

CS354R

DR SARAH ABRAHAM

GODOT OVERVIEW

OPEN SOURCE GAME ENGINE

- ▶ Godot is open source under the MIT license and provides:
 - ▶ Cross-platform development support
 - ▶ GUI editor tools
 - ▶ Support for 2D and 3D game development
 - ▶ Ability to export to multiple platforms
- ▶ Free to use and readily viewable/modifiable code

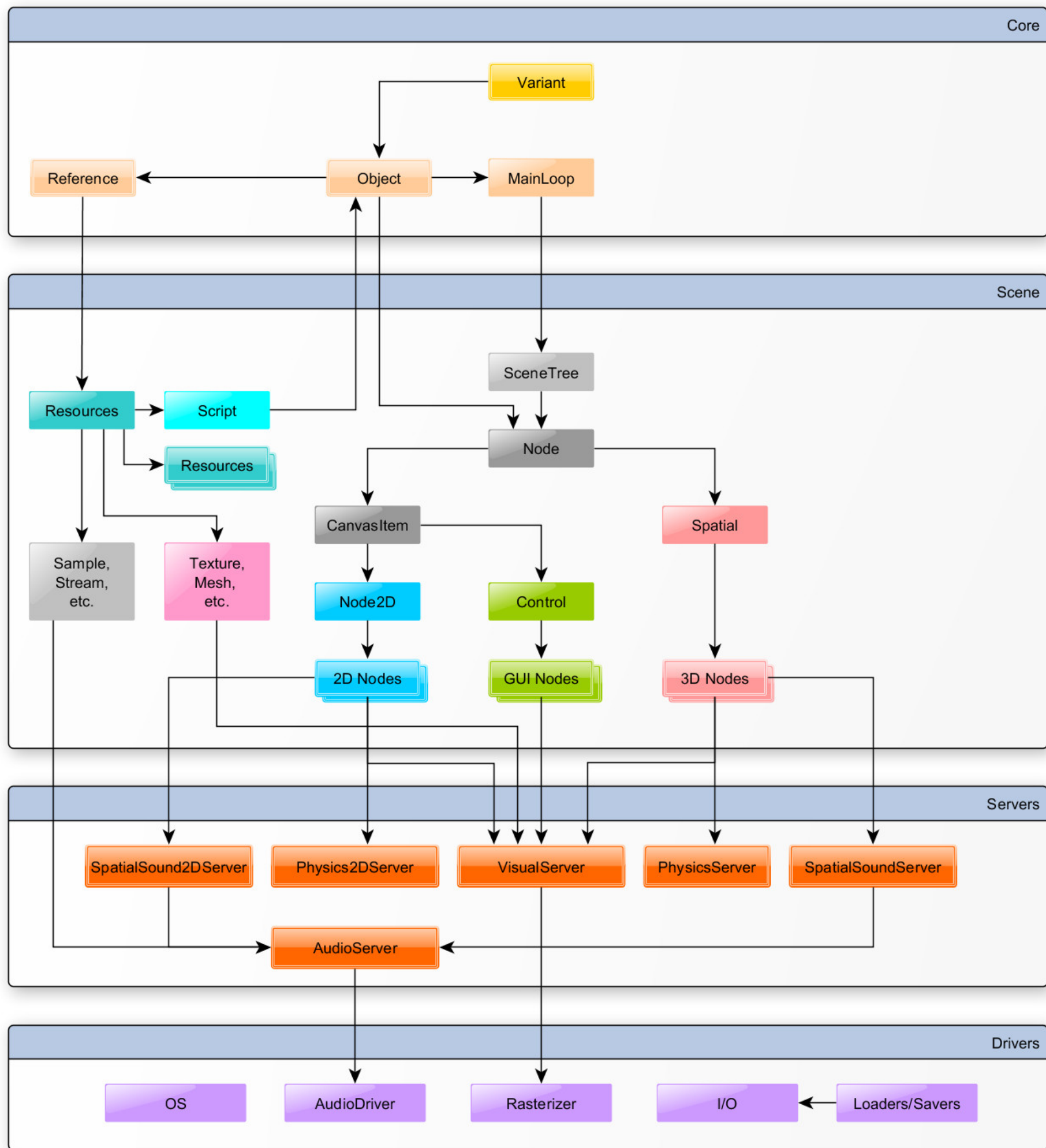


ADDITIONAL ASSETS

- ▶ Additional plugins and modules available in the Asset Library:
