2D` is worth knowing exists for anything that should behave like a genuinely physical object rather than a directly-controlled character.

COMING FROM PYTHON

The CharacterBody2D-vs-RigidBody2D split maps reasonably well onto a distinction you may already know from game-adjacent Python work: writing your own explicit update loop (you decide exactly what happens to an object's position each tick — CharacterBody2D) versus handing an object to a physics simulation library and letting it compute the result for you (RigidBody2D). Area2D has no close Python parallel by itself — it's closest to a pure event/collision-detector with no physical behavior attached at all.

Coding Challenges

CHALLENGE 1

Build a CharacterBody2D with a CollisionShape2D (a real shape assigned), plus a couple of static wall nodes with their own collision shapes nearby. Attach the four-directional `move_and_slide()` controller from this chapter and confirm the character stops at the walls instead of passing through.

→ Solution

CHALLENGE 2

Add gravity and jumping to a CharacterBody2D on a static floor, using `is_on_floor()` to only allow jumping while grounded. Confirm the character can't "double jump" by holding the jump key in mid-air.

→ Solution

CHALLENGE 3

Build an Area2D "coin" with a CollisionShape2D. Add a CharacterBody2D player tagged into a `"player"` group. Connect the coin's `body_entered` signal so that walking into it prints "Coin collected!" and removes the coin from the scene.

→ Solution

QUICK REFERENCE — PHYSICS & COLLISION

- CharacterBody2D — you control velocity and call `move_and_slide()`; respects walls/floors
- RigidBody2D — the engine simulates it; nudge with `apply_central_impulse()`/forces
- Area2D — detects overlap only, no physical collision response; used for triggers/pickups
- Every one of the three needs a CollisionShape2D child with a real Shape assigned
- Godot 4: `move_and_slide()` takes no arguments — set `velocity` first, it updates automatically
- `is_on_floor()` — true only while a CharacterBody2D is resting on the ground
- Area2D's `body_entered(body)` signal — connect it exactly like a custom signal (Chapter 3)

CHAPTER 7: SIGNALS & GAME EVENTS

Godot Fundamentals

Chapter 7 · Signals & Game Events

Chapter 3 declared a first custom signal; Chapter 6 connected a built-in one, `body_entered`. Both times, the connection was made entirely from code. This chapter goes deeper: connecting signals visually in the editor, controlling exactly how a connection behaves, pausing a function until a signal fires, and — the real point of all of it — using signals to design game systems that don't depend tightly on each other.

Connecting Signals in the Editor

Every connection shown so far has used `.connect()` in a script. Godot's editor also offers a fully visual alternative:

1. Select a node in the Scene panel.
2. Open the **Node** dock (usually tabbed next to the Inspector) and click "Signals."
3. Every signal that node offers — built-in and custom — is listed here.
4. Double-click a signal, choose the node that should handle it, and Godot generates a matching handler function automatically, already connected.

> **Editor vs. code — which to use?** Both create the exact same underlying connection. The editor is convenient for a connection that's fixed and known in advance (a specific button's own `pressed` signal, wired once at design time). Code-based connections are the better choice when the connection needs to happen dynamically — for instance, connecting to a signal on a node that was only just instanced at runtime, the way Chapter 3's spawned scenes were.

One-Shot Connections & Disconnecting

By default a connection stays active and fires every time the signal is emitted, for as long as both nodes exist. Two situations need something different: a connection that should only ever fire once, and a connection that needs to be torn down deliberately, before either node is freed.

```
# Fires exactly once, then Godot removes the connection automatically.
enemy.died.connect(_on_first_kill, CONNECT_ONE_SHOT)

# Manually tearing down a connection made earlier.
enemy.died.disconnect(_on_first_kill)
```

A leftover connection can crash your game — if node A connects to a signal on node B, and B is later freed with `queue_free()` while A still expects to receive that signal, referencing a freed node from the handler function raises a real error. Disconnecting explicitly (or using `CONNECT_ONE_SHOT` for anything genuinely one-time) avoids this class of bug.

