# Introduction to the sbt Build Tool

Effective Programming in Scala

# sbt

sbt is a build tool commonly used in Scala.

Example: the assignments of this course are sbt projects.

Note: there are other build tools that support Scala: Maven, Gradle, Mill, etc.

# Hello, sbt!

Make sure the `sbt` command-line tool is installed according to the instructions at https://scala-sbt.org.

An sbt project is a directory with the following two files:

```
project/build.properties
build.sbt
```

# Hello, sbt! (2)

The file `project/build.properties` defines the version of sbt to use:

```
sbt.version=1.4.6
```

The file `build.sbt` defines (among others) the version of Scala to use:

```
scalaVersion := "3.0.0-RC1"
```

- The sbt launcher will make sure to use the appropriate version of sbt to load your build definition, and to use the appropriate version of the Scala compiler to compile your Scala code.

# Hello, sbt! (3)

In a command-line prompt, invoke `sbt` from the project root directory (the one that contains the file `build.sbt`):

```
$ sbt
```

The sbt launcher downloads the required version of sbt, if necessary, and starts the **sbt shell**, which lets you invoke tasks:

```
sbt:hello-sbt>
```

# Hello, sbt! (4)

By default, sbt compiles source files that are in the `src/main/scala` directory. Let us define a main method in the package `hellosbt`, and play with the program:

```
// File src/main/scala/hellosbt/HelloSbt.scala
package hellosbt

val greeting = "Hello, sbt!"

@main def run(): Unit = println(greeting)
```

# Compile

The `compile` task compiles the program source files:

```
sbt:hello-sbt> compile
```

- Output of compilation is cached in a directory `target/`
- Try to change the source code and compile again!

# Run

The `run` task compiles and then runs the program:

```
sbt:hello-sbt> run
```

# Read-Eval-Print Loop

The `console` task compiles the program and opens a REPL:

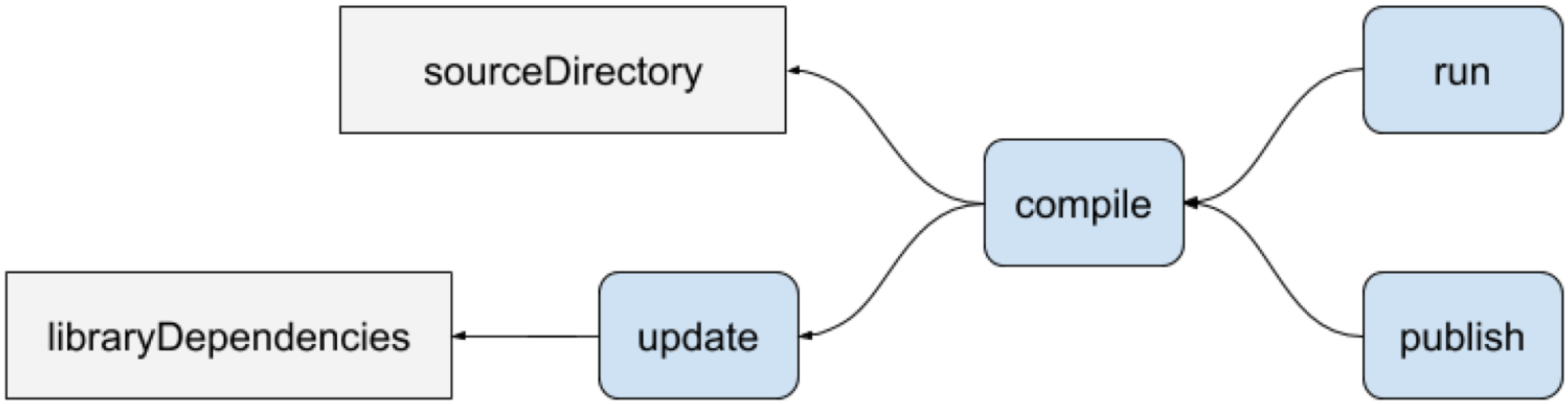
```
sbt:hello-sbt> console
```

In the REPL, you can evaluate your program, or parts of your program:

