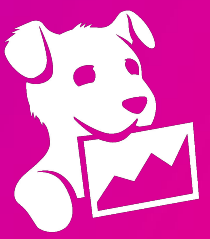
Adding new Script Languages to Godot

GodotCon Boston 2025

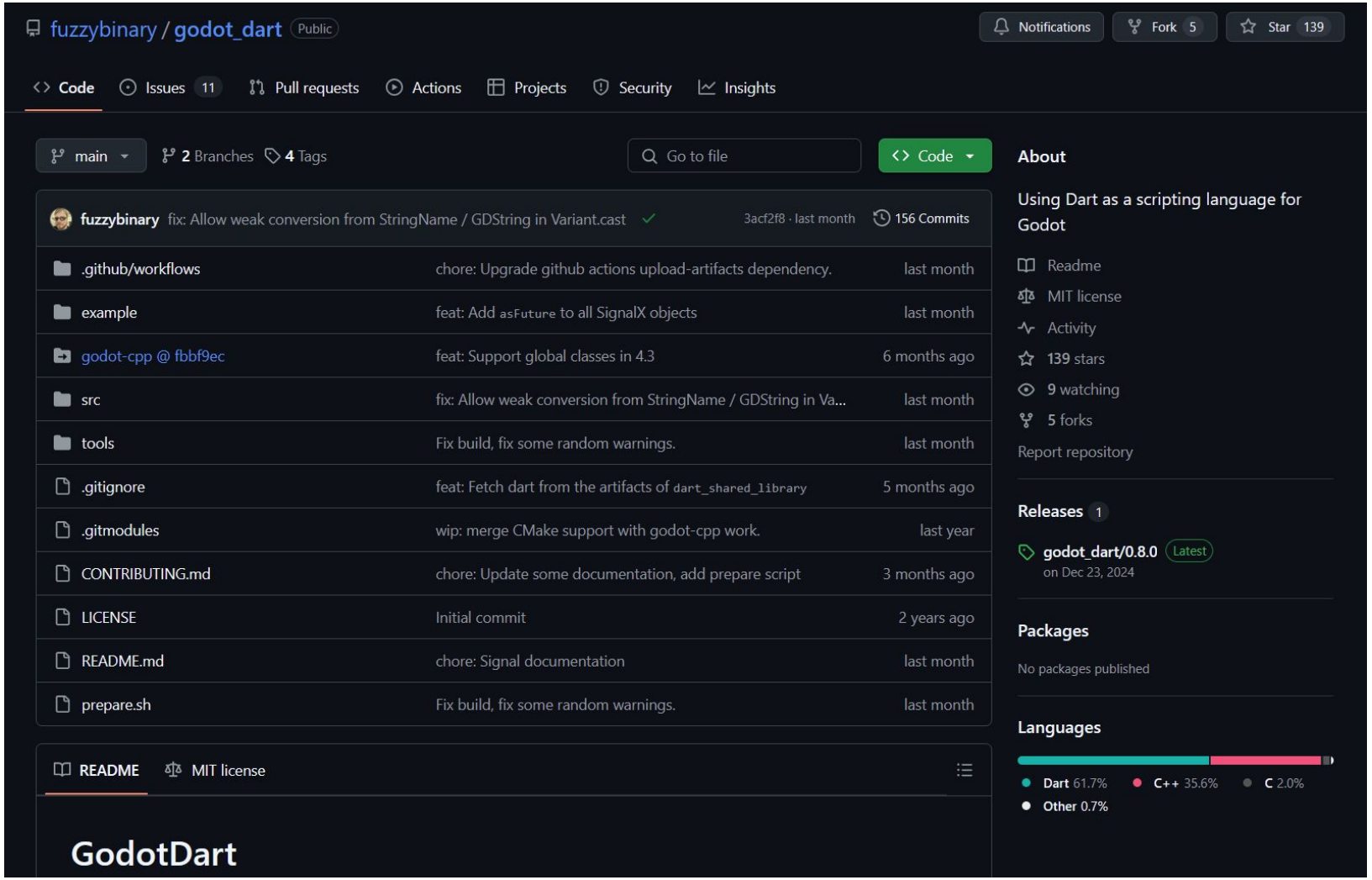


DATADOG

Jeff Ward

Senior Software Engineer, RUM SDKs
Datadog

What I Do In My Spare Time



Why?

- GDScript is great!
  - Syntactics sugar for NodePath access! (`$node`, `%node`)
  - `@on_ready`!
  - `@preload`!
- GDScript will always be the first to gain new features!

I like working with Dart

The more flexible we can make Godot

the better it will be

Starting Point

- What I assume you know:
  - Basic understanding of GDExtension
  - How to initialize your language’s runtime
  - How to access C / FFI functions in your language