 - ▶ <https://godotengine.org/asset-library/asset>
- ▶ Can also import materials and 3D meshes from asset creation tools like Blender and Maya
 - ▶ Godot does not support fbx, so export as collada
- ▶ 3D models and animations can be found on <https://www.turbosquid.com/>
 - ▶ Only some models are free
 - ▶ Check copyright before using

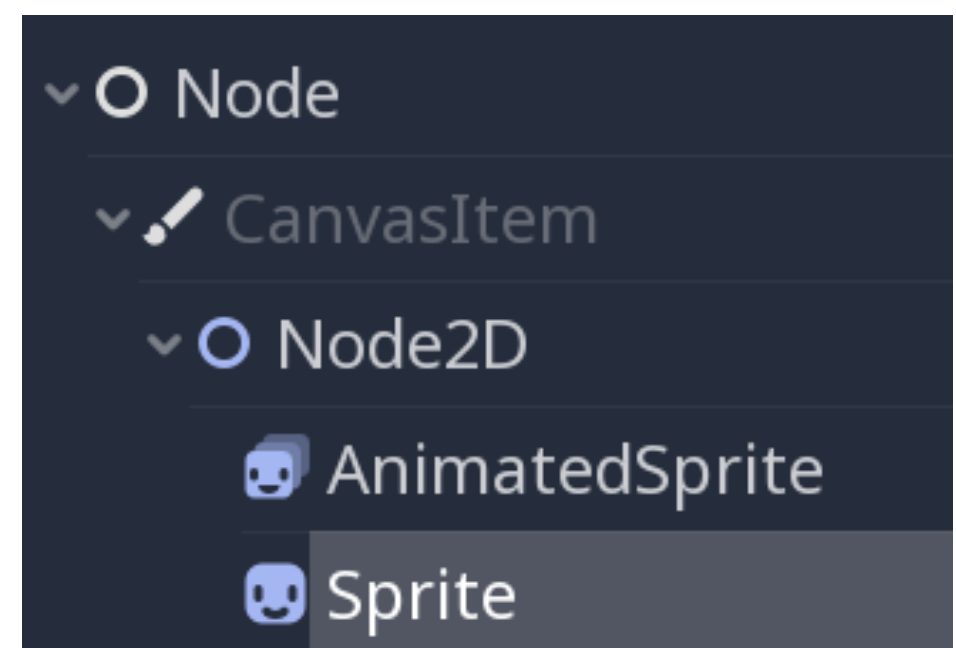
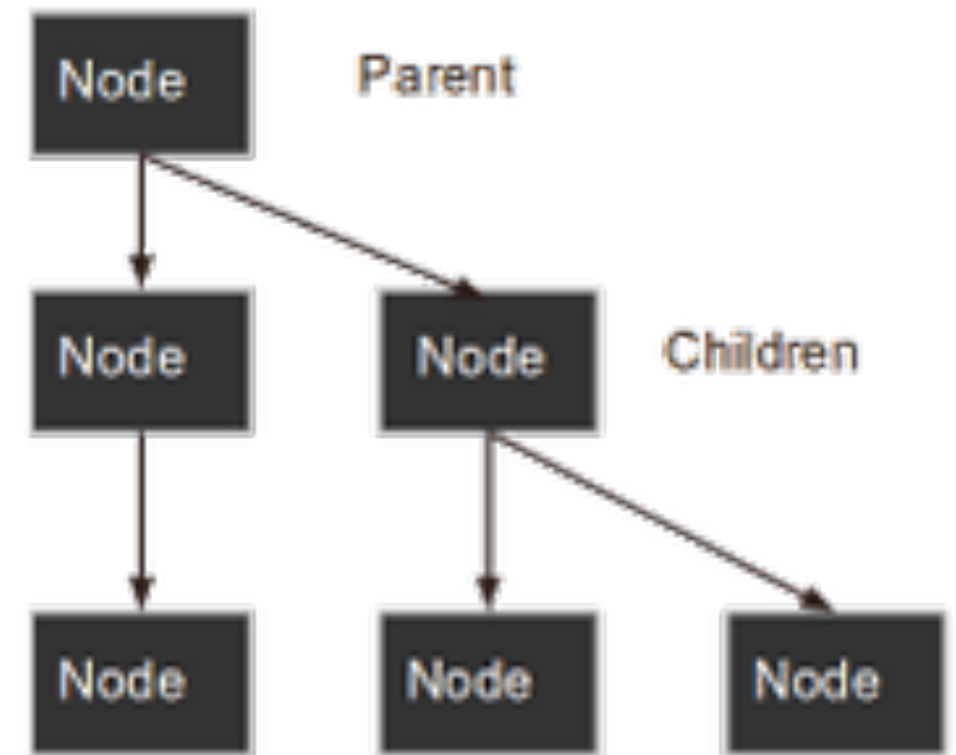
GODOT ARCHITECTURE

