SBT

sbt is a build tool for Scala and Java projects that aims to do the basics well. It requires Java 1.6 or later.

Features

- Easy to set up for simple projects
- .sbt build definition uses a Scala-based "domain-specific language" (DSL)
- More advanced .scala build definitions and extensions use the full flexibility of unrestricted Scala code
- Accurate incremental recompilation using information extracted from the compiler
- Continuous compilation and testing with triggered execution
- Packages and publishes jars
- Generates documentation with scaladoc
- Supports mixed Scala/Java projects
- Supports Testing with ScalaCheck, specs, and ScalaTest (JUnit is supported by a plugin)
- Starts the Scala REPL with project classes and dependencies on the classpath
- Sub-project support (put multiple packages in one project)
- External project support (list a git repository as a dependency!)
- Parallel task execution, including parallel test execution
- Library management support: inline declarations, external Ivy or Maven configuration files, or manual management

Getting Started

To get started, read the Getting Started Guide.

Please read the Getting Started Guide. You will save yourself a *lot* of time if you have the right understanding of the big picture up-front.

If you are familiar with 0.7.x, please see the migration page. Documentation for 0.7.x is still available on the Google Code Site. This wiki applies to sbt 0.10 and later.

The mailing list is at http://groups.google.com/group/simple-build-tool/topics. Please use it for questions and comments!

This wiki is editable if you have a GitHub account. Feel free to make corrections and add documentation. Use the mailing list if you have questions or comments.

Welcome

Getting Started Guide page 1 of 14. Next

This Getting Started Guide will get you started with sbt.

sbt uses a small number of concepts to support flexible and powerful build definitions. There are not that many concepts, but sbt is not exactly like other build systems and there are details you *will* stumble on if you haven't read the documentation.

The Getting Started Guide covers the concepts you need to know to create and maintain an sbt build definition.

It is *highly recommended* to read the Getting Started Guide!

If you are in a huge hurry, the most important conceptual background can be found in .sbt build definition, scopes, and more about settings. But we don't promise that it's a good idea to skip the other pages in the guide.

It's best to read in order, as later pages in the Getting Started Guide build on concepts introduced earlier.

The guide begins with Setup.

Thanks for trying out sbt and *have fun*!

Getting Started Setup

Getting Started Guide page 2 of 14. Next

Overview

To create an sbt project, you'll need to take these steps:

- Install sbt and create a script to launch it.
- Setup a simple hello world project
  - Create a project directory with source files in it.
  - Create your build definition.
- Move on to running to learn how to run sbt.
- Then move on to .sbt build definition to learn more about build definitions.

Installing sbt

You need two files; sbt-launch.jar and a script to run it.

Note: Relevant information is moving to the download page

Yum

The sbt package is available from the Typesafe Yum Repository. Please install this rpm to add the typesafe yum repository to your list of approved sources. Then run:

```
yum install sbt
```

to grab the latest release of sbt.
*Note: please make sure to report any issues you may find here.

Apt

The sbt package is available from the Typesafe Debian Repository. Please install this deb to add the typesafe debian repository to your list of approved sources. Then run:

```
apt-get install sbt
```

to grab the latest release of sbt. If sbt cannot be found, dont forget to update your list of repositories. To do so, run:

```
apt-get update
```

*Note: please make sure to report any issues you may find here.

Mac

Use either MacPorts:

```
$ sudo port install sbt
```

Or HomeBrew:

```
$ brew install sbt
```

There is no need to download the sbt-launch.jar separately with either approach.

Windows

You can download the msi

or

Create a batch file `sbt.bat`:

```
set SCRIPT_DIR=%~dp0
java -Xmx512M -jar "%SCRIPT_DIR%sbt-launch.jar" %*
```

and put sbt-launch.jar in the same directory as the batch file. Put `sbt.bat` on your path so that you can launch `sbt` in any directory by typing `sbt` at the command prompt.

Unix

Download sbt-launch.jar and place it in `~/bin`.

Create a script to run the jar, by placing this in a file called `sbt` in your `~/bin` directory:

```
java -Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=384M -jar
`dirname $0`/sbt-launch.jar "$@"
```

Make the script executable:

```
$ chmod u+x ~/bin/sbt
```

Tips and Notes

If you have any trouble running `sbt`, see Setup Notes on terminal encodings, HTTP proxies, and JVM options.

To install sbt, you could also use this fairly elaborated shell script: https://github.com/paulp/sbt-extras (see sbt file int the root dir). It has the same purpose as the simple shell script above but it will install sbt if necessary. It knows all recent versions of sbt and it also comes with a lot of useful command line options.

Getting Started Hello

Getting Started Guide page 3 of 14. Next

This page assumes you've installed sbt.