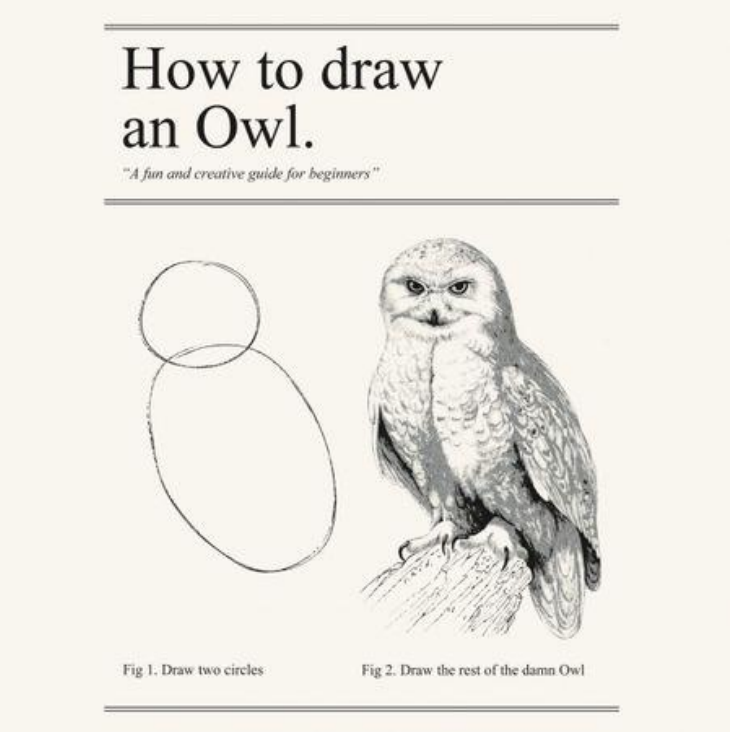
GDExtension

GDExtension

- GDExtension is how you'll load your extension and initialize your language runtime
- GDExtension is how you'll call from your Language into Godot

Suggested First Steps

- Initialize your your language from a GDExtension
- Call a method in your language from GDExtension
- Call a method in Godot from your language
- Generate bindings from extension_api.json



GDExtension Languages

C++ (Maintained by Godot)
https://github.com/godotengine/godot-cpp

Swift
https://github.com/migueldeicaza/SwiftGodot

Rust
https://godot-rust.github.io/

D
https://github.com/godot-dlang/godot-dlang

Key Areas of GDExtension

Subtitle, 18pt Normal

Built Ins vs Engine Classes

Built Ins

- Contain all their own memory and are usually copied
- Accessing properties is usually just about reading memory.
- Have their own methods in GDExtension for construction, destruction, referencing, dereferencing, and calling methods

Engine Classes

- Inherit from Object
- Pointers - passed by reference
- Some are Reference Counted (inherit from RefCounted)
- Constructed by ClassDb
- Have separate methods in GDExtension for calling methods, accessing properties.

Built Ins vs Engine Classes

Built Ins

- Allocate memory
- Call the correct constructor

Engine Classes

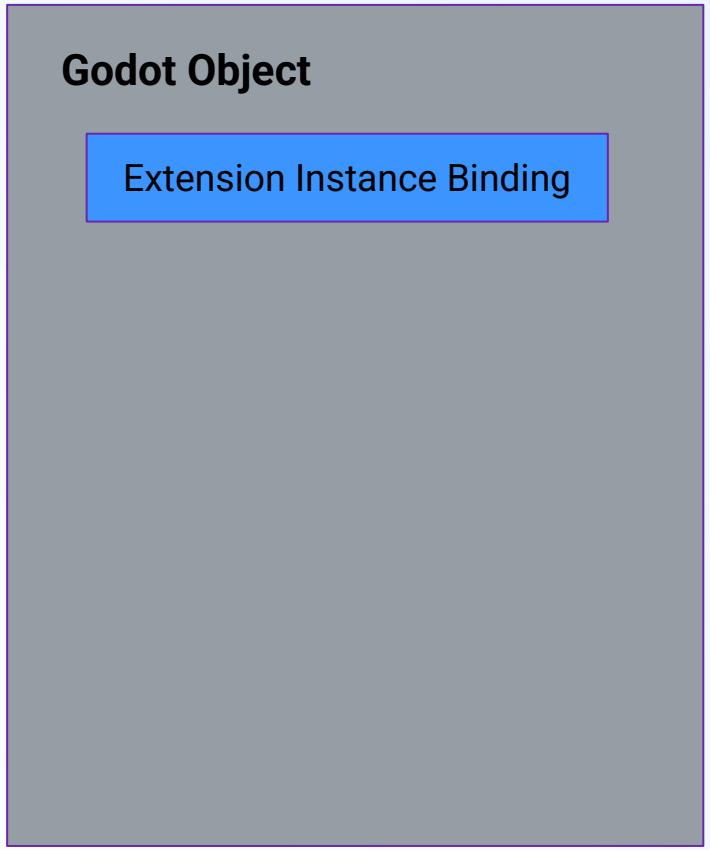
- Ask the ClassDb to create the instance for you.
- Tie your language's version to the returned pointer via an "instance binding"

Built Ins vs Engine Classes

```dart
/// Core interface for types that can convert to Variant (the builtin types)
abstract class BuiltinType implements Finalizable {
  @internal
  static final finalizer =
      NativeFinalizer(gde.dartBindings.finalizeBuiltinObject);

  Pointer<Uint8> _opaque = nullptr;

  //...

  BuiltinType(int size, GDExtensionPtrDestructor? destructor) {
    allocateOpaque(size, destructor);
    finalizer.attach(this, _opaque.cast());
  }

  /// This constructor allows classes that we implement to lazily
  /// initialize their nativePtr members
  BuiltinType.nonFinalized();

  @protected
  Pointer<Uint8> allocateOpaque(
      int size, GDExtensionPtrDestructor? destructor) {
    _opaque =
        gde.ffiBindings.gde_mem_alloc(GodotDart.destructorSize + size).cast();
    _opaque.cast<GDExtensionPtrDestructor>().value = destructor ?? nullptr;
    return _opaque + GodotDart.destructorSize;
  }

  /// This is used by the generators to call the FFI copy constructors for
  /// builtin types, usually as part of returning them from a ptr call.
  void constructCopy(GDExtensionTypePtr ptr);
}
```

```dart
/// Core interface for engine classes
abstract class ExtensionType implements Finalizable {
  // This finalizer is used for objects we own in Dart world, and for
  // RefCounted objects that we own the last reference to. It has Godot
  // delete the object
  static final _finalizer =
      NativeFinalizer(gde.dartBindings.finalizeExtensionObject);

  GDExtensionObjectPtr _owner = nullptr;
  GDExtensionObjectPtr get nativePtr => _owner;

  TypeInfo get typeInfo;

  // Created from Dart
  ExtensionType() {
    _owner = gde.constructObject(typeInfo.nativeTypeName);
    _attachFinalizer();
    _tieDartToNative();
  }

  // Created from Godot
  ExtensionType.withNonNullOwner(this._owner) {
    _tieDartToNative();
  }
```

Godot Instance Bindings



- GDExtension uses “instance bindings” to bind info from your extension to the object
- Each Godot `Object` has a map of extensions to instance bindings
- You get them with `object_get_instance_binding`
- If it needs to construct one, you get a callback

- If you need to connect a representation of an object in your language to the Godot version, this is how you do it.