- ▶ Godot is object-oriented
 - ▶ Allows both composition and aggregation
- ▶ Nodes are fundamental building blocks
- ▶ Scenes are composed of nodes
- ▶ Everything is built on these two hierarchies



NODES

- ▶ Nodes can:
 - ▶ Be named
 - ▶ Have editable properties
 - ▶ Receive callbacks
 - ▶ Be extended
 - ▶ Be a child of another node
- ▶ Nodes inherit from all their parents
 - ▶ Has all properties in hierarchy

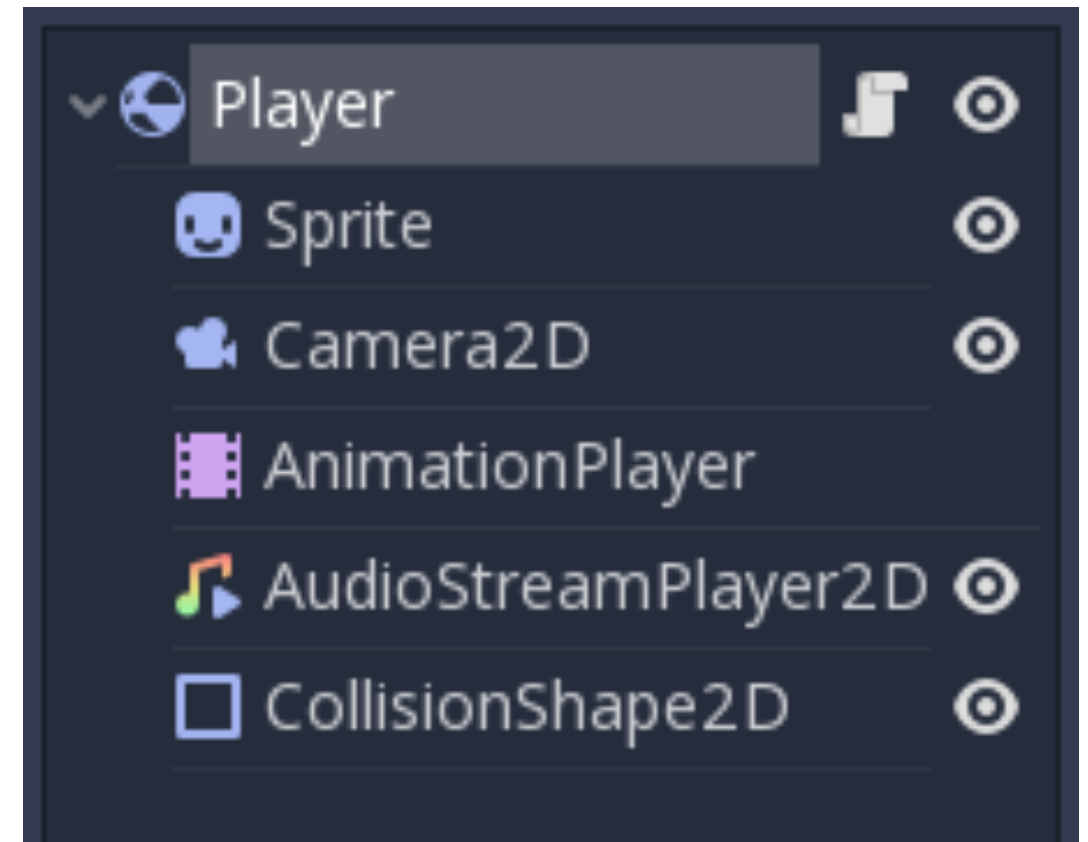


NODE FUNCTIONALITY

- ▶ Base class for all scene objects
- ▶ Some of its properties:
 - ▶ `name (String)`
 - ▶ `owner (Node)`
- ▶ Some of its functions:
 - ▶ `_process ( )`
 - ▶ `_physics_process(float delta)`
 - ▶ `_ready ( )`
- ▶ Inherits from `Object`, which is base class for all classes
 - ▶ `Servers`, `MainLoop`, `Loaders`, etc

SCENES

- ▶ Formed from a hierarchy of nodes
 - ▶ Have one root node
 - ▶ Can be saved and loaded from disk
 - ▶ Can be instanced
- ▶ Every game has a main scene (.tscn)
- ▶ Scenes can be anything
 - ▶ Levels, characters, cameras, etc
- ▶ Inheritance allows for nested prefabs
 - ▶ Changing parent modifies all children
- ▶ Scenes can be instanced and added to other scenes



SERVERS

- ▶ Used for multithreading and data separation
- ▶ Implement mediator pattern to process data and push results back to the engine
- ▶ Servers run as parallel as possible and sync only when necessary
 - ▶ Used in lieu of a heavier-weight job scheduler
- ▶ Servers **do not handle objects or classes**