Create a project directory with source code

A valid sbt project can be a directory containing a single source file. Try creating a directory `hello` with a file `hw.scala`, containing the following:

```
object Hi {
  def main(args: Array[String]) = println("Hi!")
}
```

Now from inside the `hello` directory, start sbt and type `run` at the sbt interactive console. On Linux or OS X the commands might look like this:

```
$ mkdir hello
$ cd hello
$ echo 'object Hi { def main(args: Array[String]) = println("Hi!") }' > hw.scala
$ sbt
...
> run
...
Hi!
```

In this case, sbt works purely by convention. sbt will find the following automatically:

- Sources in the base directory
- Sources in `src/main/scala` or `src/main/java`
- Tests in `src/test/scala` or `src/test/java`
- Data files in `src/main/resources` or `src/test/resources`
- jars in `lib`

By default, sbt will build projects with the same version of Scala used to run sbt itself.

You can run the project with `sbt run` or enter the Scala REPL with `sbt console`. `sbt console` sets up your project's classpath so you can try out live Scala examples based on your project's code.

Build definition

Most projects will need some manual setup. Basic build settings go in a file called `build.sbt`, located in the project's base directory.

For example, if your project is in the directory `hello`, in `hello/build.sbt` you might write:

```
name := "hello"

version := "1.0"

scalaVersion := "2.9.1"
```

Notice the blank line between every item. This isn't just for show; they're actually required in order to separate each item. In .sbt build definition you'll learn more about how to write a `build.sbt` file.

If you plan to package your project in a jar, you will want to set at least the name and version in a `build.sbt`.

Setting the sbt version

You can force a particular version of sbt by creating a file `hello/project/build.properties`. In this file, write:

```
sbt.version=0.11.3
```

From 0.10 onwards, sbt is 99% source compatible from release to release. Still, setting the sbt version in `project/build.properties` avoids any potential confusion.

Getting Started Directories

Getting Started Guide page 4 of 14. Next

This page assumes you've installed sbt and seen the Hello, World example.

Base directory

In sbt's terminology, the "base directory" is the directory containing the project. So if you created a project `hello` containing `hello/build.sbt` and `hello/hw.scala` as in the Hello, World example, `hello` is your base directory.

Source code

Source code can be placed in the project's base directory as with `hello/hw.scala`. However, most people don't do this for real projects; too much clutter.

sbt uses the same directory structure as Maven for source files by default (all paths are relative to the base directory):

```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
    java/
       <test Java sources>
```

Other directories in `src/` will be ignored. Additionally, all hidden directories will be ignored.

sbt build definition files

You've already seen `build.sbt` in the project's base directory. Other sbt files appear in a `project` subdirectory.

`project` can contain `.scala` files, which are combined with `.sbt` files to form the complete build definition. See .scala build definitions for more.

```
build.sbt
project/
  Build.scala
```

You may see `.sbt` files inside `project/` but they are not equivalent to `.sbt` files in the project's base directory. Explaining this will come later, since you'll need some background information first.

Build products

Generated files (compiled classes, packaged jars, managed files, caches, and documentation) will be written to the `target` directory by default.

Configuring version control

Your `.gitignore` (or equivalent for other version control systems) should contain:

```
target/
```

Note that this deliberately has a trailing `/` (to match only directories) and it deliberately has no leading `/` (to match `project/target/` in addition to plain `target/`).

Running

Getting Started Guide page 5 of 14. Next

This page describes how to use `sbt` once you have set up your project. It assumes you've installed sbt and created a Hello, World or other project.

Interactive mode

Run sbt in your project directory with no arguments:

```
$ sbt
```

Running sbt with no command line arguments starts it in interactive mode. Interactive mode has a command prompt (with tab completion and history!).

For example, you could type `compile` at the sbt prompt:

```
> compile
```

To `compile` again, press up arrow and then enter.

To run your program, type `run`.

To leave interactive mode, type `exit` or use Ctrl+D (Unix) or Ctrl+Z (Windows).

Batch mode

You can also run sbt in batch mode, specifying a space-separated list of sbt commands as arguments. For sbt commands that take arguments, pass the command and arguments as one argument to `sbt` by enclosing them in quotes. For example,

```
$ sbt clean compile "test-only TestA TestB"
```

In this example, `test-only` has arguments, `TestA` and `TestB`. The commands will be run in sequence (`clean`, `compile`, then `test-only`).

Continuous build and test

To speed up your edit-compile-test cycle, you can ask sbt to automatically recompile or run tests whenever you save a source file.

Make a command run when one or more source files change by prefixing the command with ~. For example, in interactive mode try:

```
> ~ compile
```

Press enter to stop watching for changes.

You can use the ~ prefix with either interactive mode or batch mode.

See Triggered Execution for more details.

Common commands

Here are some of the most common sbt commands. For a more complete list, see Command Line Reference.

- `clean` Deletes all generated files (in the `target` directory).
- `compile` Compiles the main sources (in `src/main/scala` and `src/main/java` directories).
- `test` Compiles and runs all tests.
- `console` Starts the Scala interpreter with a classpath including the compiled sources and all dependencies. To return to sbt, type `:quit`, Ctrl+D (Unix), or Ctrl+Z (Windows).
- `run <argument>*` Runs the main class for the project in the same virtual machine as `sbt`.
- `package` Creates a jar file containing the files in `src/main/resources` and the classes compiled from `src/main/scala` and `src/main/java`.
- `help <command>` Displays detailed help for the specified command. If no command is provided, displays brief descriptions of all commands.
- `reload` Reloads the build definition (`build.sbt`, `project/*.scala`, `project/*.sbt` files). Needed if you change the build definition.