`await` — Pausing Until a Signal Fires

GDScript's `await` keyword pauses a function's own execution until a given signal actually fires, then continues from exactly that point — useful for sequencing things like a short delay, an animation finishing, or a cutscene waiting for input.

```
func show_damage_popup() -> void:
    print("Ouch!")

    # Pause here for exactly 1 second, without freezing the rest of the game.
    await get_tree().create_timer(1.0).timeout

    print("Popup faded")

func wait_for_enemy_death() -> void:
    await enemy.died
    print("The enemy is gone - continuing")
```

`await` **doesn't freeze the game** — unlike a blocking `sleep()` call in plain Python, everything else in the game keeps running normally while one function is paused on `await`. Only that specific function's own execution is suspended, waiting for its signal.

Designing With Signals: A Decoupled Health System

The real payoff of signals shows up once several systems need to react to the same event without knowing about each other. A health system is the classic example: it shouldn't need to know a health bar, a screen shake effect, or a "low health" warning sound all exist — it should just announce that health changed, and let anything interested react on its own.

```
# --- health.gd, attached to the Player ---
extends Node

signal health_changed(current: int, max_health: int)
signal died

var current: int = 100
var max_health: int = 100

func take_damage(amount: int) -> void:
    current = max(current - amount, 0)
    health_changed.emit(current, max_health)

    if current == 0:
        died.emit()
```

A completely separate HUD scene, and a completely separate audio-warning script, can each independently connect to the exact same `health_changed` signal — neither needs to know the other exists, and neither needs any change if a third listener is added later. Chapter 8 builds a real HUD that does exactly this.

> **COMING FROM PYTHON**
>
> This "one emitter, many independent listeners" shape is the same reasoning behind Python's own `observer` patterns, a pub/sub library, or Django's own signal framework, if you've encountered it — the emitter never imports or references any of its listeners. What's distinctive in Godot is how lightweight declaring a new one is: a single `signal` line, no separate event-bus class to set up, no manual list of callback functions to maintain by hand.

Coding Challenges

CHALLENGE 1

Add a Button node to a scene. Using the editor's Node dock (not code), connect its `pressed` signal to a new handler function that prints "Button clicked!". Confirm it works by running the scene.

→ Solution

CHALLENGE 2

Write a function that prints "Get ready...", uses `await` with `get_tree().create_timer()` to pause for 2 seconds, then prints "Go!". Confirm the rest of the scene (e.g. a spinning sprite from Chapter 4) keeps moving during that 2-second pause rather than freezing.

→ Solution

CHALLENGE 3

Build the `health.gd` script from this chapter's own example. From two separate, unrelated nodes, each independently connect to its `health_changed` signal — one printing "HUD: health is now X/Y", the other printing "Audio: low health warning!" only when health drops to 30 or below. Call `take_damage()` a few times and confirm both listeners react correctly on their own.

→ Solution

QUICK REFERENCE — SIGNALS & GAME EVENTS

- Node dock > Signals tab — connect visually in the editor; generates a handler automatically
- `signal_name.connect(handler, CONNECT_ONE_SHOT)` — fires once, then auto-disconnects
- `signal_name.disconnect(handler)` — tears down a connection manually
- `await signal_name` / `await get_tree().create_timer(secs).timeout` — pauses one function without freezing the game

- Design systems to emit signals about what happened; let listeners react independently, with no direct reference back to the emitter

CHAPTER 8: UI & HUD

Godot Fundamentals

Chapter 8 · UI & HUD

Chapter 7 built a health system that emits signals but has no visible interface at all. This chapter builds the missing half: a real, on-screen HUD — a health bar and a score display — that listens to those signals and updates itself, plus a simple pause menu that shows and hides on demand.