Calling a Function on a Godot Object

Get the method binding with

- `classdb_get_method_bind`

Two methods:

- `object_method_bind_call`
  - All parameters are Variants.
  - Easier to implement
  - Slower to create variants, slower to call because of conversions.
- `object_method_bind_ptrcall`
  - Parameters are pointers to their arguments.
  - Faster to call, no conversions necessary.
  - Potentially unsafe if you get your parameter types wrong

Dart Example

```dart
void addBlendShape(String name) {
  using((arena) {
    final ptrArgArray = arena.allocate<GDExtensionConstTypePtr>(sizeOf<GDExtensionConstTypePtr>() * 1);
    final gdname = StringName.fromString(name);
    (ptrArgArray + 0).value = gdname.nativePtr.cast();
    gde.ffiBindings.gde_object_method_bind_ptrcall(
      _bindings.methodAddBlendShape, nativePtr.cast(), ptrArgArray, nullptr.cast());
  });
}
```

Understanding Scripts

What’s a Script?

- It’s a Resource

- The resource is attached to a node in the same way a Texture would be.
- The Script is then “instanced” on to the Object when it’s created, and methods are called on it.

```
111
112  class Script : public Resource {
113          GDCLASS(Script, Resource);
114          OBJ_SAVE_TYPE(Script);
```

Scripts in Godot are not bindings from an Object to a Type in your Language

They are a binding from an Object to a File

Key Classes

- Each part is one or two key classes
- These are not scripts – they’re Extension classes registered to ClassDb

```
godot::ClassDB::register_class<DartScriptLanguage>();
godot::ClassDB::register_class<DartScript>();
godot::ClassDB::register_class<DartResourceFormatLoader>();
godot::ClassDB::register_class<DartResourceFormatSaver>();
```

Parts

01 Resource Loader

02 Script Extension

03 Script Instance

04 Script Language Extension

Resource Loader

- Responsible for explaining to Godot how to load your script files

- Two classes to implement
  - `ResourceFormatLoader`
  - `ResourceFormatSaver`

Resource Loader

ResourceFormatLoader

- _bind_methods
- _handles_type
- _get_recognized_extensions
- _recognize_path
- _get_resource_type
- _get_resource_script_class
- _exists
- _load

ResourceFormatSaver

- _bind_methods
- _save
- _recognize
- _recognize_path
- _get_recognized_extensions

ScriptExtension

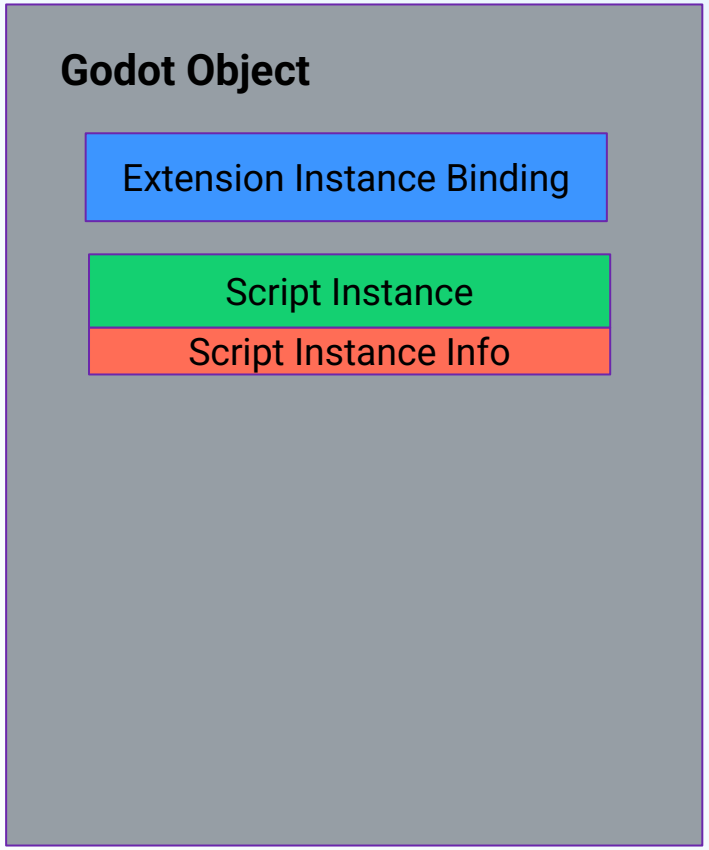
- `_get_source_code`
- `_set_source_code`