Tab completion

Interactive mode has tab completion, including at an empty prompt. A special sbt convention is that pressing tab once may show only a subset of most likely completions, while pressing it more times shows more verbose choices.

History Commands

Interactive mode remembers history, even if you exit sbt and restart it. The simplest way to access history is with the up arrow key. The following commands are also supported:

- `!` Show history command help.
- `!!` Execute the previous command again.
- `!:` Show all previous commands.
- `!:n` Show the last n commands.
- `!n` Execute the command with index `n`, as shown by the `!:` command.
- `!-n` Execute the nth command before this one.
- `!string` Execute the most recent command starting with 'string'
- `!?string` Execute the most recent command containing 'string'

.sbt Build Definition

Getting Started Guide page 6 of 14. Next

This page describes sbt build definitions, including some "theory" and the syntax of `build.sbt`. It assumes you know how to use sbt and have read the previous pages in the Getting Started Guide.

`.sbt` vs. `.scala` Definition

An sbt build definition can contain files ending in `.sbt`, located in the base directory, and files ending in `.scala`, located in the `project` subdirectory of the base directory.

You can use either one exclusively, or use both. A good approach is to use `.sbt` files for most purposes, and use `.scala` files only to contain what can't be done in `.sbt`:

- to customize sbt (add new settings or tasks)
- to define nested sub-projects

This page discusses `.sbt` files. See .scala build definition (later in Getting Started) for more on `.scala` files and how they relate to `.sbt` files.

What is a build definition?

** PLEASE READ THIS SECTION **

After examining a project and processing any build definition files, sbt will end up with an immutable map (set of key-value pairs) describing the build.

For example, one key is `name` and it maps to a string value, the name of your project.

Build definition files do not affect sbt's map directly.

Instead, the build definition creates a huge list of objects with type `Setting[T]` where `T` is the type of the value in the map. (Scala's `Setting[T]` is like `Setting<T>` in Java.) A `Setting` describes a *transformation to the map*, such as adding a new key-value pair or appending to an existing value. (In the spirit of functional programming, a transformation returns a new map, it does not update the old map in-place.)

In `build.sbt`, you might create a `Setting[String]` for the name of your project like this:

```
name := "hello"
```

This `Setting[String]` transforms the map by adding (or replacing) the `name` key, giving it the value `"hello"`. The transformed map becomes sbt's new map.

To create its map, sbt first sorts the list of settings so that all changes to the same key are made together, and values that depend on other keys are processed after the keys they depend on. Then sbt walks over the sorted list of `Setting` and applies each one to the map in turn.

Summary: *A build definition defines a list of `Setting[T]`, where a `Setting[T]` is a transformation affecting sbt's map of key-value pairs and `T` is the type of each value.*

How `build.sbt` defines settings

`build.sbt` defines a `Seq[Setting[_]]`; it's a list of Scala expressions, separated by blank lines, where each one becomes one element in the sequence. If you put `Seq(` in front of the `.sbt` file and `)` at the end and replace the blank lines with commas, you'd be looking at the equivalent `.scala` code.

Here's an example:

```
name := "hello"

version := "1.0"

scalaVersion := "2.9.1"
```

A `build.sbt` file is a list of `Setting`, separated by blank lines. Each `Setting` is defined with a Scala expression.

The expressions in `build.sbt` are independent of one another, and they are expressions, rather than complete Scala statements. An implication of this is that you can't define a top-level `val`, `object`, class, or method in `build.sbt`.

On the left, `name`, `version`, and `scalaVersion` are *keys*. A key is an instance of `SettingKey[T]`, `TaskKey[T]`, or `InputKey[T]` where `T` is the expected value type. The kinds of key are explained more below.

Keys have a method called `:=`, which returns a `Setting[T]`. You could use a Java-like syntax to call the method:

```
name.:=("hello")
```

But Scala allows `name := "hello"` instead (in Scala, any method can use either syntax).

The `:=` method on key `name` returns a `Setting`, specifically a `Setting[String]`. `String` also appears in the type of `name` itself, which is `SettingKey[String]`. In this case, the returned `Setting[String]` is a transformation to add or replace the `name` key in sbt's map, giving it the value `"hello"`.

If you use the wrong value type, the build definition will not compile:

```
name := 42  // will not compile
```

Settings are separated by blank lines

You can't write a `build.sbt` like this:

```
// will NOT work, no blank lines
name := "hello"
version := "1.0"
```

```
scalaVersion := "2.9.1"
```

sbt needs some kind of delimiter to tell where one expression stops and the next begins.

`.sbt` files contain a list of Scala expressions, not a single Scala program. These expressions have to be split up and passed to the compiler individually.

If you want a single Scala program, use .scala files rather than `.sbt` files; `.sbt` files are optional. Later on this guide explains how to use `.scala` files. (Preview: the same settings expressions found in a `.sbt` file can always be listed in a `Seq[Setting]` in a `.scala` file instead.)

Keys are defined in the Keys object

The built-in keys are just fields in an object called Keys. A `build.sbt` implicitly has an `import sbt.Keys._`, so `sbt.Keys.name` can be referred to as `name`.

