Kotlin/Native

We'll cover the following

In addition to JVM, JavaScript, Android devices, iOS, and Raspberry Pi, you can target your Kotlin code to native platforms like Windows, Mac OS, and Linux. The Kotlin compiler's front end compiles your Kotlin code to an intermediate representation. Kotlin/Native is an LLVM-based back end of the compiler that targets the intermediate representation to self-contained programs that can run natively without a virtual machine. Your Kotlin code can interoperate with native libraries—for example, code written using the C language, Objective-C, and Swift. You can use this option to create fast running native applications and to interoperate with native libraries.

In this appendix, we'll take a look at a small example that illustrates compiling Kotlin code to target MacOS. To target the example to other operating systems, refer to the toolchain requirements on the Kotlin/Native website.

Downloading native environment

To compile Kotlin code to native platform, you need the Kotlin/Native compiler. Download it from the binary distribution website. Once you install it, add the `bin` directory that contains the tools like `cinterop` and `kotlinc-native` to the path.

To get a glimpse of the power of Kotlin/Native, let's create a small function in C and then call it from within Kotlin code. As a first step, let's create a C header file named `fib.h` under a `c` directory.

```
// fib.h
int fib(int n);
```

The header file declares a function named `fib()` that takes an integer value and

returns an integer value.

Next, let's implement this function in a file named `fib.c`, also placed in the `c` directory:

```
#include "fib.h"

int fib(int n) {
  return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
}
```

fib.c

The function computes, using recursion, the Fibonacci number for the given parameter. We'll need a C compiler to compile this code and create a library. You may use any C compiler that you're familiar with. Here, we'll use the GCC GNU compiler and then the ar archival tool to bundle the object file created by the compiler into a library archive file.

```
gcc -c c/fib.c
ar crv libfib.a fib.o
```

The `gcc` command as shown above will compile the C code that is in the file `fib.c` into an object file named `fib.o`. Then the `ar` command will bundle that file into an archive file named `libfib.a`.

Using `cinterop`

Next, we'll use this library from within Kotlin code. To invoke the `int fib(int);` C function from within Kotlin, we need a Kotlin representation `fun fib(n: Int): Int`. Creating this manually can be tedious and error prone when working with any nontrivial libraries. The `cinterop` tool, which is part of the Kotlin/Native distribution, removes that burden by autogenerating the function signature from the C header files.

To assist the `cinterop` tool, we need to create an interop `def` that provides the necessary information, like the name and location of the header files, library file to link with, and so on. You may also include declarations and implementations of C functions directly in the `def` file after a triple dash separator line (—). That's a way to try interop without the need to have a C compiler.

Let's create the `def` file, also under the c directory:

```
// interop.def
headers = fib.h
linkerOpts.osx = -L. -lfib
```

The `headers` property lists the header files for the `cinterop` tool to process. The `linkerOpts` property lists the location and the name of the archive file—the `-l` option mentions the archive file name without the `lib` prefix and the `.a` suffix.

Let's run the `cinterop` tool to generate the Kotlin signature for the C function:

```
cinterop -def c/interop.def -compilerOpts -I./c -o libfib
```

This command instructs the `cinterop` tool to process the `interop.def` file. The `-I` option provides the location of the header file mentioned within the `def` file. The `-o` option specifies the name of the Kotlin linking file to generate.

After running the command, examine the files generated. The Kotlin signature file for the function resides in the generated file `libfib-build/kotlin/interop/interop.kt`. We'll import this file from within our Kotlin code to call the `fib()` C function.

Here's the sample Kotlin code to call the `fib()` function, written in a file named `sample.kt` under a new kotlin directory.

```
import interop.*

fun main() {
  println("fib(10) is ${fib(10)}")
}
```

sample.kt

We'll use the `kotlinc-native` compiler to compile this Kotlin source code to a native target.

```
kotlinc-native kotlin/sample.kt -library libfib -o sample
```

The previous command generates a native executable file named `sample.kexe`. Run the executable to watch Kotlin code call the C function:

```
./sample.kexe
```

```
fib(10) is 89
```

This example illustrates both the Kotlin-C interop and how Kotlin code can be targeted to a native platform and can be run as a native executable.