Chapter 1

Calling Foreign Code with Spock

Interacting with the external world is a mandatory element of modern systems. Been able to call libraries is often a key success to business. In this chapter, we will explain you how to do that in Pharo using Foreign Function Interface.

Suppose you want to use OpenGL in Pharo. The solution is to make a *binding* of each C library function to a Smalltalk method, and then to call these methods from Smalltalk code. But we should answer several questions: how does a binding call a C function? How do you write such binding? How do we pass heap allocated objects? The libraries and mechanisms in charge of doing this kind of work are usually called FFI, for Foreign Function Interface. In this chapter we will help you to use this interface by exposing some practical examples of how to solve common problems faced when communicating with other languages.

Just to mention, there is another possible approach that would be to invoke C libraries by defining a specific plugin for the virtual machine, but it's more complex and it offers no important advantage compared to the results of using bindings in the same language and with the same constructions you are used to.

We will start to show you how to use Spock , a new foreign function library. Spock is based on NativeBoost and previous FFI libraries. `Luc` ▶*Alien and FFI. cite them here no?*◀

1.1   Getting Started

> **Luc** ▶ *Before the section "calling Ext func", I think we should first introduce the basics (part of the old "What is a FFI and how it works?" section). Because the reader will want to test while reading. And for that he probably will need a special VM (NativeBoost VM), an image with NativeBoost inside, ...* ◀

Right now to run Spock, the new FFI library described in this chapter, you will need a vm that has the NativeBoost plugin and you can find one at https://ci.lille.inria.fr/pharo/job/NB-Cog-Mac-Carbon/.

Then you should load in your image, the package NativeBoost−Installer that you can load as follows:

```
Gofer it
    squeaksource: 'NativeBoost';
    package: 'NativeBoost−Installer';
    load.
```

Once loaded you should execute the following expression: NBInstaller install.

1.2   Calling External a Simple Function

Suppose you want to know the amount of time the image has been running by calling the underlying OS function named clock. This function is part of the standard C library. Its C declaration is[1]:

```
int clock (void)
```

To call clock from the image, you should write a binding: a normal Smalltalk method annotated with the <primitive:module:> pragma. This pragma specifies that a native call should be generated using the Native-BoostPlugin. The native class is described using the message cdecl:module: message. Here is the full definition. We explain the details below.

```
CExamples class>>ticksSinceStart
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ↑ NBFFICallout cdecl: #( int clock () ) module: NativeBoost CLibrary
```

> **Stéf** ▶ *Igor in 1.4 I get a vmVersion -> deprecated.* ◀

You can define this method in any class, on its instance or class side. It doesn't make any difference. In the above example, we defined a class

---

[1] According to man its return type is clock_t instead of int, but we deliberately made it int for the sake of simplicity. We will see how to deal with typedefs in the following sections

named CExamples and define the method on its class side. This way it is easy to access it by sending a message to this class without creating unnecessary instances.

Then, to know the time since the program started you can execute and print it. Note that these are just the native clock ticks, which aren't of much use.

Script 1.1: *Invoking a C binding*

```
CExamples ticksSinceStart
```

Analysis of an FFI Callout

Now that the call worked, let us look at the definition again:

```
CExamples class>>ticksSinceStart
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ↑ NBFFICallout cdecl: #( int clock () ) module: NativeBoost CLibrary
```

Callout. What we have just done is usually called an FFI *callout*. NBFFICallout is the class that defines behavior to perform such calls to external functions. We use the message #cdecl:module: to describe this call.

This message has two arguments: the *signature of the function* that will be invoked and the *module or library* where to look for it. In this example we look this function in the standard C library.

The signature is described by an array: here #(int clock ()).

- Its first element is the C return type, here int.

- Its second is the name of the called function, here clock.

- Its third element is also an array which describes the function parameters when there's any.

About marshaling. The return type is int and not SmallInteger. This is the same with function arguments. They are all described in terms of C language. In general you don't have to worry too much about this because the Spock library does an automatic conversion between C types and Smalltalk types. So, when this message is executed, the return value will be a SmallInteger. This conversion process for types from different languages is called *marshaling*. We will see more examples of automatic conversions in this Chapter.