- `_instance_create`
- `_placeholder_instance_create`
- `_get_language`

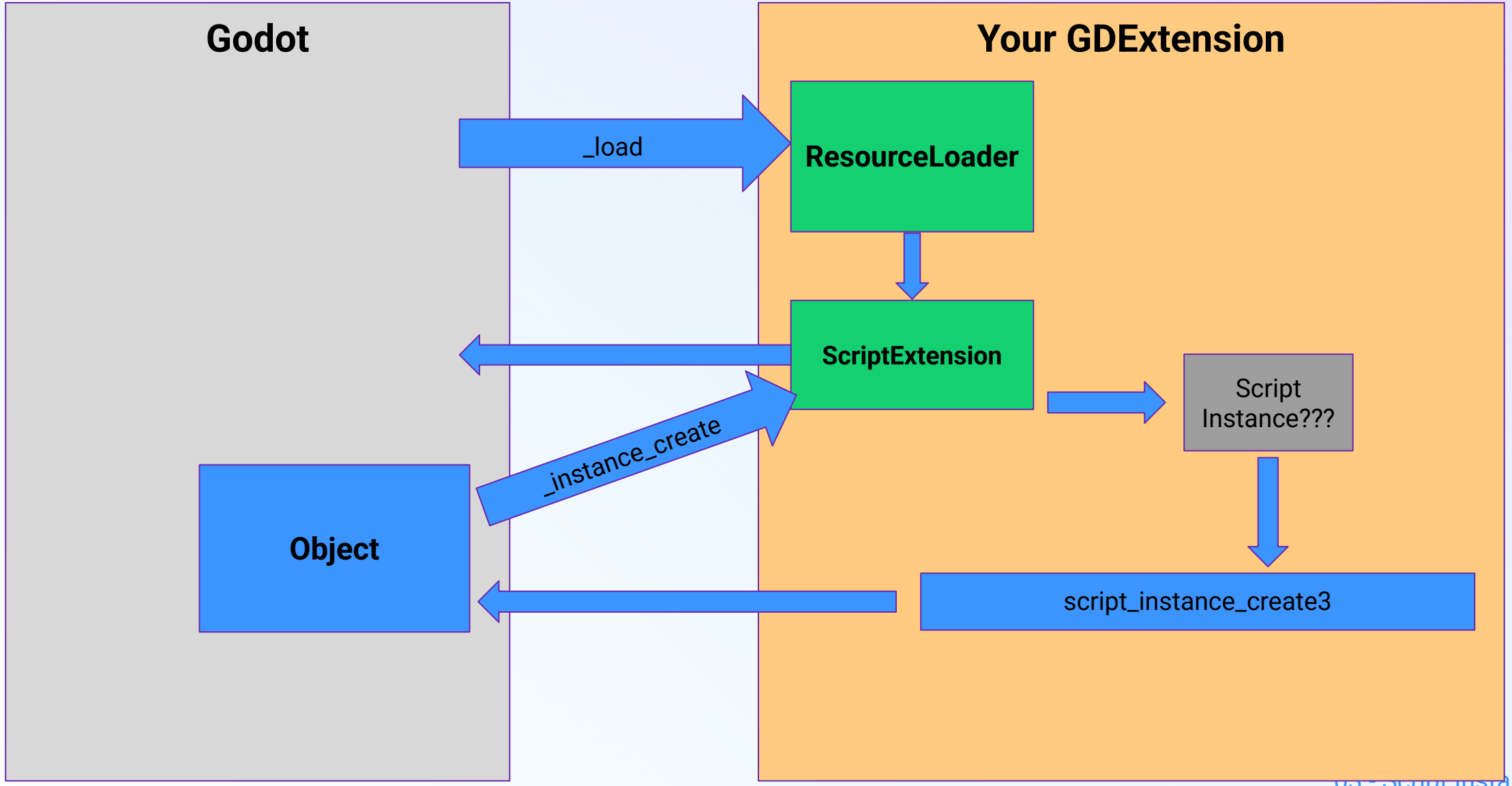
```
class ScriptExtension : public Script {
    GDEXTENSION_CLASS(ScriptExtension, Script)

public:

    virtual bool _editor_can_reload_from_file();
    virtual void _placeholder_erased(void *p_placeholder);
    virtual bool _can_instantiate() const;
    virtual Ref<Script> _get_base_script() const;
    virtual StringName _get_global_name() const;
    virtual bool _inherits_script(const Ref<Script> &p_script) const;
    virtual StringName _get_instance_base_type() const;
    virtual void *_instance_create(Object *p_for_object) const;
    virtual void *_placeholder_instance_create(Object *p_for_object) const;
    virtual bool _instance_has(Object *p_object) const;
    virtual bool _has_source_code() const;
    virtual String _get_source_code() const;
    virtual void _set_source_code(const String &p_code);
    virtual Error _reload(bool p_keep_state);
    virtual TypedArray<Dictionary> _get_documentation() const;
    virtual String _get_class_icon_path() const;
    virtual bool _has_method(const StringName &p_method) const;
    virtual bool _has_static_method(const StringName &p_method) const;
    virtual Variant _get_script_method_argument_count(const StringName &p_method) const;
    virtual Dictionary _get_method_info(const StringName &p_method) const;
    virtual bool _is_tool() const;
    virtual bool _is_valid() const;
    virtual bool _is_abstract() const;
    virtual ScriptLanguage *_get_language() const;
    virtual bool _has_script_signal(const StringName &p_signal) const;
    virtual TypedArray<Dictionary> _get_script_signal_list() const;
    virtual bool _has_property_default_value(const StringName &p_property) const;
    virtual Variant _get_property_default_value(const StringName &p_property) const;
    virtual void _update_exports();
    virtual TypedArray<Dictionary> _get_script_method_list() const;
    virtual TypedArray<Dictionary> _get_script_property_list() const;
    virtual int32_t _get_member_line(const StringName &p_member) const;
    virtual Dictionary _get_constants() const;
    virtual TypedArray<StringName> _get_members() const;
    virtual bool _is_placeholder_fallback_enabled() const;
    virtual Variant _get_rpc_config() const;
protected:
```