Custom keys may be defined in a .scala file or a plugin.

Other ways to transform settings

Replacement with `:=` is the simplest transformation, but there are several others. For example you can append to a list value with `+=`.

The other transformations require an understanding of scopes, so the next page is about scopes and the page after that goes into more detail about settings.

Task Keys

There are three flavors of key:

- `SettingKey[T]`: a key with a value computed once (the value is computed one time when loading the project, and kept around).
- `TaskKey[T]`: a key with a value that has to be recomputed each time, potentially creating side effects.
- `InputKey[T]`: a task key which has command line arguments as input. The Getting Started Guide doesn't cover `InputKey`, but when you finish this guide, check out Input Tasks for more.

A `TaskKey[T]` is said to define a *task*. Tasks are operations such as `compile` or `package`. They may return `Unit` (`Unit` is Scala for `void`), or they may return a value related to the task, for example `package` is a `TaskKey[File]` and its value is the jar file it creates.

Each time you start a task execution, for example by typing `compile` at the interactive sbt prompt, sbt will re-run any tasks involved exactly once.

sbt's map describing the project can keep around a fixed string value for a setting such as `name`, but it has to keep around some executable code for a task such as `compile` -- even if that executable code eventually returns a string, it has to be re-run every time.

A given key always refers to either a task or a plain setting. That is, "taskiness" (whether to re-run each time) is a property of the key, not the value.

Using `:=`, you can assign a computation to a task, and that computation will be re-run each time:

```
hello := { println("Hello!") }
```

From a type-system perspective, the `Setting` created from a task key is slightly different from the one created from a setting key. `taskKey := 42` results in a `Setting[Task[T]]` while `settingKey := 42` results in a `Setting[T]`. For most purposes this makes no difference; the task key still creates a value of type `T` when the task executes.

The `T` vs. `Task[T]` type difference has this implication: a setting key can't depend on a task key, because a setting key is evaluated only once on project load, and not re-run. More on this in more about settings, coming up soon.

Keys in sbt interactive mode

In sbt's interactive mode, you can type the name of any task to execute that task. This is why typing `compile` runs the compile task. `compile` is a task key.

If you type the name of a setting key rather than a task key, the value of the setting key will be displayed. Typing a task key name executes the task but doesn't display the resulting value; to see a task's result, use `show <task name>` rather than plain `<task name>`.

In build definition files, keys are named with `camelCase` following Scala convention, but the sbt command line uses `hyphen-separated-words` instead. The hyphen-separated string used in sbt comes from the definition of the key (see Keys). For example, in `Keys.scala`, there's this key:

```
val scalacOptions = TaskKey[Seq[String]]("scalac-options", "Options for the Scala
compiler.")
```

In sbt you type `scalac-options` but in a build definition file you use `scalacOptions`.

To learn more about any key, type `inspect <keyname>` at the sbt interactive prompt. Some of the information `inspect` displays won't make sense yet, but at the top it shows you the setting's value type and a brief description of the setting.

Imports in `build.sbt`

You can place import statements at the top of `build.sbt`; they need not be separated by blank lines.

There are some implied default imports, as follows:

```
import sbt._
import Process._
import Keys._
```

(In addition, if you have .scala files, the contents of any `Build` or `Plugin` objects in those files will be imported. More on that when we get to .scala build definitions.)

Adding library dependencies

To depend on third-party libraries, there are two options. The first is to drop jars in `lib/` (unmanaged dependencies) and the other is to add managed dependencies, which will look like this in `build.sbt`:

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

This is how you add a managed dependency on the Apache Derby library, version 10.4.1.3.

The `libraryDependencies` key involves two complexities: `+=` rather than `:=`, and the `%` method. `+=` appends to the key's old value rather than replacing it, this is explained in more about settings. The `%` method is used to construct an Ivy module ID from strings, explained in library dependencies.

We'll skip over the details of library dependencies until later in the Getting Started Guide. There's a whole page covering it later on.

Scopes

Getting Started Guide page 7 of 14. Next

This page describes scopes. It assumes you've read and understood the previous page, .sbt build definition.

The whole story about keys

Previously we pretended that a key like `name` corresponded to one entry in sbt's map of key-value pairs. This was a simplification.

In truth, each key can have an associated value in more than one context, called a "scope."

Some concrete examples:

- if you have multiple projects in your build definition, a key can have a different value in each project.
- the `compile` key may have a different value for your main sources and your test sources, if you want to compile them differently.
- the `package-options` key (which contains options for creating jar packages) may have different values when packaging class files (`package-bin`) or packaging source code (`package-src`).

There is no single value for a given key name, because the value may differ according to scope.

However, there is a single value for a given *scoped* key.

If you think about sbt processing a list of settings to generate a key-value map describing the project, as discussed earlier, the keys in that key-value map are *scoped* keys. Each setting defined in the build definition (for example in `build.sbt`) applies to a scoped key as well.

Often the scope is implied or has a default, but if the defaults are wrong, you'll need to mention the desired scope in `build.sbt`.