Presentation Conventions

There are always three things to take into account when doing callouts: a C function, a specification of the binding between C and Smalltalk, and location for such declaration at the Smalltalk level.

1. First, of course, is the signature of the C function we will call, this definition is the definition of the function from a C point of view. Through this book we describe it in a C header box as follows:

   C header.

   ```
   int clock (void)
   ```

2. Second, how do we communicate between the C and Smalltalk worlds. That's the *Pragma declaration*. It describes the C header and 'binds' Smalltalk arguments to C arguments.

   Pragma declaration.

   ```
   CExamples class>>ticksSinceStart
     <primitive: #primitiveNativeCall module: #NativeBoostPlugin>
     ↑ NBFFICallout cdecl: #( int clock () ) module: NativeBoost CLibrary
   ```

3. Third, we have to see how we organize the C calls from the Smalltalk side. Often we create a class that groups callout definitions. Such a class can then be used as an entry point to perform callouts.

   Callout grouping.

   🕐   *CExamples ticksSinceStart*

1.3   Passing arguments to a function

The previous example was the simplest we could find: we asked for the execution of a function without parameters and got the results. Now let's look how we can execute functions that require arguments.

Passing an integer

Let's try with a really simple function, `abs`, which takes an integer and returns its absolute value.

C header.

```
int abs ( int n );
```

Pragma declaration.

```
CExamples class>>abs: anInteger
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ NBFFICallout cdecl: #( int abs (int anInteger) ) module: NativeBoost CLibrary
```

Callout grouping.

      CExamples abs: -20

Compared with the previous example, we changed the name of the function and added an argument. When functions have arguments you have to specify two things for each of them: their type and the object that you want to have sent to the C function. That is, in the arguments array, we put the type of the argument, and after that, the *name of the Smalltalk variable* we pass as argument.

Here anInteger in #(int abs (int anInteger) means that the variable is bound to the abs: method parameter and that will be a C int.

This type-and-name pair will be repeated, separated by comma for each argument, as we will show in the next examples.

Now you can try printing this:

      CExamples abs: -20.

About arguments

In the pragma declaration we are expressing not one but *two* different aspects: the obvious one is the C function signature, the other is which objects we are passing as arguments to the C function when the method is invoked. In this second aspect there are many possibilities. In our example the argument of the C is the method argument: anInteger. But it is not always necessary the case.

You can also put constants, self or any instance variable as arguments to a C call. Suppose you want to add this abs wrapper (let us call it absoluteValue) to the class SmallInteger, so that we can execute −50 absoluteValue.

In that case we simply add the absoluteValue method to SmallInteger, and we directly pass self as illustrated below.

```
SmallInteger>>absoluteValue
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ NBFFICallout cdecl: #( int abs (int self) ) module: NativeBoost CLibrary
```

Imagine that we want to have a wrapper that always calls the absolute function with the number -45. Then we directly define it as follows:

```
CExamples class>>absMinusFortyFive
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ NBFFICallout cdecl: #( int abs (−45) ) module: NativeBoost CLibrary
```

Note that this time we don't put the type of the argument. It is is also possible to pass an instance variable, but we let you do it as an exercise.

Passing strings

As you may know strings in C are sequences of characters terminated with a special character \0. It is then interesting to see how Spock deals with them since they are an important data structure in C. For this, we will call the very well known strlen function. This function requires a string as argument and returns its number of characters.

C header.

```
int strlen ( const char * str );
```

Pragma declaration.

```
CExamples class>>stringLength: aString
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ NBFFICallout cdecl: #( int strlen (String aString) ) module: NativeBoost
    CLibrary
```

Callout grouping.

> ☙ *CExamples stringLength: 'awesome!'*

Example analysis.   You may have noticed that the callout description is not exactly the same as the C function header.

In the signature #(int strlen (String aString)) there are two differences with the C signature.

- The first difference is the const keyword of the argument. For those not used to C, that's only a modifier keyword that the compiler take advantage of to make some static validations at compile time. It's has no value when describing the signature for calling a function at run-time.