 - ▶ Contain references to RID (Resource ID) types
 - ▶ Similar to the "System" part of ECS
- ▶ Servers for physics, rendering, logic, etc

VARIANTS

- ▶ Common type used in Godot
 - ▶ Maximum of 20 bytes
 - ▶ Can contain most Godot engine data types
- ▶ Used for communication, editing, and serialization
 - ▶ Not for long-term storage
 - ▶ Allows for dynamic type handling more efficiently than native C++
- ▶ Dictionaries and Arrays both implemented using Variants
- ▶ **Data retrieved via callbacks may also be Variant data!!**

MULTITHREADING

- ▶ Not all of Godot is thread safe!
 - ▶ Active scene trees must be locked
 - ▶ Cannot access nodes from background processes
- ▶ Use mutexes when multiple threads need to access data
- ▶ Use semaphores when threads are waiting on data

CONCURRENCY

- ▶ `call_deferred/CallDeferred` queues up a call to run during the main thread's idle time
- ▶ `await` creates a **coroutine** or a function that can pause execution
 - ▶ Will wait on signal before continuing execution
 - ▶ Invoking coroutine with `await` will pause the calling function until the coroutine itself completes

GDSCRIPT AWAIT EXAMPLE

```
func callerFunc():  
    await coroutineFunc()  
    //This will wait for the timer to complete  
func coroutineFunc():  
    await get_tree().create_timer(1).timeout
```

```
func callerFunc():  
    coroutineFunc()  
    //This will not wait for the timer to complete  
func coroutineFunc():  
    await get_tree().create_timer(1).timeout
```