Script Instance

- No base class - defined by a struct of function pointers
- Does most of your heavy lifting



Creating a Script Instance



Script Instance Info

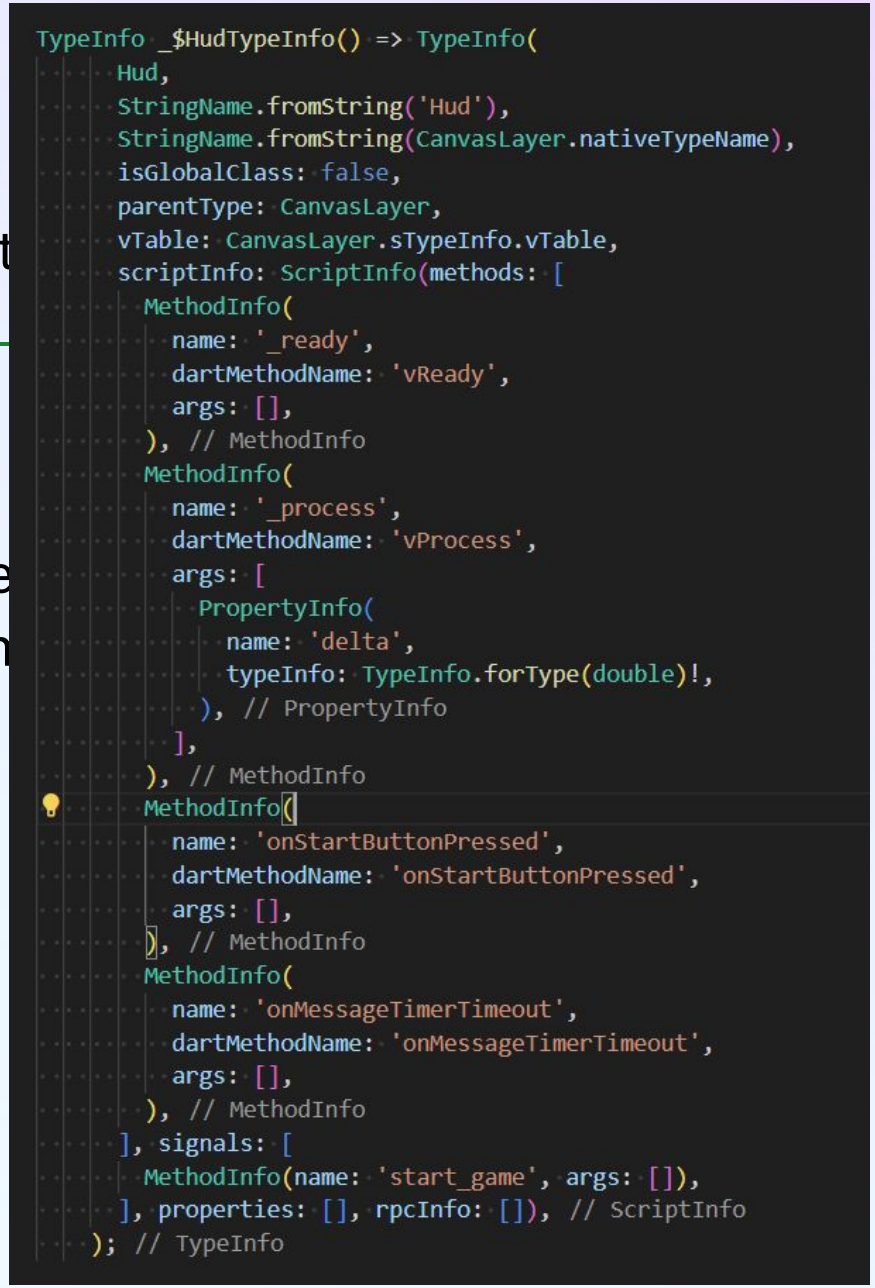
- Function pointers - each one will pass back your “Script Instance”
- Most functions revolve around introspection - getting methods and properties
- Some are method calls, or property getter / setters.
- Some revolve around memory management

```
typedef struct {
	GDExtensionScriptInstanceSet set_func;
	GDExtensionScriptInstanceGet get_func;
	GDExtensionScriptInstanceGetPropertyList get_property_list_func;
	GDExtensionScriptInstanceFreePropertyList2 free_property_list_func;
	GDExtensionScriptInstanceGetClassCategory get_class_category_func; // Optional. Set to NULL for the default behavior.

	GDExtensionScriptInstancePropertyCanRevert property_can_revert_func;
	GDExtensionScriptInstancePropertyGetRevert property_get_revert_func;

	GDExtensionScriptInstanceGetOwner get_owner_func;
	GDExtensionScriptInstanceGetPropertyState get_property_state_func;

	GDExtensionScriptInstanceGetMethodList get_method_list_func;
	GDExtensionScriptInstanceFreeMethodList2 free_method_list_func;
	GDExtensionScriptInstanceGetPropertyType get_property_type_func;
	GDExtensionScriptInstanceValidateProperty validate_property_func;

	GDExtensionScriptInstanceHasMethod has_method_func;

	GDExtensionScriptInstanceGetMethodArgumentCount get_method_argument_count_func;

	GDExtensionScriptInstanceCall call_func;
	GDExtensionScriptInstanceNotification2 notification_func;

	GDExtensionScriptInstanceToString to_string_func;

	GDExtensionScriptInstanceRefCountIncremented refcount_incremented_func;
	GDExtensionScriptInstanceRefCountDecremented refcount_decremented_func;

	GDExtensionScriptInstanceGetScript get_script_func;

	GDExtensionScriptInstanceIsPlaceholder is_placeholder_func;

	GDExtensionScriptInstanceSet set_fallback_func;
	GDExtensionScriptInstanceGet get_fallback_func;

	GDExtensionScriptInstanceGetLanguage get_language_func;

	GDExtensionScriptInstanceFree free_func;

} GDExtensionScriptInstanceInfo3;
```