Scope axes

A *scope axis* is a type, where each instance of the type can define its own scope (that is, each instance can have its own unique values for keys).

There are three scope axes:

- Projects
- Configurations
- Tasks

Scoping by project axis

If you put multiple projects in a single build, each project needs its own settings. That is, keys can be scoped according to the project.

The project axis can also be set to "entire build", so a setting applies to the entire build rather than a single project. Build-level settings are often used as a fallback when a project doesn't define a project-specific setting.

Scoping by configuration axis

A *configuration* defines a flavor of build, potentially with its own classpath, sources, generated packages, etc. The configuration concept comes from Ivy, which sbt uses for managed dependencies, and from MavenScopes.

Some configurations you'll see in sbt:

- `Compile` which defines the main build (`src/main/scala`).
- `Test` which defines how to build tests (`src/test/scala`).
- `Runtime` which defines the classpath for the `run` task.

By default, all the keys associated with compiling, packaging, and running are scoped to a configuration and therefore may work differently in each configuration. The most obvious examples are the task keys `compile`, `package`, and `run`; but all the keys which *affect* those keys (such as `source-directories` or `scalac-options` or `full-classpath`) are also scoped to the configuration.

Scoping by task axis

Settings can affect how a task works. For example, the `package-src` task is affected by the `package-options` setting.

To support this, a task key (such as `package-src`) can be a scope for another key (such as `package-options`).

The various tasks that build a package (`package-src`, `package-bin`, `package-doc`) can share keys related to packaging, such as `artifact-name` and `package-options`. Those keys can have distinct values for each packaging task.

Global scope

Each scope axis can be filled in with an instance of the axis type (for example the task axis can be filled in with a task), or the axis can be filled in with the special value `Global`.

`Global` means what you would expect: the setting's value applies to all instances of that axis. For example if the task axis is `Global`, then the setting would apply to all tasks.

Delegation

A scoped key may be undefined, if it has no value associated with it in its scope.

For each scope, sbt has a fallback search path made up of other scopes. Typically, if a key has no associated value in a more-specific scope, sbt will try to get a value from a more general scope, such as the `Global` scope or the entire-build scope.

This feature allows you to set a value once in a more general scope, allowing multiple more-specific scopes to inherit the value.

You can see the fallback search path or "delegates" for a key using the `inspect` command, as described below. Read on.

Referring to scoped keys when running sbt

On the command line and in interactive mode, sbt displays (and parses) scoped keys like this:

```
{<build-uri>}<project-id>/config:key(for task-key)
```

- `{<build-uri>}<project-id>` identifies the project axis. The `<project-id>` part will be missing if the project axis has "entire build" scope.
- `config` identifies the configuration axis.
- `(for task-key)` identifies the task axis.
- `key` identifies the key being scoped.

`*` can appear for each axis, referring to the `Global` scope.

If you omit part of the scoped key, it will be inferred as follows:

- the current project will be used if you omit the project.
- a key-dependent configuration will be auto-detected if you omit the configuration.
- the `Global` task scope will be used if you omit the task.

For more details, see Inspecting Settings.

Examples of scoped key notation

- `full-classpath`: just a key, so the default scopes are used: current project, a key-dependent configuration, and global task scope.
- `test:full-classpath`: specifies the configuration, so this is `full-classpath` in the `test` configuration, with defaults for the other two scope axes.
- `*:full-classpath`: specifies `Global` for the configuration, rather than the default configuration.
- `full-classpath(for doc)`: specifies the `full-classpath` key scoped to the `doc` task, with the defaults for the project and configuration axes.
- `{file:/home/hp/checkout/hello/}default-aea33a/test:full-classpath` specifies a project, `{file:/home/hp/checkout/hello/}default-aea33a`, where the project is identified with the build `{file:/home/hp/checkout/hello/}` and then a project id inside that build `default-aea33a`. Also specifies configuration `test`, but leaves the default task axis.
- `{file:/home/hp/checkout/hello/}/test:full-classpath` sets the project axis to "entire build" where the build is `{file:/home/hp/checkout/hello/}`
- `{.}/test:full-classpath` sets the project axis to "entire build" where the build is `{.}`. `{.}` can be written `ThisBuild` in Scala code.
- `{file:/home/hp/checkout/hello/}/compile:full-classpath(for doc)` sets all three scope axes.

Inspecting scopes

In sbt's interactive mode, you can use the `inspect` command to understand keys and their scopes. Try `inspect test:full-classpath`:

```
$ sbt
> inspect test:full-classpath
[info] Task: scala.collection.Seq[sbt.Attributed[java.io.File]]
[info] Description:
[info]      The exported classpath, consisting of build products and unmanaged and
managed, internal and external dependencies.
[info] Provided by:
[info]      {file:/home/hp/checkout/hello/}default-aea33a/test:full-classpath
[info] Dependencies:
[info]      test:exported-products
[info]      test:dependency-classpath
[info] Reverse dependencies:
[info]      test:run-main
[info]      test:run
[info]      test:test-loader
[info]      test:console
[info] Delegates:
[info]      test:full-classpath
[info]      runtime:full-classpath
[info]      compile:full-classpath
```

```
[info]      *:full-classpath
[info]      {.}/test:full-classpath
[info]      {.}/runtime:full-classpath
[info]      {.}/compile:full-classpath
[info]      {.}/*:full-classpath
[info]      */test:full-classpath
[info]      */runtime:full-classpath
[info]      */compile:full-classpath
[info]      */*:full-classpath
[info] Related:
[info]      compile:full-classpath
[info]      compile:full-classpath(for doc)
[info]      test:full-classpath(for doc)
[info]      runtime:full-classpath
```

On the first line, you can see this is a task (as opposed to a setting, as explained in .sbt build definition). The value resulting from the task will have type `scala.collection.Seq[sbt.Attributed[java.io.File]]`.