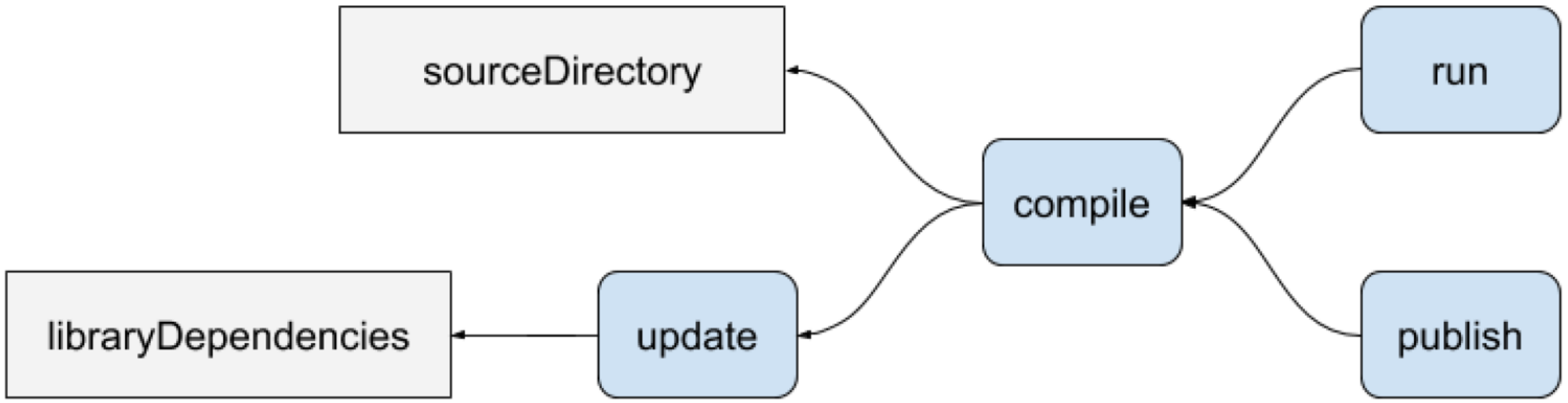
```
scala> hellosbt.greeting
val res0: String = Hello, sbt!

scala> hellosbt.greeting.toUpperCase
val res1: String = HELLO, SBT!
```

# Main Concepts



# Main Concepts




**Settings** parameterize the build, they are evaluated once

**Tasks** perform actions (download, compile, run, etc.), they are evaluated at each invocation

Tasks can be parameterized by settings' values and other tasks' results.

# Using Libraries

We indicate, in the build definition, that we want to use a library by adding it to the `libraryDependencies` setting:

```
// File build.sbt
scalaVersion := "3.0.0-RC1"
libraryDependencies += "com.lihaoyi" %% "fansi" % "0.2.10"
```

A library is identified by an organization name, an artifact name, and a revision number.

You can find these "coordinates" in an index, such as `https://index.scala-lang.org`.

Invoke the `reload` command after you change the build definition!

# Using Libraries (2)

```scala
// File src/main/scala/hellosbt/HelloSbt.scala
package hellosbt

val greeting = "Hello, sbt!"

@main def run(): Unit =
  println(fansi.Color.Red(greeting))
```

We can now use the library in our source code. The build tool downloads the library and adds it to our classpath when we run the program.

# Build Definition

The `build.sbt` file contains the build definition. It is written in Scala.

```
scalaVersion := "3.0.0-RC1"
libraryDependencies += "com.lihaoyi" %% "fansi" % "0.2.10"
```

Note that there are some differences between regular Scala programs and sbt build definitions. In particular, `.sbt` files automatically import sbt setting keys (such as `libraryDependencies` or `scalaVersion`).

# Keys Manipulation

Setting and task keys are regular Scala values.

They have operations like :=, += or ++= to set their values.

For example, to add several library dependencies:

```
libraryDependencies ++= List(
  "com.lihaoyi"                 %% "fansi" % "0.2.10",
  "com.softwaremill.sttp.client" %% "core"  % "2.2.9"
)
```

# Test

Running tests requires adding a test framework:

```
// File build.sbt
libraryDependencies += "org.scalameta" %% "munit" % "0.7.22" % Test
testFrameworks += new TestFramework("munit.Framework")
```

Note: we explicitly mention that this library should be added to the `Test` "scope" only (as opposed to the program itself).

# Test (2)

By default, sbt looks for test sources in the `src/test/scala` directory.

Let's add a test to our program:

```
// File src/test/scala/hellosbt/HelloSbtSuite.scala
package hellosbt

import munit.FunSuite

class HelloSbtSuite extends FunSuite:
  test("try the impossible") {
    assert(1 == 0)
  }
```

# Test (3)

Run the tests:

```
sbt:hello-sbt> test
```

# sbt Plugins

sbt provides common predefined tasks and settings out of the box.

Additional tasks or predefined configurations can be provided by **plugins**.

Plugins define how the build itself is managed and must be declared in the file `project/plugins.sbt`.

# Example of Plugin: Errors Summary

This plugin configures the compiler error reporter to present errors in a different way (for instance, the first error is shown at the end of the output).

```
// File project/plugins.sbt
addSbtPlugin("org.duhemm" % "sbt-errors-summary" % "0.6.3")
```

# Example of Plugin: sbt-site

This plugin defines new tasks and settings to make it easier to build a website.

```
// File project/plugins.sbt
addSbtPlugin("com.typesafe.sbt" % "sbt-site" % "1.4.1")

// File build.sbt
makeSite / mappings := {
  val indexFile = target.value / "index.html"
  IO.write(indexFile, "<h1>Hello, sbt!</h1>")
  Seq(indexFile -> "index.html")
}
```

# Summary

sbt is a build tool for Scala.

It is interactive: you start the sbt shell in the morning and manage your project from there.

The build definition is written in Scala. A build definition essentially assigns values to setting keys (such as `scalaVersion`, or `libraryDependencies`).