Godot Engine – Object Class (Architecture Overview)

Introduction

This document summarizes how **Godot's Object class** works internally in C++.
It is *not* the same as the scripting API reference.
This explains the engine-side architecture behind all objects in Godot.

General Definition

`Object` is the **base class for almost everything** in Godot.
Almost all engine classes inherit directly or indirectly from it.

Objects include: - Reflection (runtime type information) - Editable properties - Methods and signals - Serialization support - Editor integration

A minimal custom object looks like this:

```
class CustomObject : public Object {
    GDCLASS(CustomObject, Object);
};
```

Using `GDCLASS()` gives the object many features automatically.

Example usage:

```
obj = memnew(CustomObject);
print_line("Object class: ", obj->get_class());

obj2 = Object::cast_to<OtherClass>(obj); // Safe casting between Object types
```

Registering an Object

`ClassDB` is the global registry of all `Object`-derived classes.

Register a class with:

```
ClassDB::register_class<MyCustomClass>();
```

This allows: - Instancing the class from scripts - Serialization/deserialization - Editor and tool access

Registering a **virtual** class (cannot be instanced):

```
ClassDB::register_virtual_class<MyCustomClass>();
```

`_bind_methods()`

Every Object-derived class may implement:

```
static void _bind_methods();
```

Inside it, you register: - Methods
- Properties
- Constants
- Signals

Example method binding:

```
ClassDB::bind_method(
    D_METHOD("methodname", "arg1name", "arg2name"),
    &MyCustomType::method
);
```

With default argument values:

```
ClassDB::bind_method(
    D_METHOD("methodname", "a", "b", "c"),
    &MyCustomType::method,
    DEFVAL(-1), DEFVAL(-2)
);
```

The `D_METHOD()` macro: - Converts names to efficient `StringName` - Argument names help introspection (ignored in release builds)

Constants (Enums)

Example enum:

```
enum SomeMode {
    MODE_FIRST,
    MODE_SECOND
};
```

To make enum parameters bindable:

```
VARIANT_ENUM_CAST(MyClass::SomeMode);
```

Binding constants:

```
BIND_CONSTANT(MODE_FIRST);
BIND_CONSTANT(MODE_SECOND);
```

Properties

Properties allow: - Editor exposure
- Serialization
- Runtime reflection

Defined using `PropertyInfo`:

```
PropertyInfo(type, name, hint, hint_string, usage_flags)
```

Example integer range:

```
PropertyInfo(Variant::INT, "amount",
             PROPERTY_HINT_RANGE, "0,49,1",
             PROPERTY_USAGE_EDITOR)
```

Example enum property:

```
PropertyInfo(Variant::STRING, "modes",
             PROPERTY_HINT_ENUM,
             "Enabled,Disabled,Turbo")
```

Binding via setter/getter:

```
ADD_PROPERTY(PropertyInfo(Variant::INT, "amount"), "set_amount", "get_amount");
```

Advanced Property Binding

Override these (NOT virtual):

```
void _get_property_list(List<PropertyInfo> *r_props) const;
bool _get(const StringName &p_property, Variant &r_value) const;
bool _set(const StringName &p_property, const Variant &p_value);
```

Useful for: - Dynamic properties - Context-based properties

Downside: slower (string comparisons)

Dynamic Casting

```
Button *button = Object::cast_to<Button>(some_obj);
```

If cast fails → returns **NULL**.

Works without RTTI (slower but functional), ideal for: - HTML5 - Consoles - Small-binary builds

---

Signals

Objects can define signals (similar to delegates/events).

Connecting:

```
obj->connect("enter_tree", this, "_node_entered_tree");
```

Adding signals:

```
ADD_SIGNAL(MethodInfo("been_killed"));
```

Methods used as callbacks must be registered via `bind_method()`.

Notifications

Objects receive engine-level callbacks via:

```
void _notification(int what);
```

Examples: - Tree entering/leaving - Ready/not-ready states - Input notifications

See *Godot notifications documentation* for full list.

Reference Counting (RefCounted)

`RefCounted` inherits from `Object` and adds automatic memory management.

Example:

```
class MyReference : public RefCounted {
    GDCLASS(MyReference, RefCounted);
};

Ref<MyReference> myref = memnew(MyReference);
```

When no `Ref<>` exists → object is freed automatically.

Resources

`Resource` inherits from `RefCounted`.

A Resource: - Can have a **path** to a file - Is reference-counted - Can be shared across scenes

Rules: - Two different resources cannot share the same path - Subresources without paths get auto-IDs like:
`res://file.res::1`

Resource Loading

Using `ResourceLoader`:

```
Ref<Resource> res = ResourceLoader::load("res://someresource.res");
```

If already loaded → returns existing reference (only one copy in memory).

Resource Saving

Using `ResourceSaver`:

```
ResourceSaver::save("res://someresource.res", instance);
```

With behavior: - Subresources with paths → saved as references - Subresources without paths → embedded with auto IDs

---

References

- `core/object/object.h`
- `core/object/class_db.h`
- `core/object/reference.h`
- `core/io/resource.h`
- `core/io/resource_loader.h`

Source

Based on:
https://docs.godotengine.org/en/stable/engine_details/architecture/object_class.html