"Provided by" points you to the scoped key that defines the value, in this case `{file:/home/hp/checkout/hello/}default-aea33a/test:full-classpath` (which is the `full-classpath` key scoped to the `test` configuration and the `{file:/home/hp/checkout/hello/}default-aea33a` project).

"Dependencies" may not make sense yet; stay tuned for the next page.

You can also see the delegates; if the value were not defined, sbt would search through:

- two other configurations (`runtime:full-classpath`, `compile:full-classpath`). In these scoped keys, the project is unspecified meaning "current project" and the task is unspecified meaning `Global`

- configuration set to `Global` (`*:full-classpath`), since project is still unspecified it's "current project" and task is still unspecified so `Global`

- project set to `{.}` or `ThisBuild` (meaning the entire build, no specific project)

- project axis set to `Global` (`*/test:full-classpath`) (remember, an unspecified project means current, so searching `Global` here is new; i.e. `*` and "no project shown" are different for the project axis; i.e. `*/test:full-classpath` is not the same as `test:full-classpath`)

- both project and configuration set to `Global` (`*/*:full-classpath`) (remember that unspecified task means `Global` already, so `*/*:full-classpath` uses `Global` for all three axes)

Try `inspect full-classpath` (as opposed to the above example, `inspect test:full-classpath`) to get a sense of the difference. Because the configuration is omitted, it is autodetected as `compile`. `inspect compile:full-classpath` should therefore look the same as `inspect full-classpath`.

Try `inspect *:full-classpath` for another contrast. `full-classpath` is not defined in the `Global` configuration by default.

Again, for more details, see Inspecting Settings.

Referring to scopes in a build definition

If you create a setting in `build.sbt` with a bare key, it will be scoped to the current project, configuration `Global` and task `Global`:

```
name := "hello"
```

Run sbt and `inspect name` to see that it's provided by `{file:/home/hp/checkout/hello/}default-aea33a/*:name`, that is, the project is `{file:/home/hp/checkout/hello/}default-aea33a`, the configuration is `*` (meaning global), and the task is not shown (which also means global).

`build.sbt` always defines settings for a single project, so the "current project" is the project you're defining in that particular `build.sbt`. (For multi-project builds, each project has its own `build.sbt`.)

Keys have an overloaded method called `in` used to set the scope. The argument to `in` can be an instance of any of the scope axes. So for example, though there's no real reason to do this, you could set the name scoped to the `Compile` configuration:

```
name in Compile := "hello"
```

or you could set the name scoped to the `package-bin` task (pointless! just an example):

```
name in packageBin := "hello"
```

or you could set the name with multiple scope axes, for example in the `packageBin` task in the `Compile` configuration:

```
name in (Compile, packageBin) := "hello"
```

or you could use `Global` for all axes:

```
name in Global := "hello"
```

(`name in Global` implicitly converts the scope axis `Global` to a scope with all axes set to `Global`; the task and configuration are already `Global` by default, so here the effect is to make the project `Global`, that is, define `*/*:name` rather than `{file:/home/hp/checkout/hello/}default-aea33a/*:name`)

If you aren't used to Scala, a reminder: it's important to understand that `in` and `:=` are just methods, not magic. Scala lets you write them in a nicer way, but you could also use the Java style:

```
name.in(Compile).:=("hello")
```

There's no reason to use this ugly syntax, but it illustrates that these are in fact methods.

When to specify a scope

You need to specify the scope if the key in question is normally scoped. For example, the `compile` task, by default, is scoped to `Compile` and `Test` configurations, and does not exist outside of those scopes.

To change the value associated with the `compile` key, you need to write `compile in Compile` or `compile in Test`. Using plain `compile` would define a new compile task scoped to the current project, rather than overriding the standard compile tasks which are scoped to a configuration.

If you get an error like *"Reference to undefined setting"*, often you've failed to specify a scope, or you've specified the wrong scope. The key you're using may be defined in some other scope. sbt will try to suggest what you meant as part of the error message; look for "Did you mean compile:compile?"

One way to think of it is that a name is only *part* of a key. In reality, all keys consist of both a name, and a scope (where the scope has three axes). The entire expression `packageOptions in (Compile, packageBin)` is a key name, in other words. Simply `packageOptions` is also a key name, but a different one (for keys with no `in`, a scope is implicitly assumed: current project, global config, global task).

More Kinds of Setting

Getting Started Guide page 8 of 14. Next

This page explains other ways to create a `Setting`, beyond the basic `:=` method. It assumes you've read .sbt build definition and scopes.