USING AWAIT WITH C#

- ▶ Use the `async` keyword to allow for asynchronous execution of instructions
- ▶ Use `await` keyword to allow for non-blocking tasks

```
public async void CallerFunc()  
{  
    //This timer will not block the thread...  
    await ToSignal(GetTree()).CreateTimer(1, "timeout");  
    //But it's still waiting for timer to complete...  
}
```

GODOT SCRIPTING

- ▶ GDScript is native Godot scripting language
 - ▶ Integrated into editor
- ▶ C# and C++ bindings also available
 - ▶ Must be developed in separate IDE
- ▶ GDExtensions allows C++ scripting without recompiling the engine
 - ▶ Can also add C++ functionality via modules but must recompile engine

GDSCRIPT

- ▶ Designed specifically for game development/Godot engine
 - ▶ Optimized for Godot
 - ▶ Integrated in the editor
 - ▶ Built-in types for linear algebra
 - ▶ Multithreading support
 - ▶ No garbage collection (uses reference counting)
 - ▶ Dynamically typed
- ▶ Accesses functions and properties of node that script is attached to
- ▶ Example:

```
func _on_Button_pressed():  
    get_node("Label").text = "Hello!"
```

GDEXTENSION

- ▶ Module that can run native code and load shared libraries
 - ▶ Connects third party libraries to Godot without recompiling the engine
- ▶ Has C++ bindings (separate download from Godot source)
 - ▶ Can access whole of GDScript API

WHY GDEXTENSION?

- ▶ Allows for highly optimized code without modifying source to create modules
- ▶ Connects additional third-party libraries to project without modifying source

BUT WHY ARE WE USING GDEXTENSION?

- ▶ Generally speaking, you'll try to develop in GDScript/C# for general purpose game development
- ▶ Using GDExtension for everything ignores a lot of the power Godot provides
 - ▶ But this is a game technology/engine building class, so we are going to take a lower level approach :)
 - ▶ "Pulls back covers" on some of the things the higher level scripting languages intentionally hide

GDEXTENSION SETUP

- ▶ For gameplay objects, inherit from Object class
 - ▶ Expected includes
 - ▶ Constructor/Destructor
 - ▶ Necessary properties and functions
- ▶ GDCLASS () macro required for internal setup:

```
class GDSprite : public Sprite2D {  
    GDCLASS(GDSprite, Sprite2D)
```

REGISTERING FUNCTIONS AND PROPERTIES

- ▶ Must expose properties and functions that are called from Godot editor
- ▶ Use static function `_bind_methods()`
 - ▶ Call `bind_method(D_METHOD("method name"), &method)` to expose methods
 - ▶ Call `add_property("Class Name", PropertyInfo(Variant::TYPE, "name", PROPERTY_HINT_RANGE, "min, increment, max"), "setter", "getter");` to expose property

INTER-NODE COMMUNICATION?

SIGNALS

- ▶ Implementation of Godot's **observer** pattern
 - ▶ Observer nodes *subscribe* to a subject node's messages
 - ▶ Subject nodes *emit* messages that all observing nodes will execute
- ▶ Used for callbacks on events
- ▶ Allows for decoupled nodes to communicate safely

CONNECTING SIGNALS

- ▶ Can connect in GDExtension:
 - ▶ `subject->connect("signalname", Callable(observer, StringName("callback")));`
- ▶ Can connect through editor interface
- ▶ Must register signal in `_register_methods`
 - ▶ `ADD_SIGNAL(MethodInfo("signal name", PropertyInfo(Variant::TYPE, "signal parameter 1"), PropertyInfo(Variant::TYPE, "signale parameter 2"), ...));`