- The second difference, an important one, is that specify that the argument is String aString instead of char * aString. With String aString, Spock will automatic do the arguments conversion from Smalltalk strings to C strings (null terminated). Therefore it is important to use String and not char *.

  In the example, the string passed will be put in an external C char array and a null termination character will be added to it. Also, this array will be automatically released after the call ends. Using (String aString) is equivalent to (someString copyWith: (Character value:0) as in CExamples stringLength: (someString copyWith: (Character value:0). Conversely, Spock will take the C result value of calling the C function and convert it to a proper Smalltalk object, an Integer in this particular case.

Passing two strings

To continue with the examples we will call strcmp, which takes two arguments and returns -1, 0 or 1 depending on the relationship between both strings.

C header.

```
int strcmp ( const char * str1, const char * str2 );
```

Pragma declaration.

```
CExamples class>>stringCompare: aString with: anotherString
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ NBFFICallout cdecl: #( int strcmp (String aString, String anotherString) )
    module: NativeBoost CLibrary
```

Callout grouping.

    ☼ *CExamples stringCompare: 'awesome!' with: 'awesome'*

Notice that you can add arguments by appending them to the arguments array, using a comma to separate them. Also notice that you have to explicitly tell which object is going to be sent for each argument, as already told. In this case, aString is the first one and anotherString is the second.

Handling Heap-based Objects: Passing an Array

So far we only used either objects that were allocated on the heap or available on the stack. `Stéf` ▶*is it true with instance variables?*◀ With the next example we will show how Spock supports the access to object that are allocated on the heap. `Stéf` ▶*here*◀ The next example adds a bit more complexity: we are going to send a ByteArray to the memset function. `Stéf` ▶*explain what memset does*◀

C header.

```
void * memset ( void * ptr, int value, int num );
```

Sets the first num bytes of the block of memory pointed by ptr to the specified value

Pragma declaration.

```
CExamples class>>memorySet: aByteArray with: anInteger for: aByteCount
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ NBFFICallout cdecl: #( void* memset (char* aByteArray, int anInteger, int
        aByteCount) ) module: NativeBoost CLibrary
```

Callout grouping.

ⓘ
array := ByteArray new: 10.
CExamples memorySet: array with: 100 for: 5.
array

As you should notice, this time Spock takes care of finding out the memory address where aByteArray lives and sends a pointer to it as the first argument. After executing the example you can see that the byte array elements were actually changed.

Then, there is an important difference between passing arrays and passing strings. The expert user would notice that string functions in C language expects to receive a pointer to contiguous memory bytes ending with a null character to delimit its length. Spock is responsible for copying the string to a bigger buffer and inserting this special end character. This means that strings are passed by-copy, while on the other hand byte arrays are passed by reference!.

Also for expertise users, you may notice that in this last example, as sending a reference, we are sending to a native library a pointer to an object that lives in Smalltalk's memory. We already know that Smalltalk's memory is auto cleaned by a garbage collector that potentially move the objects to other

places, so we have to be careful about this special behaviour. At the moment this book is written the FFI library doesn't support asynchronous calls, so strange behaviour is not possible. But we have to be very careful when facing with C pointer arguments to Smalltalk's memory.

Default primitives types

`Stéf` ▶ *What are the default C primitives type and how do they are mapped in native boost* ◀

float, double, boolean,.....

1.4 Using C structures

1.5 Loading code from other libraries

1.6 Wrapping Cairo library

Cairo is a drawing library written in C. Here we present the basis of writing a wrapper for it, which should be complete enough to write any wrapper for any C library.

Small example

Whenever you want to start writing a wrapper, you should do it this way: find the smallest piece of code you can find that uses that library, and wrap everything you need to make it work from Smalltalk. Then you can go on adding more and more to the wrapped functionality.

