Scala Native is an optimizing ahead-of-time compiler and lightweight managed runtime designed specifically for Scala. It features:

- **Low-level primitives.**

```scala
type Vec = CStruct3[Double, Double, Double]
val vec = stackalloc[Vec] // allocate c struct on stack
vec._1 = 10.0  // initialize fields
vec._2 = 20.0
vec._3 = 30.0
length(vec) // pass by reference
```

Pointers, structs, you name it. Low-level primitives let you hand-tune your application to make it work exactly as you want it to. You’re in control.

- **Seamless interop with native code.**

```scala
@extern object stdlib {
  def malloc(size: CSize): Ptr[Byte] = extern
}
val ptr = stdlib.malloc(32)
```

Calling C code has never been easier. With the help of extern objects you can seamlessly call native code without any runtime overhead.

- **Instant startup time.**

```scala
> time hello-native
hello, native!
real 0m0.005s
user 0m0.002s
sys 0m0.002s
```

Scala Native is compiled ahead-of-time via LLVM. This means that there is no sluggish warm-up phase that’s common for just-in-time compilers. Your code is immediately fast and ready for action.
CHAPTER 1

Community

- Want to follow project updates? Follow us on twitter.
- Want to chat? Join our Gitter chat channel.
- Have a question? Ask it on Stack Overflow with tag scala-native.
- Found a bug or want to propose a new feature? Open an issue on Github.
This documentation is divided into different parts. It’s recommended to go through the User’s Guide to get familiar with Scala Native. Libraries will walk you through all the known libraries that are currently available. Contributor’s Guide contains valuable information for people who want to either contribute to the project or learn more about the internals and the development process behind the project.

2.1 User’s Guide

2.1.1 Environment setup

Scala Native has the following build dependencies:

- Java 8 or newer
- sbt 0.13.13 or newer
- LLVM/Clang 3.7 or newer

And following completely optional runtime library dependencies:

- Boehm GC 7.6.0 (optional)
- zlib 1.2.8 or newer (optional)

These are only required if you use the corresponding feature.

Installing sbt

Please refer to this link for instructions for your operating system.
Installing clang and runtime dependencies

Scala Native requires Clang, which is part of the LLVM toolchain. The recommended LLVM version is 3.7 or newer, however, the Scala Native sbt plugin uses feature detection to discover the installed version of Clang so older versions may also work.

Scala Native uses the inmix garbage collector by default. You can use the Boehm garbage collector instead, as described here. If you chose to use the Boehm garbage collector both the native library and header files must be provided at build time.

If you use classes from the `java.util.zip` for compression zlib needs to be installed.

**Note:** Some package managers provide the library header files in separate `-dev` packages.

Here are install instructions for a number of operating systems Scala Native has been used with:

**macOS**

$ brew install llvm
$ brew install bdw-gc # optional

*Note:* A version of zlib that is sufficiently recent comes with the installation of macOS.

**Ubuntu**

$ sudo apt install clang
$ sudo apt install libgc-dev # optional

**Arch Linux**

$ sudo pacman -S llvm clang
$ sudo pacman -S gc # optional

*Note:* A version of zlib that is sufficiently recent comes with the installation of Arch Linux.

**Fedora 26**

$ sudo dnf install llvm clang
$ sudo dnf install gc-devel zlib-devel # both optional

**FreeBSD**

$ pkg install llvm38
$ pkg install boehm-gc # optional

*Note:* A version of zlib that is sufficiently recent comes with the installation of FreeBSD.

**Nix/NixOS**

$ nix-shell scala-native.nix -A clangEnv

Continue to *Building projects with sbt*.

### 2.1.2 Building projects with sbt

If you have reached this section you probably have a system that is now able to compile and run Scala Native programs.
Minimal sbt project

The easiest way to make a fresh project is to use our official gitter8 template:

\[ \texttt{sbt new scala-native/scala-native.g8} \]

This generates the following files:

- \texttt{project/plugins.sbt} to add a plugin dependency:
  \[ \texttt{addSbtPlugin("org.scala-native" \% "sbt-scala-native" \% "0.4.0-M2")} \]

- \texttt{project/build.properties} to specify the sbt version:
  \[ \texttt{sbt.version = 1.2.6} \]

- \texttt{build.sbt} to enable the plugin and specify Scala version:
  \[ \texttt{enablePlugins(ScalaNativePlugin)} \]
  \[ \texttt{scalaVersion := "2.11.12"} \]

- \texttt{src/main/scala/Main.scala} with minimal application:
  \[ \texttt{object Main { \}
  \texttt{  def main(args: Array[String]): Unit = \}
  \texttt{     println("Hello, world!"))} \]

Now, simply run \texttt{sbt run} to get everything compiled and have the expected output! Please refer to the FAQ if you encounter any problems.

Scala versions

Scala Native supports following Scala versions for corresponding releases:

<table>
<thead>
<tr>
<th>Scala Native Version</th>
<th>Scala Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1.x</td>
<td>2.11.8</td>
</tr>
<tr>
<td>0.2.x</td>
<td>2.11.8, 2.11.11</td>
</tr>
<tr>
<td>0.3.0-0.3.3</td>
<td>2.11.8, 2.11.11</td>
</tr>
<tr>
<td>0.3.4+, 0.4.0-M1, 0.4.0-M2</td>
<td>2.11.8, 2.11.11, 2.11.12</td>
</tr>
</tbody>
</table>
Sbt settings and tasks

<table>
<thead>
<tr>
<th>Since</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>compile</td>
<td>Analysis</td>
<td>Compile Scala code to NIR</td>
</tr>
<tr>
<td>0.1</td>
<td>run</td>
<td>Unit</td>
<td>Compile, link and run the generated binary</td>
</tr>
<tr>
<td>0.1</td>
<td>package</td>
<td>File</td>
<td>Similar to standard package with addition of NIR</td>
</tr>
<tr>
<td>0.1</td>
<td>publish</td>
<td>Unit</td>
<td>Similar to standard publish with addition of NIR (1)</td>
</tr>
<tr>
<td>0.1</td>
<td>nativeLink</td>
<td>File</td>
<td>Link NIR and generate native binary</td>
</tr>
<tr>
<td>0.1</td>
<td>nativeClang</td>
<td>File</td>
<td>Path to clang command</td>
</tr>
<tr>
<td>0.1</td>
<td>nativeCompileOptions</td>
<td>Seq[String]</td>
<td>Extra options passed to clang verbatim during compilation</td>
</tr>
<tr>
<td>0.1</td>
<td>nativeLinkingOptions</td>
<td>Seq[String]</td>
<td>Extra options passed to clang verbatim during linking</td>
</tr>
<tr>
<td>0.1</td>
<td>nativeMode</td>
<td>String</td>
<td>One of &quot;debug&quot;, &quot;release-fast&quot; or &quot;release-full&quot; (2)</td>
</tr>
<tr>
<td>0.2</td>
<td>nativeGC</td>
<td>String</td>
<td>One of &quot;none&quot;, &quot;boehm&quot; or &quot;immix&quot; (3)</td>
</tr>
<tr>
<td>0.3.3</td>
<td>nativeLinkStubs</td>
<td>Boolean</td>
<td>Whether to link @stub definitions, or to ignore them</td>
</tr>
<tr>
<td>0.4.0</td>
<td>nativeLTO</td>
<td>String</td>
<td>One of &quot;none&quot;, &quot;full&quot; or &quot;thin&quot; (4)</td>
</tr>
<tr>
<td>0.4.0</td>
<td>nativeCheck</td>
<td>Boolean</td>
<td>Shall the linker check intermediate results for correctness?</td>
</tr>
<tr>
<td>0.4.0</td>
<td>nativeDump</td>
<td>Boolean</td>
<td>Shall the linker dump intermediate results to disk?</td>
</tr>
</tbody>
</table>

1. See Publishing and Cross compilation for details.
2. See Compilation modes for details.
3. See Garbage collectors for details.
4. See Link-Time Optimization (LTO) for details.

Compilation modes

Scala Native supports three distinct linking modes:

1. **debug.** (default)
   
   Default mode. Optimized for shortest compilation time. Runs fewer optimizations and is much more suited for iterative development workflow. Similar to clang’s `−O0`.

2. **release.** (deprecated since 0.4.0)
   
   Aliases to **release-full**.

3. **release-fast.** (introduced in 0.4.0)
   
   Optimize for runtime performance while still trying to keep quick compilation time and small emitted code size. Similar to clang’s `−O2` with addition of link-time optimization over the whole application code.

4. **release-full.** (introduced in 0.4.0)
   
   Optimized for best runtime performance, even if hurts compilation time and code size. This modes includes a number of more aggressive optimizations such type-driven method duplication and more aggressive inliner. Similar to clang’s `−O3` with addition of link-time optimization over the whole application code.

Garbage collectors

1. **immix.** (default since 0.3.8, introduced in 0.3)
   
   Immix is a mostly-precise mark-region tracing garbage collector. More information about the collector is available as part of the original 0.3.0 announcement.
2. **boehm.** (default through 0.3.7)

Conservative generational garbage collector. More information is available at the [project’s page](#).

3. **none.** (experimental, introduced in 0.2)

Garbage collector that allocates things without ever freeing them. Useful for short-running command-line applications or applications where garbage collections pauses are not acceptable.

### Link-Time Optimization (LTO)

Scala Native relies on link-time optimization to maximize runtime performance of release builds. There are three possible modes that are currently supported:

1. **none.** (default)

   Does not inline across Scala/C boundary. Scala to Scala calls are still optimized.

2. **full.** (available on Clang 3.8 or older)

   Inlines across Scala/C boundary using legacy FullLTO mode.

3. **thin.** (recommended on Clang 3.9 or newer)

   Inlines across Scala/C boundary using LLVM’s latest ThinLTO mode. Offers both better compilation speed and better runtime performance of the generated code than the legacy FullLTO mode.

### Publishing

Scala Native supports sbt’s standard workflow for the package distribution:

1. Compile your code.
2. Generate a jar with all of the class files and NIR files.
3. Publish the jar to [sonatype](#), [bintray](#) or any other 3rd party hosting service.

Once the jar has been published, it can be resolved through sbt’s standard package resolution system.

### Cross compilation

`sbt-crossproject` is an sbt plugin that lets you cross-compile your projects against all three major platforms in Scala: JVM, JavaScript via Scala.js, and native via Scala Native. It is based on the original cross-project idea from Scala.js and supports the same syntax for existing JVM/JavaScript cross-projects. Please refer to the project’s [README](#) for details.

Continue to [Language semantics](#).

### 2.1.3 Language semantics

In general, the semantics of the Scala Native language are the same as Scala on the JVM. However, a few differences exist, which we mention here.

### Interop extensions

Annotations and types defined `scala.scalanative.unsafe` may modify semantics of the language for sake of interoperability with C libraries, read more about those in [Native code interoperability](#) section.
Multithreading

Scala Native doesn’t yet provide libraries for parallel multi-threaded programming and assumes single-threaded execution by default.

It’s possible to use C libraries to get access to multi-threading and synchronization primitives but this is not officially supported at the moment.

Finalization

Finalize method from java.lang.Object is never called in Scala Native.

Undefined behavior

Generally, Scala Native follows most of the special error conditions similarly to JVM:

1. Arrays throw IndexOutOfBoundsException on out-of-bounds access.
2. Casts throw ClassCastException on incorrect casts.
3. Accessing a field or method on null, throwing null exception, throws NullPointerException.
4. Integer division by zero throws ArithmeticException.

There are a few exceptions:

1. Stack overflows are undefined behavior and would typically segfault on supported architectures instead of throwing StackOverflowError.
2. Exhausting a heap space results in crash with a stack trace instead of throwing OutOfMemoryError.

Continue to Native code interoperability.

2.1.4 Native code interoperability

Scala Native provides an interop layer that makes it easy to interact with foreign native code. This includes C and other languages that can expose APIs via C ABI (e.g. C++, D, Rust etc.)