Refresher: Settings

Remember, a build definition creates a list of `Setting`, which is then used to transform sbt's description of the build (which is a map of key-value pairs). A `Setting` is a transformation with sbt's earlier map as input and a new map as output. The new map becomes sbt's new state.

Different settings transform the map in different ways. Earlier, you read about the `:=` method.

The `Setting` which `:=` creates puts a fixed, constant value in the new, transformed map. For example, if you transform a map with the setting `name := "hello"` the new map has the string `"hello"` stored under the key `name`.

Settings must end up in the master list of settings to do any good (all lines in a `build.sbt` automatically end up in the list, but in a .scala file you can get it wrong by creating a `Setting` without putting it where sbt will find it).

Appending to previous values: += and ++=

Replacement with `:=` is the simplest transformation, but keys have other methods as well. If the `T` in `SettingKey[T]` is a sequence, i.e. the key's value type is a sequence, you can append to the sequence rather than replacing it.

- `+=` will append a single element to the sequence.
- `++=` will concatenate another sequence.

For example, the key `sourceDirectories in Compile` has a `Seq[File]` as its value. By default this key's value would include `src/main/scala`. If you wanted to also compile source code in a directory called `source` (since you just have to be nonstandard), you could add that directory:

```
sourceDirectories in Compile += new File("source")
```

Or, using the `file()` function from the sbt package for convenience:

```
sourceDirectories in Compile += file("source")
```

(`file()` just creates a new `File`.)

You could use `++=` to add more than one directory at a time:

```
sourceDirectories in Compile ++= Seq(file("sources1"), file("sources2"))
```

Where `Seq(a, b, c, ...)` is standard Scala syntax to construct a sequence.

To replace the default source directories entirely, you use `:=` of course:

```
sourceDirectories in Compile := Seq(file("sources1"), file("sources2"))
```

Transforming a value: ~=

What happens if you want to *prepend* to `sourceDirectories in Compile`, or filter out one of the default directories?

You can create a `Setting` that depends on the previous value of a key.

- `~=` applies a function to the setting's previous value, producing a new value of the same type.

To modify `sourceDirectories in Compile`, you could use `~=` as follows:

```
// filter out src/main/scala
sourceDirectories in Compile ~= { srcDirs => srcDirs filter(!
_.getAbsolutePath.endsWith("src/main/scala")) }
```

Here, `srcDirs` is a parameter to an anonymous function, and the old value of `sourceDirectories in Compile` gets passed in to the anonymous function. The result of this function becomes the new value of `sourceDirectories in Compile`.

Or a simpler example:

```
// make the project name upper case
name ~= { _.toUpperCase }
```

The function you pass to the `~=` method will always have type `T => T`, if the key has type `SettingKey[T]` or `TaskKey[T]`. The function transforms the key's value into another value of the same type.

Computing a value based on other keys' values: <<=

`~=` defines a new value in terms of a key's previously-associated value. But what if you want to define a value in terms of *other* keys' values?

- `<<=` lets you compute a new value using the value(s) of arbitrary other keys.

`<<=` has one argument, of type `Initialize[T]`. An `Initialize[T]` instance is a computation which takes the values associated with a set of keys as input, and returns a value of type `T` based on those other values. It initializes a value of type `T`.

Given an `Initialize[T]`, `<<=` returns a `Setting[T]`, of course (just like `:=`, `+=`, `~=`, etc.).

Trivial `Initialize[T]`: depending on one other key with <<=

All keys extend the `Initialize` trait already. So the simplest `Initialize` is just a key:

```
// useless but valid
name <<= name
```

When treated as an `Initialize[T]`, a `SettingKey[T]` computes its current value. So `name <<= name` sets the value of `name` to the value that `name` already had.

It gets a little more useful if you set a key to a *different* key. The keys must have identical value types, though.

```
// name our organization after our project (both are SettingKey[String])
organization <<= name
```

(Note: this is how you alias one key to another.)

If the value types are not identical, you'll need to convert from `Initialize[T]` to another type, like `Initialize[S]`. This is done with the `apply` method on `Initialize`, like this:

```
// name is a Key[String], baseDirectory is a Key[File]
// name the project after the directory it's inside
name <<= baseDirectory.apply(_.getName)
```

`apply` is special in Scala and means you can invoke the object with function syntax; so you could also write this:

```
name <<= baseDirectory(_.getName)
```

That transforms the value of `baseDirectory` using the function `_.getName`, where the function `_.getName` takes a `File` and returns a `String`. `getName` is a method on the standard `java.io.File` object.

Settings with dependencies

In the setting `name <<= baseDirectory(_.getName)`, `name` will have a *dependency* on `baseDirectory`. If you place the above in `build.sbt` and run the sbt interactive console, then type `inspect name`, you should see (in part):

```
[info] Dependencies:
[info]      *:base-directory
```

This is how sbt knows which settings depend on which other settings. Remember that some settings describe tasks, so this approach also creates dependencies between tasks.