EMITTING AND RECEIVING SIGNALS

- ▶ By convention, Godot uses naming:
`_on_<subjectname>_<signalname>` for target function
 - ▶ Must create this function in the observing node
- ▶ Can emit a signal by calling `emit_signal("signalname", parameter1, parameter2, ...);`
- ▶ Many nodes have existing signals you can access:
 - ▶ Area nodes have `area_entered(Area area)/area_exited(Area area)`
 - ▶ BaseButton nodes have `button_down()/button_up()`

PUTTING IT ALL TOGETHER

- ▶ Must create a library of all extensions within the module as a plugin
- ▶ `initialize_name_module` and `uninitialize_name_module` called when plugin is loaded and unloaded
- ▶ Register all classes in the library `register_types`
 - ▶ `ClassDB::register_class<class>()`;
- ▶ Compile the plugin using SCons (may have to close Godot and recompile after changes)
- ▶ Define where dynamic libraries are loaded from (.gdextension)

GODOT CASTS

- ▶ Note that while C-style casts compile, this may create runtime issues
 - ▶ Use Godot-style casts at all times: `Object::cast_to<MyClass>(foo);`
- ▶ Casting Variants to the object's actual type requires a slightly different syntax
 - ▶ `Object::cast_to<MyClass>(MyClass::__get_from_variant(variant));`
- ▶ Casts will return a *pointer* to that object
- ▶ Always check that pointer is not null before accessing it

```
if (foo) {  
    //do some stuff with foo  
}
```

OBJECTS, REFCOUNTED, AND RESOURCE

- ▶ **Object** is the base class and parent of Node class
 - ▶ Require manual memory management
 - ▶ Possible to create own lighter weight classes
- ▶ **RefCounted** inherits from Object but has reference counting
- ▶ **Resource** inherits from RefCounted and act as data containers
 - ▶ Ability to serialize/deserialize
 - ▶ Only loaded once from disk

CREATING NEW REFERENCE OBJECTS

- ▶ Cannot do a standard call to `new` for `RefCounted` objects
- ▶ `RefCounted` objects are reference counted
 - ▶ Allows them to be released without a call to `free`
 - ▶ Useful for objects that are frequently created and destroyed during gameplay
- ▶ Call `ReferenceType * myReference = memnew(ReferenceType);`

DIVING INTO GDEXTENSIONS

- ▶ Not as well documented as the higher level scripting system
 - ▶ Expected to read the source code to understand available functionality
- ▶ godot-cpp folder has code for binding into the Godot engine framework
 - ▶ Code for lower level object models within src folder
 - ▶ Code for useful Godot classes and functionality within gen folder

DIVING INTO GODOT SOURCE

- ▶ Godot source code is separate from GDExtensions bindings
- ▶ Better documented but still not a fully searchable online API
 - ▶ Devs are expected to read the source code to understand details and use cases
- ▶ Main folders with functionality are **core** and **scene** for gameplay objects
- ▶ Other useful folders are **editor** (holds objects for building in editor) and **servers** (runs functionality on RIDs of game objects)

UTILITY AND ENGINE FUNCTIONS

- ▶ Utilities under `#include <godot_cpp/variant/utility_functions.hpp>`
 - ▶ `UtilityFunctions::print()`; //print to Godot console
 - ▶ `UtilityFunctions::is_same()`; //variant to variant comparison
 - ▶ ...and a bunch of math functionality...
- ▶ Engine singleton under `#include <godot_cpp/classes/engine.hpp>`
 - ▶ Checks editor vs gameplay
 - ▶ Controls physics and frame information

TUTORIALS AND GETTING STARTED

- ▶ GDNative <https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html>
- ▶ Custom Modules in C++ <https://docs.godotengine.org/en/3.1/development/cpp/custom_modules_in_cpp.html>
- ▶ Godot API <<https://docs.godotengine.org/en/stable/classes/index.html>>
- ▶ Godot Core types <https://docs.godotengine.org/en/stable/development/cpp/core_types.html>