All of the interop APIs discussed here are defined in scala.scalanative.unsafe package. For brevity, we’re going to refer to that namespace as just unsafe.

Extern objects

Extern objects are simple wrapper objects that demarcate scopes where methods are treated as their native C ABI-friendly counterparts. They are roughly analogous to header files with top-level function declarations in C.

For example, to call C’s malloc one might declare it as following:

```scala
import scala.scalanative.unsafe._

@extern
object libc {
  def malloc(size: CSize): Ptr[Byte] = extern
}
```
the right hand side of the method definition signifies that the body of the method is defined elsewhere in a native library that is available on the library path (see Linking with native libraries). The signature of the external function must match the signature of the original C function (see Finding the right signature).

Finding the right signature

To find a correct signature for a given C function one must provide an equivalent Scala type for each of the arguments:

<table>
<thead>
<tr>
<th>C Type</th>
<th>Scala Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>Unit</td>
</tr>
<tr>
<td>bool</td>
<td>unsafe.CBool</td>
</tr>
<tr>
<td>char</td>
<td>unsafe.CChar</td>
</tr>
<tr>
<td>signed char</td>
<td>unsafe.CSignedChar</td>
</tr>
<tr>
<td>unsigned char</td>
<td>unsafe.CUnsignedChar [I]</td>
</tr>
<tr>
<td>short</td>
<td>unsafe.CShort</td>
</tr>
<tr>
<td>unsigned short</td>
<td>unsafe.CUnsignedShort [I]</td>
</tr>
<tr>
<td>int</td>
<td>unsafe.CInt</td>
</tr>
<tr>
<td>long int</td>
<td>unsafe.CLongInt</td>
</tr>
<tr>
<td>unsigned int</td>
<td>unsafe.CUnsignedInt [I]</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>unsafe.CUnsignedLongInt [I]</td>
</tr>
<tr>
<td>long</td>
<td>unsafe.CLong</td>
</tr>
<tr>
<td>unsigned long</td>
<td>unsafe.CUnsignedLong [I]</td>
</tr>
<tr>
<td>long long</td>
<td>unsafe.CUnsignedLongLong [I]</td>
</tr>
<tr>
<td>size_t</td>
<td>unsafe.CSize</td>
</tr>
<tr>
<td>ptrdiff_t</td>
<td>unsafe.CPtrDiff [2]</td>
</tr>
<tr>
<td>wchar_t</td>
<td>unsafe.CWideChar</td>
</tr>
<tr>
<td>char16_t</td>
<td>unsafe.CChar16</td>
</tr>
<tr>
<td>char32_t</td>
<td>unsafe.CChar32</td>
</tr>
<tr>
<td>float</td>
<td>unsafe.CFloat</td>
</tr>
<tr>
<td>double</td>
<td>unsafe.CDouble</td>
</tr>
<tr>
<td>void*</td>
<td>unsafe.Ptr[Byte] [2]</td>
</tr>
<tr>
<td>int*</td>
<td>unsafe.Ptr[unsafe.CInt] [2]</td>
</tr>
<tr>
<td>char*</td>
<td>unsafe.CString</td>
</tr>
<tr>
<td>int (*)(int)</td>
<td>unsafe.CFuncPtr1[unsafe.CInt, unsafe.CInt] [2] [4]</td>
</tr>
<tr>
<td>struct { int x, y; }*</td>
<td>unsafe.Ptr[unsafe.CStruct2[unsafe.CInt, unsafe.CInt]] [2] [5]</td>
</tr>
</tbody>
</table>

Linking with native libraries

C compilers typically require to pass an additional -l mylib flag to dynamically link with a library. In Scala Native, one can annotate libraries to link with using the @link annotation.

import scala.scalanative.unsafe._
@link("mylib")
@extern
object mylib {

(continues on next page)
Whenever any of the members of `mylib` object are reachable, the Scala Native linker will automatically link with the corresponding native library.

As in C, library names are specified without the `lib` prefix. For example, the library `libuv` corresponds to `@link("uv")` in Scala Native.

It is possible to rename functions using the `@name` annotation. Its use is recommended to enforce the Scala naming conventions in bindings:

```scala
import scala.scalanative.unsafe._
@link("uv")
@extern
object uv {
    @name("uv_uptime")
    def uptime(result: Ptr[CDouble]): Int = extern
}
```

If a library has multiple components, you could split the bindings into separate objects as it is permitted to use the same `@link` annotation more than once.

**Variadic functions**

Scala Native supports native interoperability with C’s variadic argument list type (i.e. `va_list`), but not ... `varargs`. For example `vprintf` can be declared as:

```scala
import scala.scalanative.unsafe._
@extern
object mystdio {
    def vprintf(format: CString, args: CVarArgList): CInt = extern
}
```

One can wrap a function in a nicer API like:

```scala
import scala.scalanative.unsafe._
def myprintf(format: CString, args: CVarArg*): CInt =
    Zone {
        implicit z =>
            mystdio.vprintf(format, toCVarArgList(args.toSeq))
    }
```

And then call it just like a regular Scala function:

```scala
myprintf(c"2 + 3 = %d, 4 + 5 = %d", 2 + 3, 4 + 5)
```

**Pointer types**

Scala Native provides a built-in equivalent of C’s pointers via `unsafe.Ptr[T]` data type. Under the hood pointers are implemented using unmanaged machine pointers.

Operations on pointers are closely related to their C counterparts and are compiled into equivalent machine code:
<table>
<thead>
<tr>
<th>Operation</th>
<th>C syntax</th>
<th>Scala Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load value</td>
<td>*ptr</td>
<td>!ptr</td>
</tr>
<tr>
<td>Store value</td>
<td>*= value</td>
<td>!ptr = value</td>
</tr>
<tr>
<td>Pointer to index</td>
<td>ptr + i, &amp;ptr[i]</td>
<td>ptr + i</td>
</tr>
<tr>
<td>Elements between</td>
<td>ptr1 - ptr2</td>
<td>ptr1 - ptr2</td>
</tr>
<tr>
<td>Load at index</td>
<td>ptr[i]</td>
<td>ptr(i)</td>
</tr>
<tr>
<td>Store at index</td>
<td>ptr[i] = value</td>
<td>ptr(i) = value</td>
</tr>
<tr>
<td>Pointer to field</td>
<td>&amp;ptr-&gt;name</td>
<td>ptr.atN</td>
</tr>
<tr>
<td>Load a field</td>
<td>ptr-&gt;name</td>
<td>ptr._N</td>
</tr>
<tr>
<td>Store a field</td>
<td>ptr-&gt;name = value</td>
<td>ptr._N = value</td>
</tr>
</tbody>
</table>

Where N is the index of the field name in the struct. See Memory layout types for details.

### Function pointers

It is possible to use external functions that take function pointers. For example given the following signature in C:

```c
void test(void (* f)(char *));
```

One can declare it as following in Scala Native:

```scala
def test(f: unsafe.CFuncPtr1[CString, Unit]): Unit = unsafe.extern
```

`CFuncPtrN` types are a SAM (single abstract method) traits. You can define them by creating a class that inherits from the corresponding trait:

```scala
val myfuncptr = new unsafe.FuncPtr0[Unit] { 
  def apply(): Unit = println("hi there!")
}
```

On Scala 2.12 or newer, Scala language automatically coverts from closures to SAM types:

```scala
val myfuncptr: unsafe.FuncPtr0[Unit] = () => println("hi there!")
```

### Memory management

Unlike standard Scala objects that are managed automatically by the underlying runtime system, one has to be extra careful when working with unmanaged memory.

1. **Zone allocation.** (since 0.3)

   Zones (also known as memory regions/contexts) are a technique for semi-automatic memory management. Using them one can bind allocations to a temporary scope in the program and the zone allocator will automatically clean them up for you as soon as execution goes out of it:

```scala
import scala.scalanative.unsafe._

Zone { implicit z => 
  val buffer = alloc[Byte](n)
}
```

`alloc` requests memory sufficient to contain n values of a given type. If number of elements is not specified, it defaults to a single element. Memory is zeroed out by default.
Zone allocation is the preferred way to allocate temporary unmanaged memory. It’s idiomatic to use implicit zone parameters to abstract over code that has to zone allocate.

One typical example of this are C strings that are created from Scala strings using `unsafe.toCString`. The conversion takes implicit zone parameter and allocates the result in that zone.

When using zone allocated memory one has to be careful not to capture this memory beyond the lifetime of the zone. Dereferencing zone-allocated memory after the end of the zone is undefined behavior.

2. **Stack allocation.**

Scala Native provides a built-in way to perform stack allocations of using `unsafe.stackalloc` function:

```scala
val buffer = unsafe.stackalloc[Byte](256)
```

This code will allocate 256 bytes that are going to be available until the enclosing method returns. Number of elements to be allocated is optional and defaults to 1 otherwise. Memory is not zeroed out by default.

When using stack allocated memory one has to be careful not to capture this memory beyond the lifetime of the method. Dereferencing stack allocated memory after the method’s execution has completed is undefined behavior.

3. **Manual heap allocation.**

Scala Native’s library contains a bindings for a subset of the standard libc functionality. This includes the trio of `malloc`, `realloc` and `free` functions that are defined in `unsafe.stdlib` extern object.

Calling those will let you allocate memory using the system’s standard dynamic memory allocator. Every single manual allocation must also be freed manually as soon as it’s not needed any longer.

Apart from the standard system allocator one might also bind to plethora of 3-rd party allocators such as `jemalloc` to serve the same purpose.

**Undefined behavior**

Similarly to their C counter-parts, behavior of operations that access memory is subject to undefined behaviour for following conditions:

1. Dereferencing null.
2. Out-of-bounds memory access.
3. Use-after-free.
4. Use-after-return.
5. Double-free, invalid free.

**Memory layout types**

Memory layout types are auxiliary types that let one specify memory layout of unmanaged memory. They are meant to be used purely in combination with native pointers and do not have a corresponding first-class values backing them.

- `unsafe.Ptr[unsafe.CStructN[T1, ..., TN]]`

  Pointer to a C struct with up to 22 fields. Type parameters are the types of corresponding fields. One may access fields of the struct using `_N` helper methods on a pointer value:
```scala
val ptr = unsafe.stackalloc[unsafe.CStruct2[Int, Int]]
ptr._1 = 10
ptr._2 = 20
println(s"first $ptr._1, second $ptr._2")
```

Here _N is an accessor for the field number N.

- `unsafe.Ptr[unsafe.CArray[T, N]]`

Pointer to a C array with statically-known length N. Length is encoded as a type-level natural number. Natural numbers are types that are composed of base naturals `Nat._0, ... Nat._9` and an additional `Nat.DigitN` constructors, where N refers to number of digits in the given number. So for example number 1024 is going to be encoded as following:

```scala
import scalanative.unsafe._, Nat._
type _1024 = Digit4[_1, _0, _2, _4]
```

Once you have a natural for the length, it can be used as an array length:

```scala
val arrptr = unsafe.stackalloc[CArray[Byte, _1024]]
```

You can find an address of n-th array element via `arrptr.at(n)`.

### Byte strings

Scala Native supports byte strings via `c"..."` string interpolator that gets compiled down to pointers to statically-allocated zero-terminated strings (similarly to C):

```scala
import scalanative.unsafe._
import scalanative.libc._

// CString is an alias for Ptr[CChar]
val msg: CString = c"Hello, world!"
stdio.printf(msg)
```

Additionally, we also expose two helper functions `unsafe.toCString` and `unsafe.fromCString` to convert between C-style and Java-style strings.

### Platform-specific types

Scala Native defines the type `Word` and its unsigned counterpart, `UWord`. A word corresponds to `Int` on 32-bit architectures and to `Long` on 64-bit ones.