CanvasLayer — Drawing UI on Top of the Game

Every node used so far (Sprite2D, CharacterBody2D, Area2D) lives in the game world — it moves with the camera, and can be positioned anywhere in that world's own coordinate space. UI is different: a health bar should always sit in the same spot on screen, regardless of where the camera happens to be looking.

`CanvasLayer` solves this — anything placed under it draws in a separate layer, on top of the game world, using screen coordinates instead of world coordinates.

> **The standard shape** — a scene's UI almost always lives under one `CanvasLayer` node, itself holding every HUD element as children. This keeps a clean separation: gameplay nodes below the CanvasLayer, interface nodes inside it.

Control Nodes & Anchors

UI elements — labels, buttons, bars — all inherit from `Control`, a node type built specifically for screen-space layout. A Control node uses **anchors** to describe where it sits relative to its parent's own edges, so it stays correctly placed even if the window is resized.

Anchor presets

The editor's Layout menu offers one-click presets — Top Left, Top Right, Full Rect, Center, and more — for the most common placements.

Containers

`VBoxContainer`, `HBoxContainer`, and `MarginContainer` automatically arrange their

	own children, instead of positioning each one by hand.

> **Prefer a preset over manual dragging** — a health bar anchored to "Top Left" stays in the top left corner at any window size; one positioned only by dragging it to a pixel coordinate will drift out of place the moment the window is resized.

Label & ProgressBar

NODE	KEY PROPERTY	USE
Label	`text`	Any plain on-screen text — a score, a name, a message
ProgressBar	`value` (with `min_value` / `max_value`)	A health bar, a stamina bar, a loading indicator

```
# --- score_label.gd, attached to a Label ---
extends Label

func update_score(new_score: int) -> void:
    text = "Score: " + str(new_score)
```

Wiring the HUD to Chapter 7's Health Signal

This is where UI and signals meet directly — a `ProgressBar`-based health bar connects to the exact same `health_changed` signal from Chapter 7's example, with no changes needed to the health system itself.

```
# --- health_bar.gd, attached to a ProgressBar under a CanvasLayer ---
extends ProgressBar

func _ready() -> void:
```

```
    var health = get_node("/root/Main/Player/Health")
    health.health_changed.connect(_on_health_changed)

    # Set up the bar's own range to match the health system.
    min_value = 0
    max_value = health.max_health
    value = health.current

func _on_health_changed(current: int, max_health: int) -> void:
    value = current
```

Hardcoded paths like `/root/Main/Player/Health` are fragile — they break the moment a scene is reorganized. This is acceptable for a small, single-scene example like this course, but a larger project typically reaches for either an exported node reference (dragged into the Inspector) or an **autoload/singleton** — a globally-accessible script Godot keeps alive across every scene. Autoloads are a genuinely deep topic on their own and are outside this course's own Fundamentals scope, but worth knowing the name of for when a project outgrows a hardcoded path like this one.

Showing & Hiding Menus

Every `Control` (and every `Node2D`-derived node, in fact) has a `visible` property, along with `show()` and `hide()` convenience methods that set it for you.

```
# --- pause_menu.gd, attached to a Control that's hidden by default ---
extends Control

func _ready() -> void:
    hide()  # start hidden, same as setting visible = false

func _process(delta: float) -> void:
    if Input.is_action_just_pressed("pause"):
        visible = not visible
```

COMING FROM PYTHON

`visible = not visible` flips a boolean the same way it would in Python — nothing new syntactically. What's Godot-specific is what a hidden Control actually does: setting `visible =`

`false` stops it from being drawn or processing mouse/keyboard input from that Control at all, without needing a separate "is this menu open" flag threaded through the rest of your code the way you might manage visibility state by hand in a plain Python UI toolkit.

Coding Challenges

CHALLENGE 1

Build a CanvasLayer containing a Label anchored to the top-left corner. Write a script with a `score` variable starting at 0, a function that adds 10 to it and updates the Label's text to "Score: X", and call that function a few times from `_ready()` to confirm it updates correctly.