Introspection Functions

- Godot will ask about the methods, signals, and propert
- And if they change, you need to notify Godot (`notify_`
  this. I think…)

- If you have a statically typed language without runtime
- Dart generates code for all of this, rather than rely on n



```
TypeInfo _$HudTypeInfo() => TypeInfo(
    Hud,
    StringName.fromString('Hud'),
    StringName.fromString(CanvasLayer.nativeTypeName),
    isGlobalClass: false,
    parentType: CanvasLayer,
    vTable: CanvasLayer.sTypeInfo.vTable,
    scriptInfo: ScriptInfo(methods: [
      MethodInfo(
        name: '_ready',
        dartMethodName: 'vReady',
        args: [],
      ), // MethodInfo
      MethodInfo(
        name: '_process',
        dartMethodName: 'vProcess',
        args: [
          PropertyInfo(
            name: 'delta',
            typeInfo: TypeInfo.forType(double)!,
          ), // PropertyInfo
        ],
      ), // MethodInfo
      MethodInfo(
        name: 'onStartButtonPressed',
        dartMethodName: 'onStartButtonPressed',
        args: [],
      ), // MethodInfo
      MethodInfo(
        name: 'onMessageTimerTimeout',
        dartMethodName: 'onMessageTimerTimeout',
        args: [],
      ), // MethodInfo
    ], signals: [
      MethodInfo(name: 'start_game', args: []),
    ], properties: [], rpcInfo: []), // ScriptInfo
  ); // TypeInfo
```

Function Calls to Script Instances

```
GDExtensionScriptInstanceGetMethodList get_method_list_func;
GDExtensionScriptInstanceFreeMethodList2 free_method_list_func;
GDExtensionScriptInstanceGetPropertyType get_property_type_func;
GDExtensionScriptInstanceValidateProperty validate_property_func;

GDExtensionScriptInstanceHasMethod has_method_func;
```

- `has_method` is given your script instance and a name
- call is given your script instance, a name, a list of arguments as Variants, and variables for return values

- Update functions are called from the editor.
- If this object is a “placeholder”, it should error with `GDEXTENSION_CALL_ERROR_INVALID_METHOD`