### Size and alignment of types

In order to statically determine the size of a type, you can use the `sizeof` function which is Scala Native’s counterpart of the eponymous C operator. It returns the size in bytes:

```scala
println(unsafe.sizeof[Byte])  // 1
println(unsafe.sizeof[Bool]) // 1
println(unsafe.sizeof[Short]) // 2
println(unsafe.sizeof[Int])  // 4
println(unsafe.sizeof[Long]) // 8
```
It can also be used to obtain the size of a structure:

```scala
Type TwoBytes = unsafe.CStruct2[Byte, Byte]
println(unsafe.sizeof[TwoBytes])   // 2
```

Additionally you can also use `alignmentof` to find alignment of a given type:

```scala
println(unsafe.alignmentof[Int])   // 4
println(unsafe.alignmentof[unsafe.CStruct2[Byte, Long]])   // 8
```

**Unsigned integer types**

Scala Native provides support for four unsigned integer types:

1. unsigned.UByte
2. unsigned.USHort
3. unsigned.UInt
4. unsigned.ULong

They share the same primitive operations as signed integer types. Primitive operation between two integer values are supported only if they have the same signedness (they must both signed or both unsigned.)

Conversions between signed and unsigned integers must be done explicitly using `byteValue.toUByte, shortValue.toUSHort, intValue.toUInt, longValue.toULong` and conversely `unsignedByteValue.toByte, unsignedShortValue.toShort, unsignedIntValue.toInt, unsignedLongValue.toLong`.

Continue to Libraries.

**2.2 Libraries**

**2.2.1 Java Standard Library**

Scala Native supports a subset of the JDK core libraries reimplemented in Scala.

**Supported classes**

Here is the list of currently available classes:

- java.io.BufferedInputStream
- java.io.BufferedOutputStream
- java.io.BufferedReader
- java.io.BufferedWriter
- java.io.ByteArrayInputStream
- java.io.ByteArrayOutputStream
- java.io.Closeable
- java.io.DataInput
- java.io.DataInputStream
• java.io.DataOutput
• java.io.DataOutputStream
• java.io.EOFException
• java.io.File
• java.io.FileInputStream
• java.io.FileNotFoundException
• java.io.FileOutputStream
• java.io.FileReader
• java.io.FileWriter
• java.io.FilenameFilter
• java.io.FilterInputStream
• java.io.FilterOutputStream
• java.io.FilterReader
• java.io.Flushable
• java.io.IOException
• java.io.InputStream
• java.io.InputStreamReader
• java.io.InterruptedIOException
• java.io.LineNumberReader
• java.io.NotSerializableException
• java.io.ObjectStreamException
• java.io.OutputStream
• java.io.OutputStreamWriter
• java.io.PrintStream
• java.io.PrintWriter
• java.io.PushbackInputStream
• java.io.PushbackReader
• java.io.RandomAccessFile
• java.io.Reader
• java.io.Serializable
• java.io.StringReader
• java.io.StringWriter
• java.io.SyncFailedException
• java.io.UTFDataFormatException
• java.io.UnsupportedEncodingException
• java.io.Writer
• java.lang.AbstractMethodError
• java.lang.AbstractStringBuilder
• java.lang.Appendable
• java.lang.ArithmeticException
• java.lang.ArrayIndexOutOfBoundsException
• java.lang.ArrayStoreException
• java.lang.AssertionError
• java.lang.AutoCloseable
• java.lang.Boolean
• java.lang.BootstrapMethodError
• java.lang.Byte
• java.lang.ByteCache
• java.lang.CharSequence
• java.lang.Character
• java.lang.Character$CaseFolding
• java.lang.Character$Subset
• java.lang.Character$UnicodeBlock
• java.lang.CharacterCache
• java.lang.ClassCastException
• java.lang.ClassCircularityError
• java.lang.ClassCastException
• java.lang.ClassLoader
• java.lang.ClassNotFoundException
• java.lang.CloneNotSupportedException
• java.lang.Cloneable
• java.lang.Comparable
• java.lang.Double
• java.lang.Enum
• java.lang.EnumConstantNotPresentException
• java.lang.Error
• java.lang.Exception
• java.lang.ExceptionInInitializerError
• java.lang.Float
• java.lang.IllegalAccessError
• java.lang.IllegalArgumentException
• java.lang.IllegalStateException
• java.lang.IllegalThreadStateException
• java.lang.IncompatibleClassChangeError
• java.lang.IndexOutOfBoundsException
• java.lang.InheritableThreadLocal
• java.lang.InstantiationException
• java.lang.InstantiationError
• java.lang.Integer
• java.lang.IntegerCache
• java.lang.IntegerDecimalScale
• java.lang.InternalError
• java.lang.InterruptedException
• java.lang.Iterable
• java.lang.LinkageError
• java.lang.Long
• java.lang.LongCache
• java.lang.Math
• java.lang.MathRand
• java.lang.NegativeArraySizeException
• java.lang.NoClassDefFoundError
• java.lang.NoSuchFieldError
• java.lang.NoSuchFieldException
• java.lang.NoSuchMethodError
• java.lang.NoSuchMethodException
• java.lang.NullPointerException
• java.lang.Number
• java.lang.NumberFormatException
• java.lang.OutOfMemoryError
• java.lang.PipeIO
• java.lang.PipeIO$NullInput
• java.lang.PipeIO$NullOutput
• java.lang.PipeIO$Stream
• java.lang.PipeIO$Stream$class
• java.lang.PipeIO$StreamImpl
• java.lang.Process
• java.lang.ProcessBuilder
• java.lang.ProcessBuilder$Redirect
• java.lang.ProcessBuilder$Redirect$RedirectImpl
• java.lang.ProcessBuilder$Redirect$RedirectImpl$RedirectImpl
• java.lang.ProcessBuilder$Redirect$Type
• java.lang.Readable
• java.lang.ReflectiveOperationException
• java.lang.RejectedExecutionException
• java.langRunnable
• java.lang.Runtime
• java.lang.Runtime$ProcessBuilderOps
• java.lang.RuntimeException
• java.lang/securityException
• java.lang.Short
• java.lang.ShortCache
• java.lang.StackOverflowError
• java.lang.StackTrace
• java.lang.StackTraceElement
• java.lang.StackTraceElement$Fail
• java.lang.String
• java.lang.StringBuffer
• java.lang.StringBuilder
• java.lang.StringIndexOutOfBoundsException
• java.lang.System
• java.lang.Thread
• java.lang.Thread$UncaughtExceptionHandler
• java.lang.ThreadDeath
• java.lang.ThreadLocal
• java.lang.Throwable
• java.lang.TypeNotPresentException
• java.lang.UnixProcess
• java.lang.UnixProcess$ProcessMonitor
• java.lang.UnknownError
• java.lang.UnsatisfiedLinkError
• java.lang.UnsupportedClassVersionError
• `java.lang.UnsupportedOperationException`  
• `java.lang.VerifyError`  
• `java.lang.VirtualMachineError`  
• `java.lang.Void`  
• `java.lang._String$CaseInsensitiveComparator`  
• `java.lang.annotation.Retention`  
• `java.lang.annotation.RetentionPolicy`  
• `java.lang.ref.PhantomReference`  
• `java.lang.ref.Reference`  
• `java.lang.ref.ReferenceQueue`  
• `java.lang.ref.SoftReference`  
• `java.lang.ref.WeakReference`  
• `java.lang.reflect.AccessibleObject`  
• `java.lang.reflect.Array`  
• `java.lang.reflect.Constructor`  
• `java.lang.reflect.Executable`  
• `java.lang.reflect.Field`  
• `java.lang.reflect.InvocationTargetException`  
• `java.lang.reflect.Method`  
• `java.lang.reflect.UndeclaredThrowableException`  
• `java.math.BigDecimal`  
• `java.math.BigDecimal$QuotAndRem`  
• `java.math.BigDecimal$StringOps`  
• `java.math.BigInteger`  
• `java.math.BigInteger$QuotAndRem`  
• `java.math.BitLevel`  
• `java.math.Conversion`  
• `java.math.Division`  
• `java.math.Elementary`  
• `java.math.Logical`  
• `java.math.MathContext`  
• `java.math.Multiplication`  
• `java.math.Primality`  
• `java.math.RoundingMode`  
• `java.net.BindException`  
• `java.net.ConnectException`
• java.net.Inet4Address
• java.net.Inet6Address
• java.net.InetAddress
• java.net.InetAddressBase
• java.net.InetAddressBase$class
• java.net.InetSocketAddress
• java.net.MalformedURLException
• java.net.NoRouteToHostException
• java.net.PlainSocketImpl
• java.net.PortUnreachableException
• java.net.ServerSocket
• java.net.Socket
• java.net.SocketAddress
• java.net.SocketException
• java.net.SocketImpl
• java.net.SocketInputStream
• java.net.SocketOption
• java.net.SocketOptions
• java.net.SocketOutputStream
• java.net.SocketTimeoutException
• java.net.URI
• java.net.URI$Helper
• java.net.URIEncoderDecoder
• java.net.URISyntaxException
• java.net.URL
• java.net.URLClassLoader
• java.net.URLConnection
• java.net.URLDecoder
• java.net.UnknownHostException
• java.net.UnknownServiceException
• java.nio.Buffer
• java.nio.BufferOverflowException
• java.nio.BufferUnderflowException
• java.nio.ByteArrayBits
• java.nio.ByteBuffer
• java.nio.ByteOrder
• java.nio.CharBuffer
• java.nio.DoubleBuffer
• java.nio.FloatBuffer
• java.nio.GenBuffer
• java.nio.GenHeapBuffer
• java.nio.GenHeapBuffer$NewHeapBuffer
• java.nio.GenHeapBufferView
• java.nio.GenHeapBufferView$NewHeapBufferView
• java.nio.HeapByteBuffer
• java.nio.HeapByteBuffer$NewHeapByteBuffer
• java.nio.HeapByteBufferCharView
• java.nio.HeapByteBufferCharView$NewHeapByteBufferCharView
• java.nio.HeapByteBufferDoubleView
• java.nio.HeapByteBufferDoubleView$NewHeapByteBufferDoubleView
• java.nio.HeapByteBufferFloatView
• java.nio.HeapByteBufferFloatView$NewHeapByteBufferFloatView
• java.nio.HeapByteBufferIntView
• java.nio.HeapByteBufferIntView$NewHeapByteBufferIntView
• java.nio.HeapByteBufferLongView
• java.nio.HeapByteBufferLongView$NewHeapByteBufferLongView
• java.nio.HeapByteBufferShortView
• java.nio.HeapByteBufferShortView$NewHeapByteBufferShortView
• java.nio.HeapCharBuffer
• java.nio.HeapCharBuffer$NewHeapCharBuffer
• java.nio.HeapDoubleBuffer
• java.nio.HeapDoubleBuffer$NewHeapDoubleBuffer
• java.nio.HeapFloatBuffer
• java.nio.HeapFloatBuffer$NewHeapFloatBuffer
• java.nio.HeapIntBuffer
• java.nio.HeapIntBuffer$NewHeapIntBuffer
• java.nio.HeapLongBuffer
• java.nio.HeapLongBuffer$NewHeapLongBuffer
• java.nio.HeapShortBuffer
• java.nio.HeapShortBuffer$NewHeapShortBuffer
• java.nio.IntBuffer
• java.nio.InvalidMarkException
• java.nio.LongBuffer
• java.nio.MappedByteBuffer
• java.nio.ReadOnlyBufferException
• java.nio.ShortBuffer
• java.nio.StringCharBuffer
• java.nio.channels.ByteChannel
• java.nio.channels.Channel
• java.nio.channels.Channels
• java.nio.channels.ClosedChannelException
• java.nio.channels.FileChannel
• java.nio.channels.FileChannel$MapMode
• java.nio.channels.FileChannelImpl
• java.nio.channels.FileLock
• java.nio.channels.GatheringByteChannel
• java.nio.channels.InterruptibleChannel
• java.nio.channels.NonReadableChannelException
• java.nio.channels.NonWritableChannelException
• java.nio.channels.OverlappingFileLockException
• java.nio.channels.ReadableByteChannel
• java.nio.channels.ScatteringByteChannel
• java.nio.channels.SeekableByteChannel
• java.nio.channels.WritableByteChannel
• java.nio.channels.spi.AbstractInterruptibleChannel
• java.nio.charset.CharacterCodingException
• java.nio.charset.Charset
• java.nio.charset.CharsetDecoder
• java.nio.charset.CharsetEncoder
• java.nio.charset.CoderMalfunctionError
• java.nio.charset.CoderResult
• java.nio.charset.CodingErrorAction
• java.nio.charset.IllegalArgumentException
• java.nio.charset.MalformedInputException
• java.nio.charset.StandardCharsets
• java.nio.charset.UnmappableCharacterException
• java.nio.charset.UnsupportedCharsetException
• java.nio.file.AccessDeniedException
• `java.nio.file.CopyOption`
• `java.nio.file.DirectoryIteratorException`
• `java.nio.file.DirectoryNotEmptyException`
• `java.nio.file.DirectoryStream`
• `java.nio.file.DirectoryStream$Filter`
• `java.nio.file.DirectoryStreamImpl`
• `java.nio.file.FileAlreadyExistsException`
• `java.nio.file.FileSystem`
• `java.nio.file.FileSystemException`
• `java.nio.file.FileSystemLoopException`
• `java.nio.file.FileSystemNotFoundException`
• `java.nio.file.FileSystems`
• `java.nio.file.FileVisitOption`
• `java.nio.file.FileVisitResult`
• `java.nio.file.FileVisitor`
• `java.nio.file.Files`
• `java.nio.file.Files$TerminateTraversalException`
• `java.nio.file.LinkOption`
• `java.nio.file.NoSuchFileException`
• `java.nio.file.NotDirectoryException`
• `java.nio.file.NotLinkException`
• `java.nio.file.OpenOption`
• `java.nio.file.Path`
• `java.nio.file.PathMatcher`
• `java.nio.file.PathMatcherImpl`
• `java.nio.file.Paths`
• `java.nio.file.RegexPathMatcher`
• `java.nio.file.SimpleFileVisitor`
• `java.nio.file.StandardCopyOption`
• `java.nio.file.StandardOpenOption`
• `java.nio.file.StandardWatchEventKinds`
• `java.nio.file.WatchEvent`
• `java.nio.file.WatchEvent$Kind`
• `java.nio.file.WatchEvent$Modifier`
• `java.nio.file.WatchKey`
• `java.nio.file.WatchService`