→ Solution

CHALLENGE 2

Build the full Chapter 7 `health.gd` health system plus a ProgressBar HUD element that connects to its `health_changed` signal, matching this chapter's own `health_bar.gd` example. Call `take_damage()` a few times and confirm the bar's value visibly drops each time.

→ Solution

CHALLENGE 3

Build a Control-based pause menu that starts hidden. Bind a `"pause"` action to the Escape key in the Input Map, and toggle the menu's visibility each time it's pressed using `visible = not visible`.

→ Solution

QUICK REFERENCE — UI & HUD

- CanvasLayer — draws UI in screen space, on top of the game world

- Control nodes use anchors (Layout menu presets) to stay correctly placed at any window size
- Containers (VBoxContainer, HBoxContainer, MarginContainer) auto-arrange their children
- Label. `text` — plain text display
- ProgressBar. `value` / `min_value` / `max_value` — a health/stamina bar
- A HUD element connects to a game system's own signal (Chapter 7) — no changes needed to that system
- `visible`, `show()`, `hide()` — toggle any Control's own visibility

CHAPTER 9: AUDIO & POLISH

Godot Fundamentals

Chapter 9 · Audio & Polish

Everything built so far works correctly, but feels a little flat — a coin vanishes silently, a jump makes no sound, damage has no visual punch. Game developers call this missing quality **"juice"**: small audio and visual touches, layered on top of already-working systems, that make a game feel alive and responsive rather than merely functional. This chapter adds sound, a first animation, and a simple particle effect — the last pieces before Chapter 10's capstone.

Playing Sound: AudioStreamPlayer vs. AudioStreamPlayer2D

NODE	SOUND HAS A POSITION?	TYPICAL USE
AudioStreamPlayer	No — plays the same everywhere	Background music, UI clicks, a HUD notification
AudioStreamPlayer2D	Yes — quieter the farther the camera is	A coin pickup, footsteps, an explosion in the game world

```
# --- attached to an AudioStreamPlayer2D on the Coin from Chapter 6 ---
extends Area2D

func _ready() -> void:
    body_entered.connect(_on_body_entered)

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        $CollisionShape2D.set_deferred("disabled", true)
        $Sprite2D.hide()
        $AudioStreamPlayer2D.play()
        await $AudioStreamPlayer2D.finished
        queue_free()
```

> **Don't `queue_free()` before the sound finishes** — freeing the Coin node immediately (as Chapter 6's own version did) destroys the AudioStreamPlayer2D along with it, cutting the sound off instantly. Hiding the sprite and disabling the collision shape makes the coin disappear and stop being collectible right away, while `await $AudioStreamPlayer2D.finished` — the same `await`-a-signal pattern from Chapter 7 — delays the actual `queue_free()` until the sound has genuinely finished playing.

Background music is simpler: an `AudioStreamPlayer` with its `stream` set to a music file, `autoplay` enabled, and the stream's own "Loop" setting turned on in the editor's Import panel.

A First AnimationPlayer Animation

`AnimationPlayer` records how a node's own properties change over time — position, scale, modulate (color/transparency), and more — as a reusable, named animation.

1. Add an `AnimationPlayer` node, and open the Animation panel at the bottom of the editor.
2. Create a new animation and give it a name, e.g. `"flash"`.
3. With the timeline open, select the node to animate, change one of its properties (e.g. Sprite2D's `modulate` color) at a couple of different points in time — Godot records each change as a keyframe.

```
func take_damage(amount: int) -> void:
    current -= amount
    $AnimationPlayer.play("flash")
```

Playing an animation from code is a single line — `$AnimationPlayer.play("name")` — regardless of how many properties or keyframes that animation actually contains.