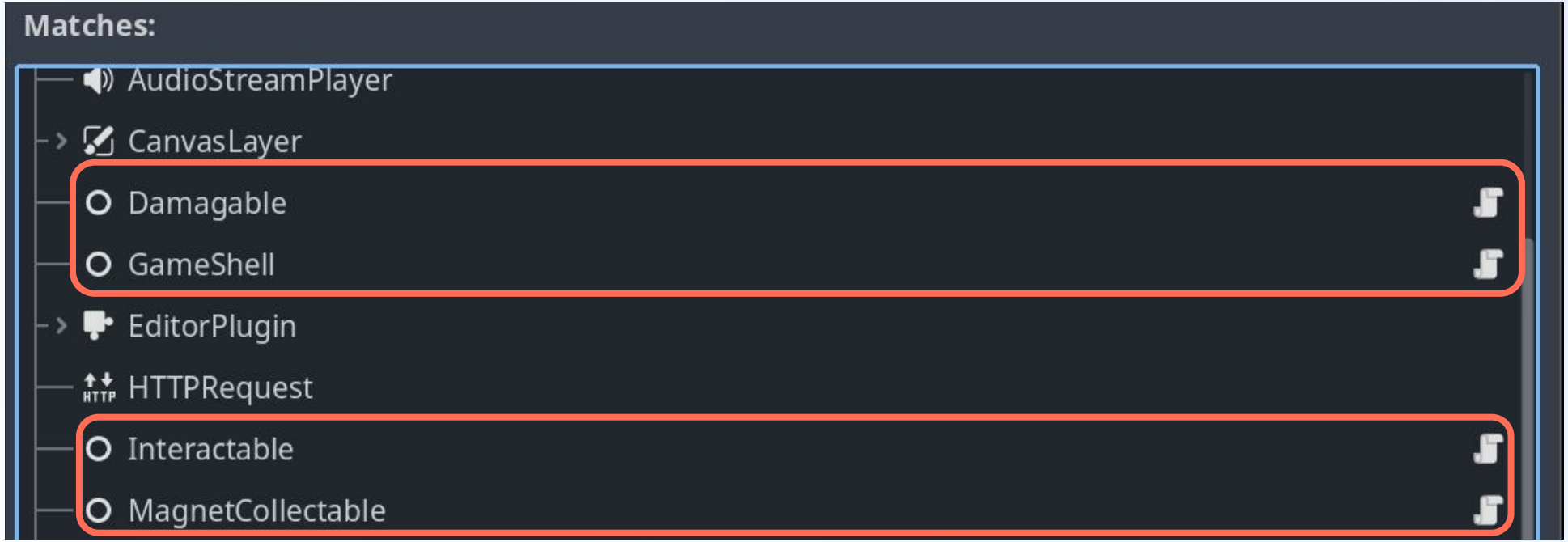
Script Language

- Quite a bit larger
- Mostly about the editor interacting with your language

```
virtual String _get_name() const;
virtual void _init();
virtual String _get_type() const;
virtual String _get_extension() const;
virtual void _finish();
virtual PackedStringArray _get_reserved_words() const;
virtual bool _is_control_flow_keyword(const String &p_keyword) const;
virtual PackedStringArray _get_comment_delimiters() const;
virtual PackedStringArray _get_doc_comment_delimiters() const;
virtual PackedStringArray _get_string_delimiters() const;
virtual Ref<Script> _make_template(const String &p_template, const String &p_class_name, const String &p_base_cl
virtual TypedArray<Dictionary> _get_built_in_templates(const StringName &p_object) const;
virtual bool _is_using_templates();
virtual Dictionary _validate(const String &p_script, const String &p_path, bool p_validate_functions, bool p_val
virtual String _validate_path(const String &p_path) const;
virtual Object *_create_script() const;
virtual bool _has_named_classes() const;
virtual bool _supports_builtin_mode() const;
virtual bool _supports_documentation() const;
virtual bool _can_inherit_from_file() const;
virtual int32_t _find_function(const String &p_function, const String &p_code) const;
virtual String _make_function(const String &p_class_name, const String &p_function_name, const PackedStringArray
virtual bool _can_make_function() const;
virtual Error _open_in_external_editor(const Ref<Script> &p_script, int32_t p_line, int32_t p_column);
virtual bool _overrides_external_editor();
virtual ScriptLanguage::ScriptNameCasing _preferred_file_name_casing() const;
virtual Dictionary _complete_code(const String &p_code, const String &p_path, Object *p_owner) const;
virtual Dictionary _lookup_code(const String &p_code, const String &p_symbol, const String &p_path, Object *p_ow
virtual String _auto_indent_code(const String &p_code, int32_t p_from_line, int32_t p_to_line) const;
virtual void _add_global_constant(const StringName &p_name, const Variant &p_value);
virtual void _add_named_global_constant(const StringName &p_name, const Variant &p_value);
virtual void _remove_named_global_constant(const StringName &p_name);
```

Script Language

- Some important exceptions
  - `_handles_global_class_type`
  - `_get_global_class_name`
- Don’t offer much over ClassDb clases, provided reloading works



A Quick Note About Memory

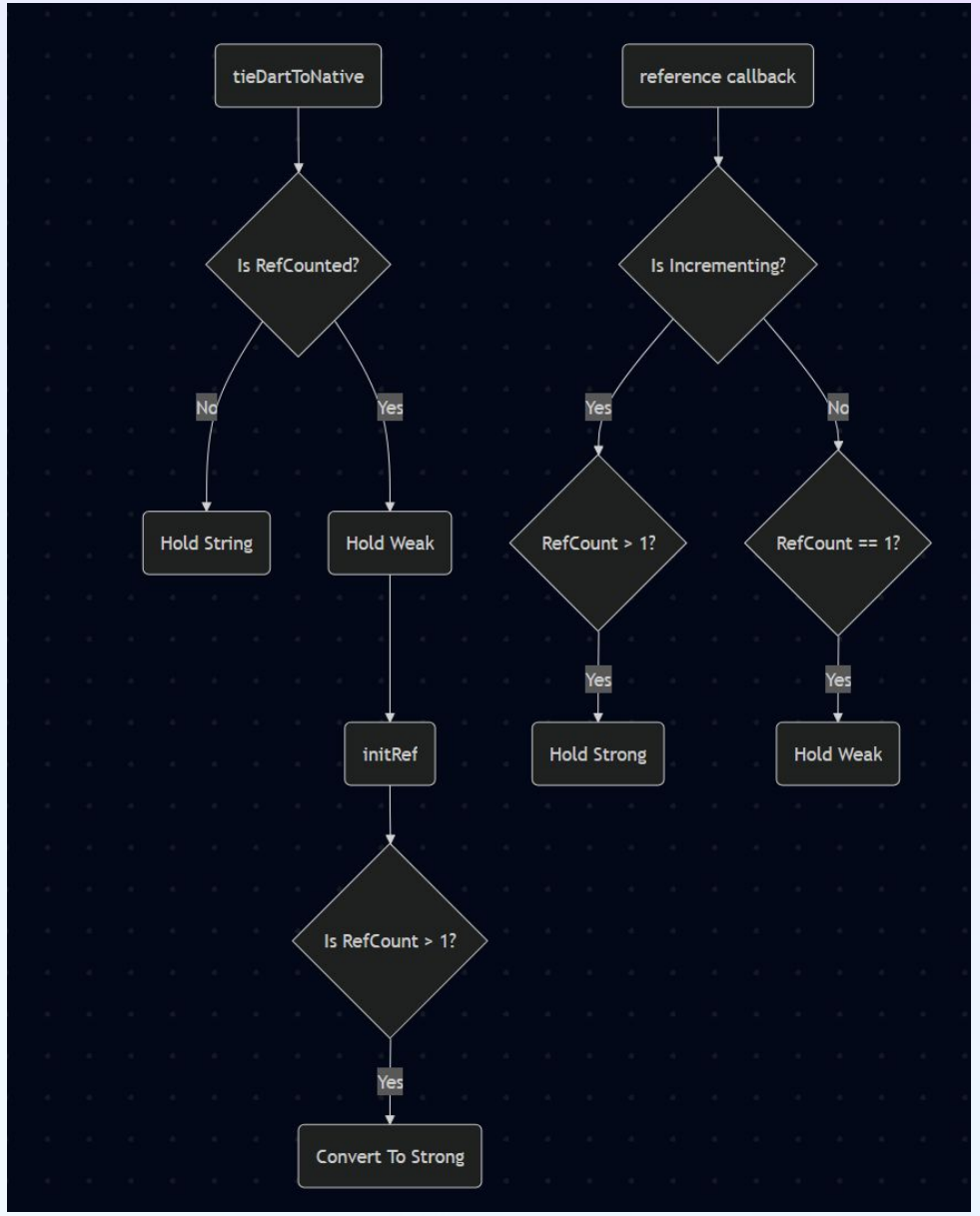
RefCounted Objects and Garbage Collectors

- Both Instance Bindings and Script Instance Info tell you
  - When an object is destroyed
  - When a the reference count on a `RefCounted` changes (<=2 references)

- You are responsible for holding references properly in your language if they are RefCounted or…
  - Your GC might collect bindings you still need
  - They may leak.