2.2. Libraries 25
• java.nio.file.Watchable
• java.nio.file.attribute.AclEntry
• java.nio.file.attribute.AclFileAttributeView
• java.nio.file.attribute.AttributeView
• java.nio.file.attribute.BasicFileAttributeView
• java.nio.file.attribute.BasicFileAttributes
• java.nio.file.attribute.DosFileAttributeView
• java.nio.file.attribute.DosFileAttributes
• java.nio.file.attribute.FileAttribute
• java.nio.file.attribute.FileAttributeView
• java.nio.file.attribute.FileAttributeView$class
• java.nio.file.attribute.FileOwnerAttributeView
• java.nio.file.attribute.FileStoreAttributeView
• java.nio.file.attribute.FileTime
• java.nio.file.attribute.GroupPrincipal
• java.nio.file.attribute.PosixFileAttributeView
• java.nio.file.attribute.PosixFileAttributes
• java.nio.file.attribute.PosixFilePermission
• java.nio.file.attribute.PosixFilePermissions
• java.nio.file.attribute.UserDefinedFileAttributeView
• java.nio.file.attribute.UserPrincipal
• java.nio.file.attribute.UserPrincipalLookupService
• java.nio.file.spi.FileSystemProvider
• java.rmi.Remote
• java.rmi.RemoteException
• java.security.AccessControlException
• java.security.CodeSigner
• java.security.DummyMessageDigest
• java.security.GeneralSecurityException
• java.security.MessageDigest
• java.security.MessageDigestSpi
• java.security.NoSuchAlgorithmException
• java.security.Principal
• java.security.Timestamp
• java.security.TimestampConstructorHelper
• java.security.cert.CertPath
• java.security.cert.Certificate
• java.security.cert.CertificateEncodingException
• java.security.cert.CertificateException
• java.security.cert.CertificateFactory
• java.security.cert.X509Certificate
• java.security.cert.X509Extension
• java.text.DateFormatSymbols
• java.text.DecimalFormat
• java.text.DecimalFormat$BigDecimalFormatting
• java.text.DecimalFormat$BigIntegerFormatting
• java.text.DecimalFormat$DoubleFormatting
• java.text.DecimalFormat$DoubleFormatting$DoubleDigits
• java.text.DecimalFormat$Formatting
• java.text.DecimalFormat$Formatting$Digits
• java.text.DecimalFormat$Formatting$class
• java.text.DecimalFormat$LongFormatting
• java.text.DecimalFormat$PatternSyntax
• java.text.DecimalFormat$PatternSyntax$Affix
• java.text.DecimalFormat$PatternSyntax$Exponent
• java.text.DecimalFormat$PatternSyntax$Fraction
• java.text.DecimalFormat$PatternSyntax$Fraction$$plus$plus
• java.text.DecimalFormat$PatternSyntax$Integer
• java.text.DecimalFormat$PatternSyntax$MinimumExponent
• java.text.DecimalFormat$PatternSyntax$MinimumFraction
• java.text.DecimalFormat$PatternSyntax$MinimumInteger
• java.text.DecimalFormat$PatternSyntax$Number
• java.text.DecimalFormat$PatternSyntax$Number$Fraction$_plus$plus
• java.text.DecimalFormat$PatternSyntax$Number$Integer$_plus$plus
• java.text.DecimalFormat$PatternSyntax$OptionalFraction
• java.text.DecimalFormat$PatternSyntax$Pattern
• java.text.DecimalFormat$PatternSyntax$Pattern$$plus$plus
• java.text.DecimalFormat$PatternSyntax$SignedPattern
• java.text.DecimalFormat$PatternSyntax$SignedPattern$Number$_plus$plus
• java.text.DecimalFormat$PatternSyntax$SignedPattern$Prefix$_plus$plus
• java.text.DecimalFormatSymbols
• java.text.FieldPosition

2.2. Libraries
• java.text.Format
• java.text.Format$Field
• java.text.NumberFormat
• java.text.ParseException
• java.time.Duration
• java.time.Instant
• java.time.temporal.TemporalAmount
• java.util.AbstractCollection
• java.util.AbstractList
• java.util.AbstractList$View
• java.util.AbstractMap
• java.util.AbstractMap$SimpleEntry
• java.util.AbstractMap$SimpleImmutableEntry
• java.util.AbstractQueue
• java.util.AbstractRandomAccessListIterator
• java.util.AbstractSequentialList
• java.util.AbstractSet
• java.util.ArrayList
• java.util.Arrays
• java.util.Arrays$AsRef
• java.util.BackedUpListIterator
• java.util.Base64
• java.util.Base64$Decoder
• java.util.Base64$DecodingInputStream
• java.util.Base64$Encoder
• java.util.Base64$EncodingOutputStream
• java.util.Base64$Wrapper
• java.util.Calendar
• java.util.Collection
• java.util.Collections
• java.util.Collections$BasicSynchronizedList$1
• java.util.Collections$CheckedCollection
• java.util.Collections$CheckedList
• java.util.Collections$CheckedListIterator
• java.util.Collections$CheckedMap
• java.util.Collections$CheckedSet
- `java.util.Collections$CheckedSortedMap`
- `java.util.Collections$CheckedSortedSet`
- `java.util.Collections$EmptyIterator`
- `java.util.Collections$EmptyListIterator`
- `java.util.Collections$ImmutableList`
- `java.util.Collections$ImmutableMap`
- `java.util.Collections$ImmutableSet`
- `java.util.Collections$UnmodifiableCollection`
- `java.util.Collections$UnmodifiableIterator`
- `java.util.Collections$UnmodifiableList`
- `java.util.Collections$UnmodifiableListIterator`
- `java.util.Collections$UnmodifiableMap`
- `java.util.Collections$UnmodifiableSet`
- `java.util.Collections$UnmodifiableSortedMap`
- `java.util.Collections$UnmodifiableSortedSet`
- `java.util.Collections$WrappedCollection`
- `java.util.Collections$WrappedCollection$class`
- `java.util.Collections$WrappedEquals`
- `java.util.Collections$WrappedEquals$class`
- `java.util.Collections$WrappedIterator`
- `java.util.Collections$WrappedIterator$class`
- `java.util.Collections$WrappedList`
- `java.util.Collections$WrappedList$class`
- `java.util.Collections$WrappedListIterator`
- `java.util.Collections$WrappedListIterator$class`
- `java.util.Collections$WrappedMap`
- `java.util.Collections$WrappedMap$class`
- `java.util.Collections$WrappedSet`
- `java.util.Collections$WrappedSortedMap`
- `java.util.Collections$WrappedSortedMap$class`
- `java.util.Collections$WrappedSortedSet`
- `java.util.Collections$WrappedSortedSet$class`
- `java.util.Comparator`
- `java.util.Comparator$class`
- `java.util.ConcurrentModificationException`
- `java.util.Date`
• java.util.Deque
• java.util.Dictionary
• java.util.DuplicateFormatFlagsException
• java.util.EmptyStackException
• java.util.EnumSet
• java.util.Enumeration
• java.util.FormatFlagsConversionMismatchException
• java.util.Formattable
• java.util.FormattableFlags
• java.util.Formatter
• java.util.Formatter$BigDecimalLayoutForm
• java.util.Formatter$DateTimeUtil
• java.util.Formatter$FloatUtil
• java.util.Formatter$FormatToken
• java.util.Formatter$ParserStateMachine
• java.util.Formatter$Transformer
• java.util.FormatterClosedException
• java.util.GregorianCalendar
• java.util.HashMap
• java.util.HashMap$AbstractMapView
• java.util.HashMap$AbstractMapView$class
• java.util.HashMap$AbstractMapViewIterator
• java.util.HashMap$EntrySet
• java.util.HashMap$KeySet
• java.util.HashMap$ValuesView
• java.util.HashSet
• java.util.Hashtable
• java.util.Hashtable$UnboxedEntry$1
• java.util.IdentityHashMap
• java.util.IllegalFormatCodePointException
• java.util.IllegalFormatConversionException
• java.util.IllegalFormatException
• java.util.IllegalFormatFlagsException
• java.util.IllegalFormatPrecisionException
• java.util.IllegalFormatWidthException
• java.util.IllformedLocaleException
• java.util.InputMismatchException
• java.util.InvalidPropertiesFormatException
• java.util.Iterator
• java.util.LinkedHashMap
• java.util.LinkedHashSet
• java.util.LinkedList
• java.util.LinkedList$Node
• java.util.List
• java.util.ListIterator
• java.util.Locale
• java.util.Map
• java.util.Map$Entry
• java.util.MissingFormatArgumentException
• java.util.MissingFormatWidthException
• java.util.MissingResourceException
• java.utilNavigableSet
• java.utilNavigableView
• java.util.NoSuchElementException
• java.util.Objects
• java.util.PriorityQueue
• java.util.PriorityQueue$BoxOrdering
• java.util.Properties
• java.util.Queue
• java.util.Random
• java.util.RandomAccess
• java.util.RandomAccessListIterator
• java.util.ServiceConfigurationError
• java.util.Set
• java.util.SizeChangeEvent
• java.util.SizeChangeEvent$class
• java.util.SortedMap
• java.util.SortedSet
• java.util.StringTokenizer
• java.util.TimeZone
• java.util.TooManyListenersException
• java.util.TreeSet
• java.util.TreeSet$BoxOrdering
• java.util.UUID
• java.util.UnknownFormatConversionException
• java.util.UnknownFormatFlagsException
• java.util.WeakHashMap
• java.util.WeakHashMap$AbstractMapView
• java.util.WeakHashMap$AbstractMapView$class
• java.util.WeakHashMap$AbstractMapViewIterator
• java.util.WeakHashMap$EntrySet
• java.util.WeakHashMap$KeySet
• java.util.WeakHashMap$ValuesView
• java.util.concurrent.Callable
• java.util.concurrent.CancellationException
• java.util.concurrent.ExecutionException
• java.util.concurrent.Executor
• java.util.concurrent.RejectedExecutionException
• java.util.concurrent.TimeUnit
• java.util.concurrent.TimeoutException
• java.util.concurrent.atomic.AtomicBoolean
• java.util.concurrent.atomic.AtomicInteger
• java.util.concurrent.atomic.AtomicLong
• java.util.concurrent.atomic.AtomicLongArray
• java.util.concurrent.atomic.AtomicReference
• java.util.concurrent.atomic.AtomicReferenceArray
• java.util.concurrent.locks.AbstractOwnableSynchronizer
• java.util.concurrent.locks.AbstractQueuedSynchronizer
• java.util.concurrent.locks.AbstractQueuedSynchronizer
• java.util.function.BiConsumer
• java.util.function.BiConsumer$class
• java.util.function.BiFunction
• java.util.function.BiFunction$class
• java.util.function.BiPredicate
• java.util.function.BiPredicate$class
• java.util.function.BinaryOperator
• java.util.function.Consumer
• java.util.function.Consumer$class
• java.util.function.Function
• java.util.function.Function$Class
• java.util.function.Predicate
• java.util.function.Predicate$Class
• java.util.function.Supplier
• java.util.function.UnaryOperator
• java.util.jar.Attributes
• java.util.jar.Attributes$Name
• java.util.jar.InitManifest
• java.util.jar.JarEntry
• java.util.jar.JarFile
• java.util.jar.JarFile$JarFileEnumerator$1
• java.util.jar.JarFile$JarFileInputStream
• java.util.jar.JarInputStream
• java.util.jar.JarOutputStream
• java.util.jar.JarVerifier
• java.util.jar.JarVerifier$VerifierEntry
• java.util.jar.Manifest
• java.util.jar.Manifest$Chunk
• java.util.package
• java.util.package$Box
• java.util.package$CompareNullablesOps
• java.util.package$IdentityBox
• java.util.regex.MatchResult
• java.util.regex.Matcher
• java.util.regex.Pattern
• java.util.regex.PatternSyntaxException
• java.util.stream.BaseStream
• java.util.stream.CompositeStream
• java.util.stream.EmptyIterator
• java.util.stream.Stream
• java.util.stream.Stream$Builder
• java.util.stream.WrappedScalaStream
• java.util.stream.WrappedScalaStream$Builder
• java.util.zip.Adler32
• java.util.zip.CRC32
Scala Native implements `java.util.regex`-compatible API using Google’s RE2 library. RE2 is not a drop-in replacement for `java.util.regex` but handles most common cases well.