A Simple Particle Burst — GPUParticles2D

`GPUParticles2D` spawns a burst of small, short-lived visual elements — sparks, dust, a puff of smoke on impact. A one-time burst (rather than a continuous effect like rain) uses its own `one_shot` property.

```
# --- attached to a GPUParticles2D, one_shot enabled in the Inspector ---
extends GPUParticles2D

func burst() -> void:
    restart()
```

Setting `emitting = true` a second time doesn't restart a one-shot burst — once a `one_shot` particle system finishes its single burst, it won't emit again just because `emitting` is set to true; that only resumes an already-stopped continuous effect. To genuinely trigger a fresh burst — for a repeated effect like an impact spark on every hit — call `restart()` instead, which does cleanly begin a new emission (interrupting any particles from a still-playing previous burst in the process).

COMING FROM PYTHON

None of AudioStreamPlayer, AnimationPlayer, or GPUParticles2D have a close Python standard-library equivalent — they're genuinely engine-specific tools, closer in spirit to calling into a dedicated game audio/animation library than anything built into the language itself. What should feel familiar is the shape of how they're used: each one is triggered by a single method call (`.play()`, `.restart()`) from exactly the same signal-driven event handlers this course has built since Chapter 3 — the "juice" in this chapter is entirely about reacting to events that already exist, not new game logic.

Coding Challenges

CHALLENGE 1

Rebuild the Coin scene from Chapter 6 with an added AudioStreamPlayer2D. On collection, hide the sprite and disable the collision shape immediately, play the sound, `await` its `finished` signal, and only then call `queue_free()`.

→ Solution

CHALLENGE 2

Add an AnimationPlayer to a Sprite2D with a short "flash" animation (e.g. briefly changing `modulate` to red and back). Connect it to Chapter 7's `health_changed` signal so the sprite flashes every time `take_damage()` is called.

→ Solution

CHALLENGE 3

Add a one-shot GPUParticles2D to a scene. Write a function that calls `restart()` on it, and call that function twice in a row a second apart (using `await` and a timer) to confirm a fresh burst genuinely happens both times, not just the first.

→ Solution

QUICK REFERENCE — AUDIO & POLISH

- AudioStreamPlayer — non-positional (music, UI); AudioStreamPlayer2D — positional (world sounds)
- `.play()` to start a sound; `await player.finished` to wait for it to end
- AnimationPlayer records keyframed property changes; `$AnimationPlayer.play("name")` plays one
- GPUParticles2D — a burst effect; `one_shot` for a single burst
- `restart()`, not `emitting = true`, genuinely re-triggers a one-shot burst

CHAPTER 10: CAPSTONE — BUILDING A COMPLETE SMALL GAME

Godot Fundamentals

Chapter 10 · Capstone: Building a Complete Small Game

Nine chapters have each built one working piece in isolation — a moving sprite, a signal, a health bar, a sound effect. This capstone wires every one of them together into a single, genuinely playable small game: **Coin Dash** — move a player around a small arena, collect every coin while avoiding a hazard that damages you on contact, and win once every coin is gone (or lose if your health reaches zero first).

CHAPTER	WHAT IT CONTRIBUTES TO COIN DASH
1 — Game Loop	The overall Node/Scene structure; `_physics_process` for every moving piece
2 — GDScript Basics	Typed variables/functions and control flow throughout every script below
3 — Nodes & Scenes	The Coin scene, `preload` / `instantiate()` / `add_child()` to spawn several coins, custom signals
4 — 2D Fundamentals	Sprite2D, `position` / `flip_h` for the player facing its movement direction
5 — Input & Movement	The Input Map and `Input.get_vector()` driving the player
6 — Physics & Collision	CharacterBody2D + `move_and_slide()` for the player; Area2D + `body_entered` for the Coin and Hazard
7 — Signals & Events	The Health system's `health_changed` / `died` signals; a decoupled score system
8 — UI & HUD	A CanvasLayer HUD with a ProgressBar health bar, a score Label, and a pause menu
9 — Audio & Polish	A coin pickup sound (`await ... .finished`), a damage-flash AnimationPlayer animation, a win-condition particle burst