Dart’s Process (Copied from C#)

- Make sure the Dart runtime is always holding a strong reference to the Godot Object IF Godot is.
  - Prevent the GC from removing the Dart Object / Instance Binding / Script Instance prematurely
- If Dart is the only thing holding, allow the GC to collect and drop the last reference
  - Prevent memory leaks



Issues with Script Language Extensions

Communication with Addons

- No simple way to know what methods addons offer
- No way to generate bindings against them.
- Addon code needs to be written manually

```
class NetworkTime {
  static NetworkTime? _instance;
  static NetworkTime get instance {
    if (_instance == null) {
      final obj = Engine.singleton
          .getMainLoop()
          ?.as<SceneTree>()
          ?.getRoot()
          ?.getNodeT<GodotObject>('NetworkTime');
      _instance = NetworkTime(obj!);
    }
    return _instance!;
  }

  GodotObject wrapped;

  NetworkTime(this.wrapped);

  Signal get onTick {
    return wrapped.get('on_tick').cast<Signal>()!;
  }
}
```

Support for GDScript Features

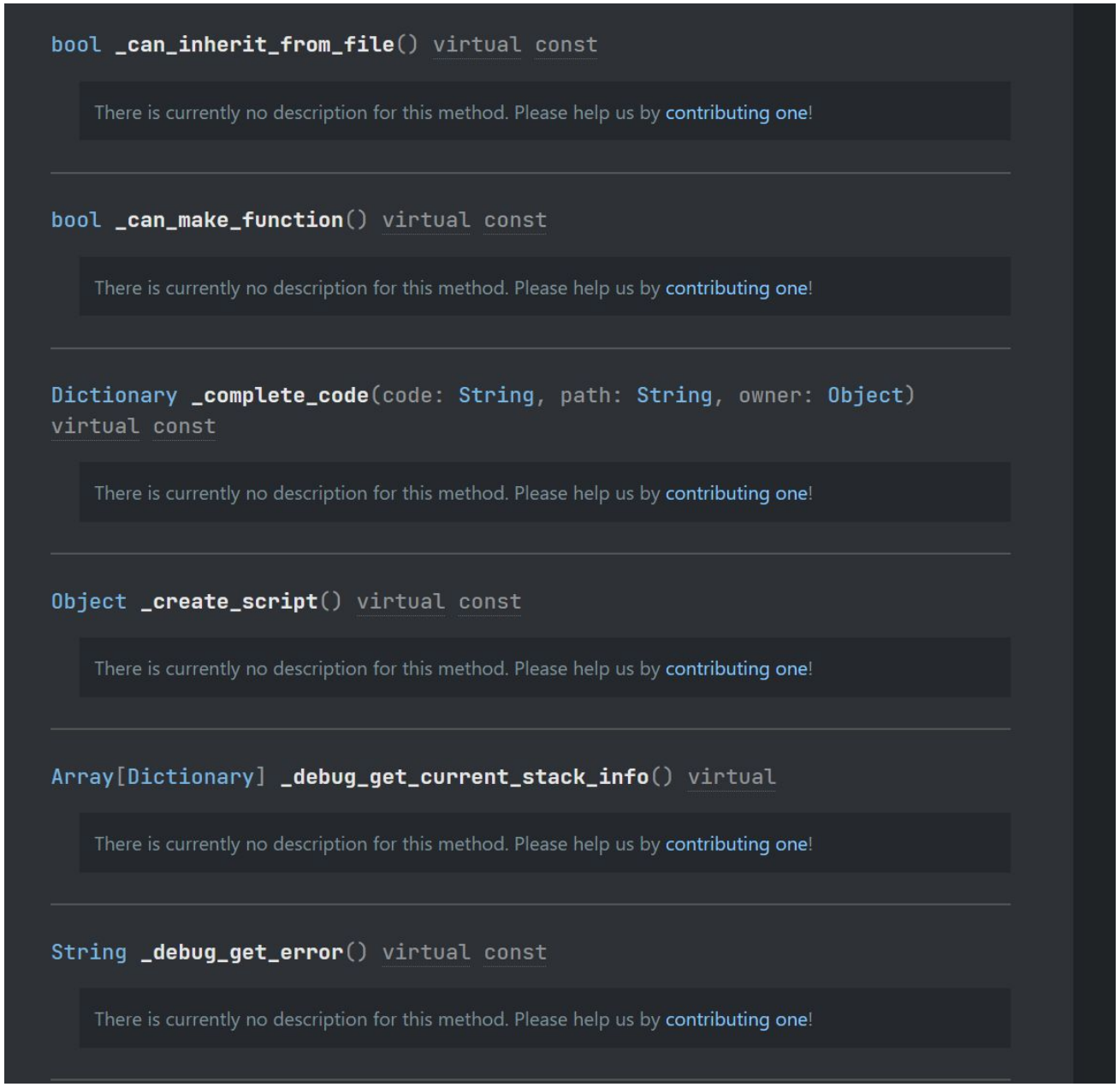
- `@preload` isn’t supported.
- `@rpc` is though.

IDE Support

- Source modification assumes everything goes at the end of the file
- Difficult to get that to work with language servers
  - Info functions are not async
  - Debugging functions are not async

Documentation

- There isn't any
- … I'm going to be starting to work on fixing that…
- … and hopefully you can help!



Thank you


github.com/fuzzybinary


fuzzybinary@mastodon.gamedev.place


@fuzzybinary.bsky.social