Some notes on the implementation:

1. The included RE2 implements a Unicode version lower than the version used in the Scala Native Character class (>= 7.0.0). The RE2 Unicode version is in the 6.n range. For reference, Java 8 released with Unicode 6.2.0.

2. This implementation of RE2 does not support:
   - Character classes:
   - Unions: [a-d[m-p]]
   - Intersections: [a-z&[^aeiou]]
   - Predefined character classes: \h, \H, \v, \\v
   - Patterns:

   The RE2 implemented may not match codepoints added or changed in later Unicode versions. Similarly, there may be slight differences for Unicode codepoints with high numeric value between values used by RE2 and those used by the Character class.
• Octal: \0100 - use decimal or hexadecimal instead.
• Two character Hexadecimal: \xFF - use \x00FF instead.
• All alphabetic Unicode: \uBEEF - use hex \x00BEEF instead.
• Escape: \e - use \u001B instead.
• Java character function classes:
  • \p{javaLowerCase}
  • \p{javaUpperCase}
  • \p{javaWhitespace}
  • \p{javaMirrored}
• Boundary matchers: \G, \R, \Z
• Possessive quantifiers: X?+, X*+, X{n}+, X{n,}+, X{n,m}+
• Lookaheads: (?=X), (?!X), (?<=X), (?<!X), (?>X)
• Options
  • CANON_EQ
  • COMMENTS
  • LITERAL
  • UNICODE_CASE
  • UNICODE_CHARACTER_CLASS
  • UNIX_LINES
• Patterns to match a Unicode binary property, such as \p{isAlphabetic} for a codepoint with the "alphabetic" property, are not supported. Often another pattern \p{isAlpha} may be used instead, \p{isAlpha} in this case.

3. The following Matcher methods have a minimal implementation:
• Matcher.hasAnchoringBounds() - always return true.
• Matcher.hasTransparentBounds() - always throws UnsupportedOperationException because RE2 does not support lookaheads.
• Matcher.hitEnd() - always throws UnsupportedOperationException.
• Matcher.region(int, int)
• Matcher.regionEnd()
• Matcher.regionStart()
• Matcher.requireEnd() - always throws UnsupportedOperationException.
• Matcher.useAnchoringBounds(boolean) - always throws UnsupportedOperationException
• Matcher.useTransparentBounds(boolean) - always throws UnsupportedOperationException because RE2 does not support lookaheads.

4. Scala Native 0.3.8 required POSIX patterns to have the form [[:alpha:]]. Now the Java standard form \p{Alpha} is accepted and the former variant pattern is not. This improves compatibility with Java but, regrettably, may require code changes when upgrading from Scala Native 0.3.8.

Continue to C Standard Library.
2.2.2 C Standard Library

Scala Native provides bindings for the core subset of the C standard library:

<table>
<thead>
<tr>
<th>C Header</th>
<th>Scala Native Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert.h</td>
<td>N/A</td>
</tr>
<tr>
<td>complex.h</td>
<td>scala.scalanative.libc.complex</td>
</tr>
<tr>
<td>ctype.h</td>
<td>scala.scalanative.libc.ctype</td>
</tr>
<tr>
<td>errno.h</td>
<td>scala.scalanative.libc.errno</td>
</tr>
<tr>
<td>fenv.h</td>
<td>N/A</td>
</tr>
<tr>
<td>float.h</td>
<td>N/A</td>
</tr>
<tr>
<td>inttypes.h</td>
<td>N/A</td>
</tr>
<tr>
<td>iso646.h</td>
<td>N/A</td>
</tr>
<tr>
<td>limits.h</td>
<td>N/A</td>
</tr>
<tr>
<td>locale.h</td>
<td>N/A</td>
</tr>
<tr>
<td>math.h</td>
<td>scala.scalanative.libc.math</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>N/A</td>
</tr>
<tr>
<td>signal.h</td>
<td>scala.scalanative.libc.signal</td>
</tr>
<tr>
<td>stdalign.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdarg.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdatomic.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdbool.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stddef.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdint.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdio.h</td>
<td>scala.scalanative.libc.stdio</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>scala.scalanative.libcstdlib</td>
</tr>
<tr>
<td>stdnoreturn.h</td>
<td>N/A</td>
</tr>
<tr>
<td>string.h</td>
<td>scala.scalanative.libc.string</td>
</tr>
<tr>
<td>tgmath.h</td>
<td>N/A</td>
</tr>
<tr>
<td>threads.h</td>
<td>N/A</td>
</tr>
<tr>
<td>time.h</td>
<td>N/A</td>
</tr>
<tr>
<td>uchar.h</td>
<td>N/A</td>
</tr>
<tr>
<td>wchar.h</td>
<td>N/A</td>
</tr>
<tr>
<td>wctype.h</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Continue to C POSIX Library.

2.2.3 C POSIX Library

Scala Native provides bindings for the core subset of the POSIX library:

<table>
<thead>
<tr>
<th>C Header</th>
<th>Scala Native Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>aio.h</td>
<td>N/A</td>
</tr>
<tr>
<td>arpa/inet.h</td>
<td>scala.scalanative.posixarpa.inet</td>
</tr>
<tr>
<td>assert.h</td>
<td>N/A</td>
</tr>
<tr>
<td>complex.h</td>
<td>scala.scalanative.libc.complex</td>
</tr>
<tr>
<td>cpio.h</td>
<td>scala.scalanative.posix.cpio</td>
</tr>
<tr>
<td>ctype.h</td>
<td>N/A</td>
</tr>
<tr>
<td>dirent.h</td>
<td>scala.scalanative.posix.dirent</td>
</tr>
<tr>
<td>dlfcn.h</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>C Header</th>
<th>Scala Native Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>errno.h</td>
<td>scala.scalanative.posix.errno</td>
</tr>
<tr>
<td>fcntl.h</td>
<td>scala.scalanative.posix.fcntl</td>
</tr>
<tr>
<td>fenv.h</td>
<td>N/A</td>
</tr>
<tr>
<td>float.h</td>
<td>N/A</td>
</tr>
<tr>
<td>fmtmsg.h</td>
<td>N/A</td>
</tr>
<tr>
<td>fnmatch.h</td>
<td>N/A</td>
</tr>
<tr>
<td>ftw.h</td>
<td>N/A</td>
</tr>
<tr>
<td>getopt.h</td>
<td>scala.scalanative.posix.getopt</td>
</tr>
<tr>
<td>glob.h</td>
<td>N/A</td>
</tr>
<tr>
<td>grp.h</td>
<td>scala.scalanative.posix.grp</td>
</tr>
<tr>
<td>iconv.h</td>
<td>N/A</td>
</tr>
<tr>
<td>inttypes.h</td>
<td>scala.scalanative.posix.inttypes</td>
</tr>
<tr>
<td>iso646.h</td>
<td>N/A</td>
</tr>
<tr>
<td>langinfo.h</td>
<td>N/A</td>
</tr>
<tr>
<td>libgen.h</td>
<td>N/A</td>
</tr>
<tr>
<td>limits.h</td>
<td>scala.scalanative.posix.limits</td>
</tr>
<tr>
<td>locale.h</td>
<td>N/A</td>
</tr>
<tr>
<td>math.h</td>
<td>N/A</td>
</tr>
<tr>
<td>monetary.h</td>
<td>N/A</td>
</tr>
<tr>
<td>mqueue.h</td>
<td>N/A</td>
</tr>
<tr>
<td>ndbm.h</td>
<td>N/A</td>
</tr>
<tr>
<td>netif.h</td>
<td>N/A</td>
</tr>
<tr>
<td>netdb.h</td>
<td>scala.scalanative.posix.netdb</td>
</tr>
<tr>
<td>netinet.in.h</td>
<td>scala.scalanative.posix.netinet.in</td>
</tr>
<tr>
<td>netinet/tcp.h</td>
<td>N/A</td>
</tr>
<tr>
<td>nl_types.h</td>
<td>N/A</td>
</tr>
<tr>
<td>poll.h</td>
<td>scala.scalanative.posix.poll</td>
</tr>
<tr>
<td>pthread.h</td>
<td>scala.scalanative.posix.pthread</td>
</tr>
<tr>
<td>pwd.h</td>
<td>scala.scalanative.posix.pwd</td>
</tr>
<tr>
<td>regex.h</td>
<td>scala.scalanative.posix.regex</td>
</tr>
<tr>
<td>sched.h</td>
<td>scala.scalanative.posix.sched</td>
</tr>
<tr>
<td>search.h</td>
<td>N/A</td>
</tr>
<tr>
<td>semaphore.h</td>
<td>N/A</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>N/A</td>
</tr>
<tr>
<td>signal.h</td>
<td>N/A</td>
</tr>
<tr>
<td>spawn.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdarg.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdbool.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stddef.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdint.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdio.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>scala.scalanative.posix.stdlib</td>
</tr>
<tr>
<td>string.h</td>
<td>N/A</td>
</tr>
<tr>
<td>strings.h</td>
<td>N/A</td>
</tr>
<tr>
<td>stropts.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/ipc.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/mman.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/msg.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/resource.h</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>C Header</th>
<th>Scala Native Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys/select.h</td>
<td>scala.scalanative.posix.sys.select</td>
</tr>
<tr>
<td>sys/sem.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/shm.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/socket.h</td>
<td>scala.scalanative.posix.sys.socket</td>
</tr>
<tr>
<td>sys/stat.h</td>
<td>scala.scalanative.posix.sys.stat</td>
</tr>
<tr>
<td>sys/statvfs.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/time.h</td>
<td>scala.scalanative.posix.sys.time</td>
</tr>
<tr>
<td>sys/times.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/types.h</td>
<td>scala.scalanative.posix.sys.types</td>
</tr>
<tr>
<td>sys/uio.h</td>
<td>scala.scalanative.posix.sys.uio</td>
</tr>
<tr>
<td>sys/un.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/utsname.h</td>
<td>N/A</td>
</tr>
<tr>
<td>sys/wait.h</td>
<td>N/A</td>
</tr>
<tr>
<td>syslog.h</td>
<td>scala.scalanative.posix.syslog</td>
</tr>
<tr>
<td>tar.h</td>
<td>N/A</td>
</tr>
<tr>
<td>termios.h</td>
<td>scala.scalanative.posix.termios</td>
</tr>
<tr>
<td>tgmath.h</td>
<td>N/A</td>
</tr>
<tr>
<td>time.h</td>
<td>scala.scalanative.posix.time</td>
</tr>
<tr>
<td>trace.h</td>
<td>N/A</td>
</tr>
<tr>
<td>ulimit.h</td>
<td>N/A</td>
</tr>
<tr>
<td>unistd.h</td>
<td>scala.scalanative.posix.unistd</td>
</tr>
<tr>
<td>utime.h</td>
<td>scala.scalanative.posix.utime</td>
</tr>
<tr>
<td>utmpx.h</td>
<td>N/A</td>
</tr>
<tr>
<td>wchar.h</td>
<td>N/A</td>
</tr>
<tr>
<td>wctype.h</td>
<td>N/A</td>
</tr>
<tr>
<td>wordexp.h</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Continue to Community Libraries.