Scene Structure

```
Main.tscn (Node2D)
├── Player (CharacterBody2D)          - Ch5, Ch6
│   ├── Sprite2D                      - Ch4
│   ├── CollisionShape2D              - Ch6
│   ├── AnimationPlayer ("flash")     - Ch9
│   └── Health (Node)                 - Ch7
├── Hazard (Area2D)                   - Ch6
│   ├── Sprite2D
│   └── CollisionShape2D
├── HUD (CanvasLayer)                 - Ch8
│   ├── HealthBar (ProgressBar)
│   ├── ScoreLabel (Label)
│   └── PauseMenu (Control, hidden)
└── CoinSpawner (Node2D)              - Ch3

Coin.tscn (Area2D)                    - Ch3, Ch6, Ch9
├── Sprite2D
├── CollisionShape2D
└── AudioStreamPlayer2D
```

The Player

Movement is Chapter 5's four-directional controller, running through Chapter 6's `CharacterBody2D` so it respects the Hazard's collision, with Chapter 4's `flip_h` facing added on top.

PLAYER.GD (ATTACHED TO PLAYER)

```gdscript
extends CharacterBody2D

const SPEED = 200.0

func _physics_process(delta: float) -> void:
    var direction := Input.get_vector("move_left", "move_right", "move_up", "move_down")
    velocity = direction * SPEED
    move_and_slide()

    if direction.x < 0:
        $Sprite2D.flip_h = true
```

```
    elif direction.x > 0:
        $Sprite2D.flip_h = false
```

The Player is tagged into the `"player"` group (Chapter 6), so both the Coin and Hazard can recognize it by group membership rather than by name.

HEALTH.GD (ATTACHED TO THE HEALTH CHILD NODE)

```
extends Node

signal health_changed(current: int, max_health: int)
signal died

var current: int = 100
var max_health: int = 100

func take_damage(amount: int) -> void:
    current = max(current - amount, 0)
    health_changed.emit(current, max_health)

    if current == 0:
        died.emit()
```

The Hazard

An Area2D that damages the player on contact — the same `body_entered` pattern as the coin from Chapter 6, applied to harming rather than collecting.

HAZARD.GD (ATTACHED TO HAZARD)

```
extends Area2D

func _ready() -> void:
    body_entered.connect(_on_body_entered)

func _on_body_entered(body: Node2D) -> void:
```

```
    if body.is_in_group("player"):
        body.get_node("Health").take_damage(20)
```

The Coin

The full Chapter 9 version — hide and disable immediately on collection, play the pickup sound, and only `queue_free()` once that sound has finished — plus a custom `coin_collected` signal (Chapter 3/7) that Main.tscn listens for.

COIN.GD (ATTACHED TO COIN.TSCN'S ROOT)

```
extends Area2D

signal coin_collected

func _ready() -> void:
    body_entered.connect(_on_body_entered)

func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        $CollisionShape2D.set_deferred("disabled", true)
        $Sprite2D.hide()
        coin_collected.emit()
        $AudioStreamPlayer2D.play()
        await $AudioStreamPlayer2D.finished
        queue_free()
```

The HUD

A health bar wired to `Health.health_changed`, a score label, and a pause menu — Chapter 8's three pieces, unchanged in shape.

HEALTH_BAR.GD (ATTACHED TO HUD/HEALTHBAR)

```
extends ProgressBar
```

```
func _ready() -> void:
    var health = get_node("/root/Main/Player/Health")
    health.health_changed.connect(_on_health_changed)
    min_value = 0
    max_value = health.max_health
    value = health.current

func _on_health_changed(current: int, max_health: int) -> void:
    value = current
```

PAUSE_MENU.GD (ATTACHED TO HUD/PAUSEMENU)

```
extends Control

func _ready() -> void:
    hide()

func _process(delta: float) -> void:
    if Input.is_action_just_pressed("pause"):
        visible = not visible
```

Spawning Coins & Wiring the Win/Lose Conditions

This is the chapter's own central piece: `Main.tscn`'s root script instances several Coins at runtime (Chapter 3), listens for every one of their `coin_collected` signals to update the score and check the win condition, and listens for the Player's `Health.died` signal for the lose condition.