In this example we have a minimal program that uses Cairo library, taken from Cairo's FAQ `Javier` ▶ *Add url to it* ◀. It creates a Cairo surface, puts some text on it and renders everything to a PNG.

```
#include <cairo.h>

int
main (int argc, char *argv[])
{
    cairo←surface←t *surface =
      cairo←image←surface←create (CAIRO←FORMAT←ARGB32, 240, 80);
    cairo←t *cr =
      cairo←create (surface);

    cairo←select←font←face (cr, "serif", CAIRO←FONT←SLANT←NORMAL,
      CAIRO←FONT←WEIGHT←BOLD);
```

```
    cairo←set←font←size (cr, 32.0);
    cairo←set←source←rgb (cr, 0.0, 0.0, 1.0);
    cairo←move←to (cr, 10.0, 50.0);
    cairo←show←text (cr, "Hello, world");

    cairo←destroy (cr);
    cairo←surface←write←to←png (surface, "hello.png");
    cairo←surface←destroy (surface);
    return 0;
}
```

The way we will work then is to wrap everything we find in our way from the first to the last line. In order to wrap a library, you should get its reference documentation, so that you know exactly which are its structures, and functions. We start by creating some classes for all Cairo structures that are going to be used here and there in the wrapper.

We create a class for all the library's constants (we will fill them later):

Class 1.2:

```
SharedPool subclass: #NBCairoConstants
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'NBCairo−Core'
```

and a class for all the library's typedefs and enums (we will also fill them later):

Class 1.3:

```
SharedPool subclass: #NBCairoTypes
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'NBCairo−Core'
```

Notice that both of them are Shared Pools. We now create a base class for structures used in the wrapper, putting the shared pools, so that we can use all of them in the derived classes.

Class 1.4:

```
NBExternalObject subclass: #NBCairoHandle
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: 'NBCairoConstants NBCairoTypes'
    category: 'NBCairo−Core'
```

Finally, to wrap the first line of main function we will create a class to represent a Cairo surface.

<div align="center">Class 1.5:</div>

```
NBCairoHandle subclass: #NBCairoSurface
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo−Core'
```

With most well written libraries, the contents of the structures they declare is private to their internals. They are opaque, in the sense that you may not know nor care what bits they have inside, because you only change them through calls to the library. For that kind of structures, Spock has the NBExternalObject. Our wrapping classes inherit from NBExternalObject, which is made for this situation, where don't need to fill the fields of the structs.

Now, NBCairoSurface is described as an abstract structure, there are different concrete ones, like the image surface, so we add it as a subclass.

<div align="center">Class 1.6:</div>

```
NBCairoSurface subclass: #NBCairoImageSurface
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo−Core'
```

Then we can finally create a factory method, to wrap the `cairo_image_surface_create` function. We go to the class side and we enter:

```
NBCairoImageSurface class>>create: aFormat width: aWidth height: aHeight

  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self call: #(NBCairoImageSurface cairo_image_surface_create (int aFormat,
                              int aWidth,
                              int aHeight) )
```

and of course, we shouldn't forget to wrap the code that destroys the instance on the C side, but this time on the instance side of the NBCairoSurface class:

```
NBCairoSurface>>destroy
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self call: #(void cairo_surface_destroy (NBCairoSurface self) )
```

In the creation method, we declare the function as returning a NBCairoImageSurface, but actually it is returning a cairo_surface_t pointer. The same happens on the destroy method which is described as an NBCairo-Surface, but actually is a cairo_surface_t pointer. In both cases, spock takes care of the conversion.

This methods are almost working. To finish, we add some helper functions to the NBCairoHandle class, the one which was going to contain some useful configuration:

```
NBCairoHandle class>>libraryHandle
  ↑NativeBoost forCurrentPlatform loadModule: '/usr/lib/libcairo.so.2'

NBCairoHandle>>libraryHandle
  ↑self class libraryHandle

NBCairoHandle(both instance and class side)>>call: fnSpec
  " you can override this method if you need to"

  | sender |

  sender := thisContext sender.
  ↑ NBFFICallout
    handleFailureIn: sender
    nativeCode: [ :gen |
      gen
        sender: sender;
        cdecl;
        generateCall: fnSpec module: self libraryHandle]
```

Javier ►*we should change something in order to remove this last method*◄

Now to be able to test a bit, we have to add some more things. We add cairo_status_t (which is an enum) to the NBCairoTypes.