### 2.2.4 Community Libraries

Third-party libraries for Scala Native can be found using:

- Scala Native libraries indexed by MVN Repository.
- Awesome Scala Native, a curated list of Scala Native libraries and projects.

Continue to FAQ.

### 2.3 Contributor’s Guide

#### 2.3.1 Contributing guidelines

**Very important notice about Javalib**

Scala Native contains a re-implementation of part of the JDK.

Although the GPL and Scala License are compatible and the GPL and Scala CLA are compatible, EPFL wish to distribute scala native under a permissive license.
When you sign the Scala CLA you are confirming that your contributions are your own creation. This is especially important, as it denies you the ability to copy any source code, e.g. Android, OpenJDK, Apache Harmony, GNU Classpath or Scala.js. To be clear, you are personally liable if you provide false information regarding the authorship of your contribution.

However, we are prepared to accept contributions that include code copied from Scala.js or Apache Harmony project on a case-by-case basis. In such cases, you must fulfil your obligations and include the relevant copyright / license information.

**Coding style**

Scala Native is formatted via `./scripts/scalafmt` and `./scripts/clangfmt`. Make sure that all of your contributions are properly formatted before suggesting any changes.

**General workflow**

This is the general workflow for contributing to Scala Native.

1. Make sure you have signed the Scala CLA. If not, sign it.
2. You should always perform your work in its own Git branch. The branch should be given a descriptive name that explains its intent.
3. When the feature or fix is completed you should open a Pull Request on GitHub.
4. The Pull Request should be reviewed by other maintainers (as many as feasible/practical), among which at least one core developer. Independent contributors can also participate in the review process, and are encouraged to do so.
5. After the review, you should resolve issues brought up by the reviewers as needed (amending or adding commits to address reviewers’ comments), iterating until the reviewers give their thumbs up, the “LGTM” (acronym for “Looks Good To Me”).
6. Once the code has passed review the Pull Request can be merged into the distribution.

**Git workflow**

Scala Native repositories maintain a linear merge-free history on the master branch. All of the incoming pull requests are merged using squash and merge policy (i.e. one merged pull request corresponds to one squashed commit to the master branch.)

You do not need to squash commits manually. It’s typical to add new commits to the PR branch to accommodate changes that were suggested by the reviewers. Squashing things manually and/or rewriting history on the PR branch is all-right as long as it’s clear that concerns raised by reviewers have been addressed.

Maintaining a long-standing work-in-progress (WIP) branch requires one to rebase on top of latest master using `git rebase --onto` from time to time. It’s strongly recommended not to perform any merges on your branches that you are planning to use as a PR branch.

**Pull Request Requirements**

In order for a Pull Request to be considered, it has to meet these requirements:

1. Live up to the current code standard:
   - Be formatted with `./scripts/scalafmt` and `./scripts/clangfmt`.
• Not violate DRY.
• Boy Scout Rule should be applied.

2. Be accompanied by appropriate tests.

3. Be issued from a branch other than master (PRs coming from master will not be accepted.)

If not all of these requirements are met then the code should not be merged into the distribution, and need not even be reviewed.

Documentation

All code contributed to the user-facing standard library (the `nativelib/` directory) should come accompanied with documentation. Pull requests containing undocumented code will not be accepted.

Code contributed to the internals (nscplugin, tools, etc.) should come accompanied by internal documentation if the code is not self-explanatory, e.g., important design decisions that other maintainers should know about.

Creating Commits And Writing Commit Messages

Follow these guidelines when creating public commits and writing commit messages.

Prepare meaningful commits

If your work spans multiple local commits (for example; if you do safe point commits while working in a feature branch or work in a branch for long time doing merges/rebases etc.) then please do not commit it all but rewrite the history by squashing the commits into one commit per useful unit of change, each accompanied by a detailed commit message. For more info, see the article: Git Workflow. Additionally, every commit should be able to be used in isolation—that is, each commit must build and pass all tests.

First line of the commit message

The first line should be a descriptive sentence about what the commit is doing, written using the imperative style, e.g., “Change this.”, and should not exceed 70 characters. It should be possible to fully understand what the commit does by just reading this single line. It is not ok to only list the ticket number, type “minor fix” or similar. If the commit has a corresponding ticket, include a reference to the ticket number, with the format “Fix #xxx: Change that.”, as the first line. Sometimes, there is no better message than “Fix #xxx: Fix that issue.”, which is redundant. In that case, and assuming that it aptly and concisely summarizes the commit in a single line, the commit message should be “Fix #xxx: Title of the ticket.”.

Body of the commit message

If the commit is a small fix, the first line can be enough. Otherwise, following the single line description should be a blank line followed by details of the commit, in the form of free text, or bulleted list.

2.3.2 Guide to the sbt build

This section gives some basic information and tips about the build system. The sbt build system is quite complex and effectively brings together all the components of Scala Native. The `build.sbt` file is at the root of the project along with the sub-projects that make up the system.
Overview

In order to effectively work with Scala Native, a knowledge of the build system is very helpful. In general the code is built and published to your local Ivy repository using the `sbt publishLocal` command so that other components in the system can depend on each other via normal `sbt` dependencies. Although the `build.sbt` file and other code in the system is the way to learn the system thoroughly, the following sections will give information that should be helpful to get started.

The build has roughly four groups of sub-projects as follows:

1. The Native Scala Compiler plugin and libraries. Each of these projects depend on the next project in the list.
   - `nscplugin`
   - `nativelib`
   - `clib`
   - `posixlib`
   - `javalib`
   - `auxlib`
   - `scalalib`

2. The Scala Native plugin and dependencies (directory names are in parentheses).
   - `sbtScalaNative (sbt-scala-native)`
   - `tools`
   - `nir, util`
   - `nirparser`

3. The Scala Native test interface and dependencies.
   - `testInterface (test-interface)`
   - `testInterfaceSerialization (test-interface-serialization)`
   - `testInterfaceSbtDefs (test-interface-sbt-defs)`

4. Tests and benchmarks (no dependencies on each other).
   - `tests (unit-tests)`
   - `tools` This has tests within the project
   - `(scripted-tests)`
   - `benchmarks`

Each of the groups above also depend on the previous group being compiled and published locally. The sbt plugin `sbtScalaNative` is used inside Scala Native exactly as it is used in a project using Scala Native. The plugin is needed by the `testInterface` and also the `tests` that use the `testInterface` to compile native code. Please refer to the `build.sbt` file as the final authority.

Building Scala Native

Once you have cloned Scala Native from git, `cd` into the base directory. Inside this directory is the `build.sbt` file which is used to build Scala Native. This file has `sbt` command aliases which are used to help build the system. In order to build Scala Native for the first time you should run the following commands:
If you want to run all the tests and benchmarks, which takes awhile, you can run the `test-all` command after the systems builds.

**Normal development workflow**

Let us suppose that you wish to work on the *javalib* project to add some code or fix a bug. Once you make a change to the code, run the following command at the `sbt` prompt to compile the code and run the tests:

```
> javalib/publishLocal
> tests/test
```

You can run only the test of interest by using one of the following commands:

```
> tests/testOnly java.lang.StringSuite
> tests/testOnly *StringSuite
```

Some additional tips are as follows.

- If you change anything in *tools* (linker, optimizer, codegen), you need to **rebuild**, not **reload**. It’s possible because we textually include code of the `sbt` plugin and the toolchain.

- If you change *nscplugin*, **rebuild** is the only option. This is because the Scala compiler uses this plugin to generate the code that Scala Native uses.

**Using Metals for development**

The Scala community has been working on Metals which uses the Language Server Protocol(LSP) to communicate with an editor of your choice such as VSCode.

The default build uses `sbt 0.13.18` for the Scala Native plugin which uses Scala 2.10 which is not supported by Metals. Switching the build to the newer `sbt` version will change the plugin code to compile with Scala 2.12 so the Metals restriction does not apply. When Metals starts it sets an environment variable `METALS_ENABLED=true` but Metals is not able to run the required **rebuild** step so we need to open `sbt` in our own terminal.

You can set the environment variable before running `sbt` which will switch the build to use the newer `sbt`. The exact version is specified in the build.

```
$ export METALS_ENABLED=true
$ sbt
> rebuild
```

If you do not wish to set an environment variable, you can set the `sbt` version at the `sbt` prompt as follows:

```
$ sbt
> ^^1.2.8
> rebuild
```

Once these steps are completed, you can open the Scala Native project with VSCode and Metals provides a very good IDE experience. Metals also allows you to run `scalafmt` from within the editor. Please refer to the link above for more information or use the Gitter chat here.
Build settings via environment variables

Two build settings, nativeGC and nativeMode can be changed via environment variables. They have default settings that are used unless changed. The setting that controls the garbage collector is nativeGC. Scala Native has a high performance Garbage Collector (GC) immix that comes with the system or the boehm GC which can be used when the supporting library is installed. The setting none also exists for a short running script or where memory is not an issue.

Scala Native uses Continuous integration (CI) to compile and test the code on different platforms and using different garbage collectors. The Scala Native sbt plugin includes the ability to set an environment variable SCALANATIVE_GC to set the garbage collector value used by sbt. Setting this as follows will set the value in the plugin when sbt is run.

```
$ export SCALANATIVE_GC=immix
$ sbt
> show nativeGC
```

This setting remains unless changed at the sbt prompt. If changed, the value will be restored to the environment variable value if sbt is restarted or reload is called at the sbt prompt. You can also revert to the default setting value by running unset SCALANATIVE_GC at the command line and then restarting sbt.

The nativeMode setting is controlled via the SCALANATIVE_MODE environment variable. The default mode, debug is designed to optimize but compile fast whereas the release mode performs additional optimizations and takes longer to compile.

Setting the GC setting via sbt

The GC setting is only used during the link phase of the Scala Native compiler so it can be applied to one or all the Scala Native projects that use the sbtScalaNative plugin. This is an example to only change the setting for the sandbox.

```
$ sbt
> show nativeGC
> set nativeGC in sandbox := "none"
> show nativeGC
> sandbox/run
```

The following shows how to set nativeGC on all the projects.

```
> set every nativeGC := "immix"
> show nativeGC
```

The same process above will work for setting nativeMode.

The next section has more build and development information for those wanting to work on The compiler plugin and code generator.