MAIN.GD (ATTACHED TO MAIN.TSCN'S ROOT)

```
extends Node2D

const CoinScene: PackedScene = preload("res://coin.tscn")
const COIN_POSITIONS = [
    Vector2(100, 100), Vector2(300, 150), Vector2(500, 100),
    Vector2(200, 350), Vector2(450, 380),
]
```

```
var score: int = 0
var coins_remaining: int = 0

func _ready() -> void:
    $Player/Health.died.connect(_on_player_died)

    for pos in COIN_POSITIONS:
        var coin = CoinScene.instantiate()
        $CoinSpawner.add_child(coin)
        coin.position = pos
        coin.coin_collected.connect(_on_coin_collected)
        coins_remaining += 1

func _on_coin_collected() -> void:
    score += 10
    coins_remaining -= 1
    $HUD/ScoreLabel.text = "Score: " + str(score)

    if coins_remaining == 0:
        $WinParticles.restart()
        print("You collected every coin - you win!")

func _on_player_died() -> void:
    print("Game over")
    get_tree().paused = true
```

Every one of these connections happens at runtime, on instanced or already-placed nodes — `main.gd` is the one script in the whole game that actually knows the Player, the HUD, and every Coin all exist. None of *them* know about each other or about `main.gd` itself. Health doesn't know a HUD exists; Coin doesn't know a score system exists; Player doesn't know Coin exists at all. This is Chapter 3's and Chapter 7's own decoupling principle, scaled up to a full game: every piece is independently reusable, and `main.gd`'s only job is wiring already-independent pieces together.

Playing Coin Dash

With everything above in place: move the player with the arrow keys or WASD, walk into a coin to collect it (with a sound and a score update), avoid the hazard (or don't, and watch the health

bar drop with a flash), press Escape to pause, and collect all five coins to trigger the win-condition particle burst.

Coding Challenges — Extending Coin Dash

CHALLENGE 1

Add a second Hazard that patrols back and forth between two points using `_physics_process` and `position` (Chapter 4's own movement pattern), rather than sitting still. It should still damage the player on contact exactly like the first Hazard.

→ Solution

CHALLENGE 2

Extend `main.gd`'s `_on_player_died` so that, instead of only printing "Game over," it also shows a Control-based "Game Over" screen (built the same way as the PauseMenu) with the final score displayed on a Label.

→ Solution

CHALLENGE 3

Give each Coin a random chance (e.g. 20%) of being a "big coin" worth 50 points instead of 10, decided once when it's instanced in `main.gd`. Have the coin's own `coin_collected` signal pass its point value as an argument so `_on_coin_collected` can add the correct amount.

→ Solution

Where to Go From Here

Coin Dash deliberately stays small — one screen, no level transitions, no save system, no enemy AI beyond a simple patrol. Every one of those is a natural next step once the fundamentals in this course feel solid: autoloads/singletons for global game state (mentioned briefly in Chapter

8), multiple scenes and level transitions, tilemaps for building larger levels, and enemy AI beyond simple back-and-forth movement are all genuine Intermediate/Advanced territory. A Unity/C# sibling course under this same Game Development subject remains a real future option too, reusing this site's own existing C# Fundamentals and C# Intermediate/Advanced courses as a running start.

QUICK REFERENCE — WHAT COIN DASH DEMONSTRATES

- CharacterBody2D player movement (Ch5, Ch6) with facing (Ch4)
- Area2D hazard and coins, both using `body_entered` (Ch6)
- Runtime scene instancing for multiple coins (Ch3)
- A decoupled Health system and a decoupled coin-collection system (Ch7)
- A HUD reacting to signals with zero direct references back into it (Ch8)
- A pickup sound that finishes before its node is freed, and a win-condition particle burst (Ch9)
- One "wiring" script (`main.gd`) as the only node that knows every other piece exists