<div align="center">Class 1.7:</div>

```
SharedPool subclass: #NBCairoTypes
  instanceVariableNames: ''
  classVariableNames: 'cairo_status_t'
  poolDictionaries: ''
  category: 'NBCairo−Core'
```

```
NBCairoTypes class>>initialize

  "self initialize"

  cairo_status_t := #int
```

and we doit:

🕮 *NBCairoTypes initialize*

.

so that cairo_status_t gets initialized. The, we add a method to the NBCairoSurface class:

```
NBCairoSurface>>status
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self call: #(cairo_status_t cairo_surface_status (NBCairoSurface self) )
```

We are now ready to test just a bit. Take a second and save the image now. Remember that if you wrongly specify the parameters of an FFI call, you could crash the image.

To test that everything is working do print this in a workspace:

🕮 *surface := NBCairoImageSurface new. surface status*

If everything is fine, status should be 0. Then don't forget to destroy the surface:

🕮 *surface destroy*

To make the example complete, let's add the final pieces needed.

Here is how the original example will look after translating it to use the wrapper:

```
CairoExamples>>minimalProgram
  "
  self new minimalProgram
  "

  | surface context fileName |
  surface := NBCairoImageSurface create: CAIRO←FORMAT←ARGB32 width: 240
      height: 80.
  context := surface createContext.
  context selectFont: 'serif' slant: CAIRO←FONT←SLANT←NORMAL weight: CAIRO
      ←FONT←WEIGHT←BOLD;
        setFontSize: 32.0;
        setSourceR: 0.0 G: 0.0 B: 1.0;
        moveToX: 10.0 Y: 50.0;
        showText: 'Hello, world';
        destroy.

  fileName := (FileDirectory default / 'Hello.png') fullName.
  surface writeToPng: fileName.
  surface destroy.
```

```
Form openImageInWindow: fileName
```

but before it works we have to add some constants and types. `CAIRO_FORMAT_ARGB32` is a constant defined on `cairo_format_t`, and we are using constants for font slants and font weights too. For each enum we add a type on NBCairoTypes and its constants to NBCairoConstants:

```
instanceVariableNames: ''
classVariableNames: 'CAIRO←FONT←SLANT←ITALIC CAIRO←FONT←SLANT
    ←NORMAL CAIRO←FONT←SLANT←OBLIQUE CAIRO←FONT←WEIGHT←
    BOLD CAIRO←FONT←WEIGHT←NORMAL CAIRO←FORMAT←A1 CAIRO←
    FORMAT←A8 CAIRO←FORMAT←ARGB32 CAIRO←FORMAT←INVALID
    CAIRO←FORMAT←RGB16←565 CAIRO←FORMAT←RGB24'
poolDictionaries: ''
category: 'NBCairo−Core'

SharedPool subclass: #NBCairoTypes
    instanceVariableNames: ''
    classVariableNames: 'cairo←font←slant←t cairo←font←weight←t cairo←status←t'
    poolDictionaries: ''
    category: 'NBCairo−Core'
```

and then we add the code to initialize these variables:

```
NBCairoTypes class>>initialize

    "self initialize"

    cairo←status←t := cairo←font←slant←t := cairo←font←weight←t := #int


NBCairoConstants class>>initialize

    "enum cairo←format←t"
    CAIRO←FORMAT←INVALID := −1.
    CAIRO←FORMAT←ARGB32 := 0.
    CAIRO←FORMAT←RGB24 := 1.
    CAIRO←FORMAT←A8 := 2.
    CAIRO←FORMAT←A1 := 3.
    CAIRO←FORMAT←RGB16←565 := 4

    "enum cairo←font←slant←t"
    CAIRO←FONT←SLANT←NORMAL := 0.
    CAIRO←FONT←SLANT←ITALIC := 1.
    CAIRO←FONT←SLANT←OBLIQUE := 2

    "enum ←cairo←font←weight←t"
    CAIRO←FONT←WEIGHT←NORMAL := 0.
    CAIRO←FONT←WEIGHT←BOLD := 1.
```

don't forget to ⓘ *NBCairoConstants initialize*

and ⓘ *NBCairoTypes initialize*

.

As a last step, you need to wrap the remaining functions. We leave here a last example and the rest of them you can take the code from the NBCairo repository on squeaksource.