2.3.3 The compiler plugin and code generator

Compilation to native code happens in two steps. First, Scala code is compiled into Native Intermediate Representation by nscplugin, the Scala compiler plugin. It runs as one of the later phases of the Scala compiler and inspects the AST and generates .nir files. Finally, the .nir files are compiled into .ll files and passed to LLVM by the native compiler.
Fig. 1: High-level overview of the compilation process.

**Tips for working on the compiler**

When adding a new intrinsic, the first thing to check is how clang would compile it in C. Write a small program with the behavior you are trying to add and compile it to `.ll` using:

```
clang -S -emit-llvm foo.c
```

Now write the equivalent Scala code for the new intrinsic in the sandbox project. This project contains a minimal amount of code and has all the toolchain set up which makes it fast to iterate and inspect the output of the compilation.

To compile the sandbox project run the following in the sbt shell:

```
sbt> ;sandbox/clean;sandbox/nativeLink
```

If the example code for the new intrinsic requires you to change APIs in `nativelib`, then remember to also publish the changes with `nativelib/publishLocal`.

After compiling the sandbox project you can inspect the `.ll` files inside `sandbox/target/scala-<version>/ll`. The files are grouped by the package name. By default the `Test.scala` file doesn’t define a package, so the resulting file will be `__empty.ll`. Locating the code you are interested in might require that you get more familiar with the LLVM assembly language.

When working on the compile plugin you’ll need to publish it and reload each time you want to recompile the sandbox project. This can be achieved with:

```
sbt> ;nscplugin/publishLocal;reload;sandbox/clean;sandbox/run
```

Certain intrinsics might require adding new primitives to the compiler plugin. This can be done in `NirPrimitives` with an accompanying definition in `NirDefinitions`. Ensure that new primitives are correctly registered.

The NIR code generation uses a builder to maintain the generated instructions. This allows to inspect the instructions before and after the part of the compilation you are working on has generated code.

### 2.3.4 Native Intermediate Representation

NIR is high-level object-oriented SSA-based representation. The core of the representation is a subset of LLVM instructions, types and values, augmented with a number of high-level primitives that are necessary to efficiently
compile modern languages like Scala.

Contents

- Native Intermediate Representation
  - Introduction
  - Definitions
    - Var
    - Const
    - Declare
    - Define
    - Struct
    - Trait
    - Class
    - Module
  - Types
    - Void
    - Vararg
    - Pointer
    - Boolean
    - Integer
    - Float
    - Array
    - Function
    - Struct
    - Unit
    - Nothing
    - Class
    - Trait
    - Module
  - Control-Flow
    - unreachable
    - ret
    - jump
    - if
    - switch
    - invoke
* throw
  * try

  ** Operands
  * call
  * load
  * store
  * elem
  * extract
  * insert
  * stackalloc
  * bin
  * comp
  * conv
  * sizeof
  * classalloc
  * field
  * method
  * dynmethod
  * as
  * is

  ** Values
  * Boolean
  * Zero and null
  * Integer
  * Float
  * Struct
  * Array
  * Local
  * Global
  * Unit
  * Null
  * String

  ** Attributes
  * Inlining
    * mayinline
Introduction

Lets have a look at the textual form of NIR generated for a simple Scala module:

```scala
object Test {
  def main(args: Array[String]): Unit =
    println("Hello, world!")
}
```

Would map to:

```scala
pin(@Test$::init) module @Test$ : @java.lang.Object
def @Test$::main_class.ssnr.ObjectArray_unit : (module @Test$, class @scala.
  →scalanative.runtime.ObjectArray) => unit {
  %src.2(%src.0 : module @Test$, %src.1 : class @scala.scalanative.runtime.
  →ObjectArray):
    %src.3 = module @scala.Predef$
    %src.4 = method %src.3 : module @scala.Predef$, @scala.Predef$::println_class.
      →java.lang.Object_unit
    %src.5 = call[(module @scala.Predef$, class @java.lang.Object) => unit] %src.4 :
      →ptr(%src.3 : module @scala.Predef$, "Hello, world!")
      ret %src.5 : unit
  }
def @Test$::init : (module @Test$) => unit {
  %src.1(%src.0 : module @Test$):
    %src.2 = call[(class @java.lang.Object) => unit] @java.lang.Object::init : ptr( 
      %src.0 : module @Test$)
    ret unit
}
```

Here we can see a few distinctive features of the representation:
1. At its core NIR is very much a classical SSA-based representation. The code consists of basic blocks of instructions. Instructions take value and type parameters. Control flow instructions can only appear as the last instruction of the basic block.

2. Basic blocks have parameters. Parameters directly correspond to phi instructions in the classical SSA.

3. The representation is strongly typed. All parameters have explicit type annotations. Instructions may be overloaded for different types via type parameters.

4. Unlike LLVM, it has support for high-level object-oriented features such as garbage-collected classes, traits and modules. They may contain methods and fields. There is no overloading or access control modifiers so names must be mangled appropriately.

5. All definitions live in a single top-level scope indexed by globally unique names. During compilation they are lazily loaded until all reachable definitions have been discovered. 

Definitions

Var

```
..$attrs var @$name: $ty = $value
```

Corresponds to LLVM’s global variables when used in the top-level scope and to fields, when used as a member of classes and modules.

Const

```
..$attrs const @$name: $type = $value
```

Corresponds to LLVM’s global constant. Constants may only reside on the top-level and can not be members of classes and modules.

Declare

```
..$attrs def @$name: $type
```

Correspond to LLVM’s declare when used on the top-level of the compilation unit and to abstract methods when used inside classes and traits.

Define

```
..$attrs def @$name: $type { ..$blocks }
```

Corresponds to LLVM’s define when used on the top-level of the compilation unit and to normal methods when used inside classes, traits and modules.
Struct

```scala
$.attrs struct @name { ..$types }
```

Corresponds to LLVM’s named struct.

Trait

```scala
$.attrs trait @name : ..$traits
```

Scala-like traits. May contain abstract and concrete methods as members.

Class

```scala
$.attrs class @name : $parent, ..$traits
```

Scala-like classes. May contain vars, abstract and concrete methods as members.

Module

```scala
$.attrs module @name : $parent, ..$traits
```

Scala-like modules (i.e. `object $name`) May only contain vars and concrete methods as members.

Types

Void

```scala
void
```

Corresponds to LLVM’s `void`.

Vararg

```scala
...
```

Corresponds to LLVM’s `varargs`. May only be nested inside function types.

Pointer

```scala
ptr
```

Corresponds to LLVM’s `pointer type` with a major distinction of not preserving the type of memory that’s being pointed at. Pointers are going to become untyped in LLVM in near future too.
**Boolean**

bool

Corresponds to LLVM’s i1.

**Integer**

i8
i16
i32
i64

Corresponds to LLVM’s integer types. Unlike LLVM we do not support arbitrary width integer types at the moment.

**Float**

f32
f64

Corresponds to LLVM’s floating point types.

**Array**

[$type x N]

Corresponds to LLVM’s aggregate array type.

**Function**

(..$args) => $ret

Corresponds to LLVM’s function type.

**Struct**

struct @$name
struct { ..$types }

Has two forms: named and anonymous. Corresponds to LLVM’s aggregate structure type.

**Unit**

unit

A reference type that corresponds to scala.Unit.
**Nothing**

```
nothing
```

Corresponds to `scala.Nothing`. May only be used a function return type.

**Class**

```
class @$name
```

A reference to a class instance.

**Trait**

```
trait @$name
```

A reference to a trait instance.

**Module**

```
module @$name
```

A reference to a module.

**Control-Flow**

**unreachable**

```
unreachable
```

If execution reaches undefined instruction the behaviour of execution is undefined starting from that point. Corresponds to LLVM’s `unreachable`.

**ret**

```
ret $value
```

Returns a value. Corresponds to LLVM’s `ret`.

**jump**

```
jump $next(..$values)
```

Jumps to the next basic block with provided values for the parameters. Corresponds to LLVM’s unconditional version of `br`. 
if

if $cond$ then $next1(..$values1)$ else $next2(..$values2)$

Conditionally jumps to one of the basic blocks. Corresponds to LLVM’s conditional form of br.

switch

switch $value$

  case $value1$ => $next1(..$values1)$
  ...
  default => $nextN(..$valuesN)$

Jumps to one of the basic blocks if $value$ is equal to corresponding $valueN$. Corresponds to LLVM’s switch.

invoke

invoke[$type$] $ptr(..$values$)$ to $success$ unwind $failure$

Invoke function pointer, jump to success in case value is returned, unwind to failure if exception was thrown. Corresponds to LLVM’s invoke.

throw

throw $value$

Throws the values and starts unwinding.

try

try $succ$ catch $failure$

Operands

All non-control-flow instructions follow a general pattern of %$name$ = $opname[..$types$]$ ..$values$. Purely side-effecting operands like store produce unit value.

call

call[$type$] $ptr(..$values$)$

Calls given function of given function type and argument values. Corresponds to LLVM’s call.
load

\texttt{load[\texttt{type}] \texttt{ptr}}

Load value of given type from memory. Corresponds to LLVM’s \texttt{load}.

store

\texttt{store[\texttt{type}] \texttt{ptr}, \texttt{value}}

Store value of given type to memory. Corresponds to LLVM’s \texttt{store}.

elem

\texttt{elem[\texttt{type}] \texttt{ptr}, ..\texttt{indexes}}

Compute derived pointer starting from given pointer. Corresponds to LLVM’s \texttt{getelementptr}.

extract

\texttt{extract[\texttt{type}] \texttt{aggrvalue}, \texttt{index}}

Extract element from aggregate value. Corresponds to LLVM’s \texttt{extractvalue}.

insert

\texttt{insert[\texttt{type}] \texttt{aggrvalue}, \texttt{value}, \texttt{index}}

Create a new aggregate value based on existing one with element at index replaced with new value. Corresponds to LLVM’s \texttt{insertvalue}.

stackalloc

\texttt{stackalloc[\texttt{type}]}

Stack allocate a slot of memory big enough to store given type. Corresponds to LLVM’s \texttt{alloca}.

bin

\texttt{$\texttt{bin[\texttt{type}] \texttt{value1}, \texttt{value2}$}}

Where \texttt{$\texttt{bin}$} is one of the following: iadd, fadd, isub, fsub, imul, fmul, sdiv, udiv, fdiv, srem, urem, frem, shl, lshr, ashr, and, or, xor. Depending on the type and signedness, maps to either integer or floating point \texttt{binary operations} in LLVM.
$\texttt{comp}$

\begin{verbatim}
$\texttt{comp[type]}\ \$value1, \$value2
\end{verbatim}

Where $\texttt{comp}$ is one of the following: eq, neq, lt, lte, gt, gte. Depending on the type, maps to either \texttt{icmp} or \texttt{fcmp} with corresponding comparison flags in LLVM.

$\texttt{conv}$

\begin{verbatim}
$\texttt{conv[type]}\ \$value
\end{verbatim}

Where $\texttt{conv}$ is one of the following: trunc, zext, sext, fptrunc, fpext, fptoui, fptosi, uitofp, sitofp, ptrtoint, inttoptr, bitcast. Corresponds to LLVM conversion instructions with the same name.

$\texttt{sizeof}$

\begin{verbatim}
$\texttt{sizeof[type]}
\end{verbatim}

Returns a size of given type.

$\texttt{classalloc}$

\begin{verbatim}
$\texttt{classalloc}\ @$\texttt{name}
\end{verbatim}

Roughly corresponds to \texttt{new \$name} in Scala. Performs allocation without calling the constructor.

$\texttt{field}$

\begin{verbatim}
$\texttt{field[type]}\ \$value, @$\texttt{name}
\end{verbatim}

Returns a pointer to the given field of given object.

$\texttt{method}$

\begin{verbatim}
$\texttt{method[type]}\ \$value, @$\texttt{name}
\end{verbatim}

Returns a pointer to the given method of given object.

$\texttt{dynmethod}$

\begin{verbatim}
$\texttt{dynmethod}\ \$obj, \$signature
\end{verbatim}

Returns a pointer to the given method of given object and signature.
as

\texttt{as[\$type] \$value}

Corresponds to \texttt{\$value.asInstanceOf[\$type]} in Scala.

is

\texttt{is[\$type] \$value}

Corresponds to \texttt{\$value.isInstanceOf[\$type]} in Scala.

Values

Boolean

\texttt{true}
\texttt{false}

Corresponds to LLVM’s true and false.

Zero and null

\texttt{null}
\texttt{zero \$type}

Corresponds to LLVM’s null and zeroinitializer.

Integer

\texttt{Ni8}
\texttt{Ni16}
\texttt{Ni32}
\texttt{Ni64}

Corresponds to LLVM’s integer values.

Float

\texttt{N.Nf32}
\texttt{N.Nf64}

Corresponds to LLVM’s floating point values.
**Struct**

```scala
struct @$name {..$values}`
```

Corresponds to LLVM’s struct values.

**Array**

```scala
array $ty {..$values}
```

Corresponds to LLVM’s array value.

**Local**

```scala
%$name
```

Named reference to result of previously executed instructions or basic block parameters.

**Global**

```scala
@$name
```

Reference to the value of top-level definition.

**Unit**

```scala
unit
```

Corresponds to () in Scala.

**Null**

```scala
null
```

Corresponds to null literal in Scala.

**String**

```scala
"..."
```

Corresponds to string literal in Scala.

**Attributes**

Attributes allow one to attach additional metadata to definitions and instructions.
Inlining

mayinline

Default state: optimiser is allowed to inline given method.

inlinenhint

Optimiser is incentivized to inline given methods but it is allowed not to.

noinline

Optimiser must never inline given method.

alwaysinline

Optimiser must always inline given method.

Linking

link

Automatically put $name on a list of native libraries to link with if the given definition is reachable.

pin

Require $name to be reachable, whenever current definition is reachable. Used to introduce indirect linking dependencies. For example, module definitions depend on its constructors using this attribute.

pin-if

pin-if($name, $cond)
Require $name to be reachable if current and $cond definitions are both reachable. Used to introduce conditional indirect linking dependencies. For example, class constructors conditionally depend on methods overridden in given class if the method that are being overridden are reachable.

**pin-weak**

```
pin-weak(@$name)
```

Require $name to be reachable if there is a reachable dynmethod with matching signature.

**stub**

```
stub
```

Indicates that the annotated method, class or module is only a stub without implementation. If the linker is configured with `linkStubs = false`, then these definitions will be ignored and a linking error will be reported. If `linkStubs = true`, these definitions will be linked.

**Misc**

**dyn**

```
dyn
```

Indication that a method can be called using a structural type dispatch.

**pure**

```
pure
```

Let optimiser assume that calls to given method are effectively pure. Meaning that if the same method is called twice with exactly the same argument values, it can re-use the result of first invocation without calling the method twice.

**extern**

```
extern
```

Use C-friendly calling convention and don’t name-mangle given method.

**override**

```
override(@$name)
```

Attributed method overrides @$name method if @$name is reachable. $name must be defined in one of the super classes or traits of the parent class.
### 2.3.5 Name mangling

Scala Native toolchain mangles names for all definitions except the ones which have been explicitly exported to C using `extern`. Mangling scheme is defined through a simple grammar that uses a notation inspired by Itanium ABI:

```latex
\begin{verbatim}
<mangled-name> ::= 
  _S <defn-name>

<defn-name> ::= 
  T <name>       // top-level name
  M <name> <sig-name> // member name

<sig-name> ::= 
  F <name>       // field name
  R <type-name>+ E // constructor name
  D <name> <type-name>+ E // method name
  P <name> <type-name>+ E // proxy name
  C <name>       // c extern name
  G <name>       // generated name
  K <sig-name> <type-name>+ E // duplicate name

<translation> ::= 
  v       // c vararg
  R _     // c pointer <type-name>
  S <type-name>+ E // c function <type-name>
  R <type-name>+ E // c anonymous struct <type-name>
  A <type-name> <number> _ // c array <type-name>
  <integer-type-name> // signed integer <type-name>
  z       // scala.Boolean
  c       // scala.Char
  f       // scala.Float
  d       // scala.Double
  u       // scala.Unit
  l       // scala.Null
  n       // scala.Nothing
  L <nullable-type-name> // nullable <type-name>
  A <type-name> _ // nonnull array <type-name>
  X <name> // nonnull exact <type-name>
  <name> // nonnull <type-name>

<nullable-type-name> ::= 
  A <type-name> _ // nullable array <type-name>
  X <name> // nullable exact <type-name>

<integer-type-name> ::= 
  b // scala.Byte
  s // scala.Short
  i // scala.Int
  j // scala.Long

<name> ::= 
  <length number> <chars> // raw identifier of given length
\end{verbatim}
```
2.4 Blog

2.4.1 Interflow: Scala Native’s upcoming flow-sensitive, profile-guided optimizer

June 16, 2018.

This post provides a sneak peak at Interflow, an upcoming optimizer for Scala Native. For more details, see our publication preprint.

The Interflow Optimizer

Scala Native relies on LLVM as its primary optimizer as of the latest 0.3.7 release. Overall, we’ve found that LLVM fits this role quite well, after all, it is an industry-standard toolchain for AOT compilation of statically typed programming languages. LLVM produces high-quality native code, and the results are getting better with each release.

However, we have also found that LLVM intermediate representation is sometimes too low-level for the Scala programming language. For example, it does not have direct support for object-oriented features such as classes, allocations, virtual calls on them, instance checks, casts, etc. We encode all of those features by lowering them into equivalent code using C-like abstractions LLVM provides us. As a side effect of this lossy conversion, some of the optimization opportunities are irreversibly lost.

To address the abstraction gap between Scala’s high-level features and LLVM’s low-level representation, we developed our own interprocedural, flow-sensitive optimizer called Interflow. It operates on the Scala Native’s intermediate representation called NIR. Unlike LLVM IR, it preserves full information about object-oriented features.

Interflow fuses following static optimizations in a single optimization pass:

- **Flow-sensitive type inference.** Interflow discards most of the original type information ascribed to the methods. Instead, we recompute it using flow-sensitive type inference starting from the entry point of the program. Type inference infers additional exact and nonnull type qualifiers which are not present in the original program. Those qualifiers aid partial evaluation in the elimination of instance checks and virtual calls.

- **Method duplication.** To propagate inferred type information across method boundaries, Interflow relies on duplication. Methods are duplicated once per unique signature, i.e., a list of inferred parameter types. Method duplication is analogous (although not strictly equivalent) to monomorphization in other languages such as C++ or Rust.

- **Partial evaluation.** As part of its traversal, Interflow partially evaluates instance checks, casts, and virtual calls away and replace them with statically predicted results. Partial evaluation removes computations that can be done at compile time and improves the precision of inferred types due to elimination of impossible control flow paths.

- **Partial escape analysis.** Interflow elides allocations which do not escape. It relies on a variation of a technique called partial escape analysis and scalar replacement. This optimization enables elimination of unnecessary closures, boxes, decorators, builders and other intermediate allocations.

- **Inlining.** Interflow performs inlining in the same pass as the rest of the optimizations. This opens the door for caller sensitive information based on partial evaluation and partial escape analysis to be taken into account to decide if method call should be inlined.

Additionally, we also add support for following profile-guided optimizations:

- **Polymorphic inline caching.** Interflow devirtualizes based on flow-sensitive type inference, but it can not predict all of the virtual calls. To aid static devirtualization, we also add support for dynamic devirtualization based on collected type profiles.
• **Untaken branch pruning.** Some of the application code paths (such as error handling) are rarely taken on typical workloads. Untaken branch pruning detects them based on profile data and hoists them out of a method. This optimization reduces code bloat and helps the inliner due to smaller code size left in the method.

• **Profile-directed code placement.** Using the basic block frequency LLVM optimizer can improve native code layout to have the likely branches closer together. It improves generated code quality and can have a significant performance impact on some of the workloads.

**Evaluation**

Note: the performance numbers shown here are based on the current development snapshot of the Interflow, they may change substantially in the final released version.

We run our current prototype of Interflow on Scala Native benchmarks on a machine equipped with Intel i9 7900X CPU. Interflow achieves up to 3.09x higher throughput (with a geometric mean speedup of 1.8x) than Scala Native 0.3.7. Moreover, with the addition of PGO, Interflow gets up to 4.71x faster (with a geometric mean speedup 1.96x) faster than the Scala Native:

![Performance Chart](image)

Additionally, we also compare our performance results with Graal Native Image (1.0-RC1 Enterprise Edition) and warmed up HotSpot JDK (1.8.0-1711-b11).
Both Scala Native 0.3.7 (geomean 0.49x) and Native Image 1.0-RC1 (geomean 0.47x) without PGO fail to achieve performance comparable to the a warmed-up JIT compiler. Native Image’s implementation of PGO obtains impressive speedups, but it is still behind JDK8 (geomean 0.73x).

On the other hand, Interflow (geomean 0.89x) outperforms Graal Native Image statically. With the addition of PGO, Interflow gets quite close to the throughput of a fully warmed JIT compiler (geomean 0.96x).

Interestingly enough, with Interflow, profile-guided optimizations are not strictly required to get the best performance on 7 out of 12 benchmarks. PGO is just an added extra that can push last 5-10% of the performance envelope.

**Conclusion**

This post provides a sneak peak at Interflow, an upcoming optimizer for Scala Native. Additionally, we’re also going to provide support for profile-guided optimization as an opt-in feature for users who want to obtain absolute best peak performance for Scala Native compiled code. Interflow and PGO are currently in development. Stay tuned for updates on general availability on twitter.com/scala_native.

### 2.5 Changelog

**2.5.1 0.4.0-M2 (May 23, 2019)**

Read release notes for 0.4.0-M2 on GitHub.

**2.5.2 0.4.0-M1 (May 23, 2019)**

Read release notes for 0.4.0-M1 on GitHub.

**2.5.3 0.3.9 (Apr 23, 2019)**

Read release notes for 0.3.8 on GitHub.
2.5.4 0.3.8 (Jul 16, 2018)
Read release notes for 0.3.8 on GitHub.

2.5.5 0.3.7 (Mar 29, 2018)
Read release notes for 0.3.7 on GitHub.

2.5.6 0.3.6 (Dec 12, 2017)
Read release notes for 0.3.6 on GitHub.

2.5.7 0.3.5 (Dec 12, 2017)
Read release notes for 0.3.5 on GitHub.

2.5.8 0.3.4 (Dec 12, 2017)
Read release notes for 0.3.4 on GitHub.

2.5.9 0.3.3 (Sep 7, 2017)
Read release notes for 0.3.3 on GitHub.

2.5.10 0.3.2 (Aug 8, 2017)
Read release notes for 0.3.2 on GitHub.

2.5.11 0.3.1 (June 29, 2017)
Read release notes for 0.3.1 on GitHub.

2.5.12 0.3.0 (June 15, 2017)
Read release notes for 0.3.0 on GitHub.

2.5.13 0.2.1 (April 27, 2017)
Read release notes for 0.2.1 on GitHub.

2.5.14 0.2.0 (April 26, 2017)
Read release notes for 0.2.0 on GitHub.
2.5.15 0.1.0 (March 14, 2017)

Read original announcement on scala-lang.org

2.6 FAQ

Q: How do I make the resulting executable smaller?
A: Compress the binary with https://upx.github.io/

Q: Does Scala Native support WebAssembly?
A: Support for WebAssembly is out of scope for the project. If you need to run Scala code in the browser, consider using Scala.js instead.

2.6.1 Troubleshooting

When compiling your Scala Native project, the linker ld may fail with the following message:

```
relocation R_X86_64_32 against `.rodata.str1.1' can not be used when making a shared object; recompile with -fPIC
```

It is likely that the LDFLAGS environment variable enables hardening. For example, this occurs when the hardening-wrapper package is installed on Arch Linux. It can be safely removed.