<center>Class 1.8:</center>

```
NBCairoSurface subclass: #NBCairoImageSurface
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo−Core'
```

```
CairoContext>>selectFont: aFamily slant: slant weight: aWeight
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self call: #(void cairo←select←font←face (NBCairoContext self,
                                String aFamily,
                                cairo←font←slant←t slant,
                                cairo←font←weight←t aWeight) )
```

Original FFI stuff

Thanks to a Foreign Function Interface (FFI), it is possible to achieve this and interact with third-party libraries. Multiple FFI libraries are available in Pharo such as FFI (http://wiki.squeak.org/squeak/1414) and Alien (http://www.squeaksource.com/Alien). As an example the dynamic `OpenGL` library (.dll, .so or .dylib depending on the operating system) is used in Croquet through the FFI library and AlienOpenGL (http://www.squeaksource.com/AlienOpenGL) use it through the Alien library.

In this chapter, we will describe what is a FFI library and how it works with the virtual machine. We will then dive into the Alien library and learn how it can be used thanks to the AlienOpenGL example.

1.7   What is a FFI and how it works?

The Pharo image is loaded and executed by a virtual machine (VM) which is itself executed on the top of the operating system. Two different virtual machines can be used for Pharo images: the squeak VM and the Cog VM. Regarding the general FFI mechanism, it is not really relevant to make a distinction[2] Both are based on a plugin architecture. A plugin enable the Smalltalk code (in the image) to access third-party functionalities (outside the image and the VM).

The Alien library uses the *IA32ABI* plugin which ....

1.8   Using the Alien Library

Installation

Alien is a Squeaksource project http://www.squeaksource.com/Alien. Use the following code to install Alien in a Pharo 1.1 image:

```
Gofer new
  url: 'http://www.squeaksource.com/Alien' ;
  package: 'ConfigurationOfAlien';
  load.

(Smalltalk at: #ConfigurationOfAlien) loadCore ;
  loadTests ;
  loadLibC.
```

---

[2]It worth noting that the Cog VM is younger and really faster than the Squeak VM but it still presents some small limitations and bugs at the time writing this book.
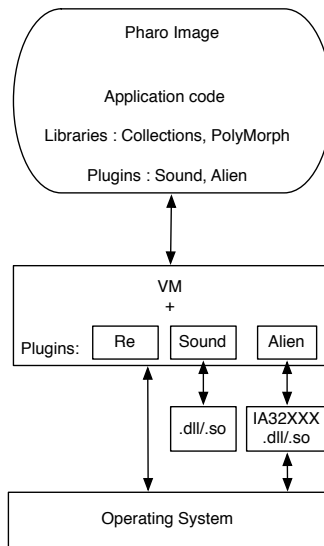
Figure 1.1: Pharo Architecture

Run the tests to ensure that Alien is well installed and that the VM plugin is correctly working.

First example

Core Classes

Figure **??** shows an UML diagram of the Alien-Core package that contains the main classes of this library.

Memory management

Calling an external C library from Smalltalk

Callbacks from C to Smalltalk code

1.9 Dissection of Alien
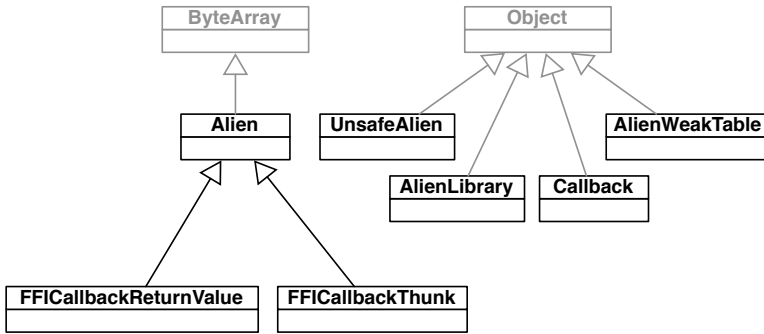
Alien

FFICallbackReturnValue

Figure 1.2: Core classes of the Alien library

FFICallbackThunk

UnsafeAlien

AlienLibrary

Callback

AlienWeakTable

1.10 Study Case: the AlienOpenGL package