For example, if you `inspect compile` you'll see it depends on another key `compile-inputs`, and if you inspect `compile-inputs` it in turn depends on other keys. Keep following the dependency chains and magic happens. When you type `compile` sbt automatically performs an `update`, for example. It Just Works because the values required as inputs to the `compile` computation require sbt to do the `update` computation first.

In this way, all build dependencies in sbt are *automatic* rather than explicitly declared. If you use a key's value in another computation, then the computation depends on that key. It just works!

Complex `Initialize[T]`: depending on multiple keys with <<=

To support dependencies on multiple other keys, sbt adds `apply` and `identity` methods to tuples of `Initialize` objects. In Scala, you write a tuple like `(1, "a")` (that one has type `(Int, String)`).

So say you have a tuple of three `Initialize` objects; its type would be `(Initialize[A], Initialize[B], Initialize[C])`. The `Initialize` objects could be keys, since all `SettingKey[T]` are also instances of `Initialize[T]`.

Here's a simple example, in this case all three keys are strings:

```
// a tuple of three SettingKey[String], also a tuple of three Initialize[String]
(name, organization, version)
```

The `apply` method on a tuple of `Initialize` takes a function as its argument. Using each `Initialize` in the tuple, sbt computes a corresponding value (the current value of the key). These values are passed in to the function. The function then returns *one* value, which is wrapped up in a new `Initialize`. If you wrote it out with explicit types (Scala does not require this), it would look like:

```
val tuple: (Initialize[String], Initialize[String], Initialize[String]) = (name,
organization, version)
val combined: Initialize[String] = tuple.apply({ (n, o, v) =>
    "project " + n + " from " + o + " version " + v })
val setting: Setting[String] = name <<= combined
```

So each key is already an `Initialize`; but you can combine up to nine simple `Initialize` (such as keys) into one composite `Initialize` by placing them in tuples, and invoking the `apply` method.

The `<<=` method on `SettingKey[T]` is expecting an `Initialize[T]`, so you can use this technique to create an `Initialize[T]` with multiple dependencies on arbitrary keys.

Because function syntax in Scala just calls the `apply` method, you could write the code like this, omitting the explicit `.apply` and just treating `tuple` as a function:

```
val tuple: (Initialize[String], Initialize[String], Initialize[String]) = (name,
organization, version)
val combined: Initialize[String] = tuple({ (n, o, v) =>
    "project " + n + " from " + o + " version " + v })
val setting: Setting[String] = name <<= combined
```

In a `build.sbt`, this code using intermediate `val` will not work, since you can only write single expressions in a `.sbt` file, not multiple statements.

You can use a more concise syntax in `build.sbt`, like this:

```
name <<= (name, organization, version) { (n, o, v) => "project " + n + " from " + o +
" version " + v }
```

Here the tuple of `Initialize` (also a tuple of `SettingKey`) works as a function, taking the anonymous function delimited by `{}` as its argument, and returning an `Initialize[T]` where `T` is the result type of the anonymous function.

Tuples of `Initialize` have one other method, `identity`, which simply returns an `Initialize` with a tuple value. `(a: Initialize[A], b: Initialize[B]).identity` would result in a value of type `Initialize[(A, B)]`. `identity` combines two `Initialize` into one, without losing or modifying any of the values.

When settings are undefined

Whenever a setting uses `~=` or `<<=` to create a dependency on itself or another key's value, the value it depends on must exist. If it does not, sbt will complain. It might say *"Reference to undefined setting"*, for example. When this happens, be sure you're using the key in the scope that defines it.

It's possible to create cycles, which is an error; sbt will tell you if you do this.

Tasks with dependencies

As noted in .sbt build definition, task keys create a `Setting[Task[T]]` rather than a `Setting[T]` when you build a setting with `:=`, `<<=`, etc. Similarly, task keys are instances of `Initialize[Task[T]]` rather than `Initialize[T]`, and `<<=` on a task key takes an `Initialize[Task[T]]` parameter.

The practical importance of this is that you can't have tasks as dependencies for a non-task setting.

Take these two keys (from Keys):

```
val scalacOptions = TaskKey[Seq[String]]("scalac-options", "Options for the Scala
compiler.")
val checksums = SettingKey[Seq[String]]("checksums", "The list of checksums to
generate and to verify for dependencies.")
```

(`scalacOptions` and `checksums` have nothing to do with each other, they are just two keys with the same value type, where one is a task.)

You cannot compile a `build.sbt` that tries to alias one of these to the other like this:

```
scalacOptions <<= checksums

checksums <<= scalacOptions
```

The issue is that `scalacOptions.<<=` expects an `Initialize[Task[Seq[String]]]` and `checksums.<<=` expects an `Initialize[Seq[String]]`. There is, however, a way to convert an `Initialize[T]` to an `Initialize[Task[T]]`, called `map`:

```
scalacOptions <<= checksums map identity
```

(`identity` is a standard Scala function that returns its input as its result.)

There is no way to go the *other* direction, that is, a setting key can't depend on a task key. That's because a setting key is only computed once on project load, so the task would not be re-run every time, and tasks expect to re-run every time.

A task can depend on both settings and other tasks, though, just use `map` rather than `apply` to build an `Initialize[Task[T]]` rather than an `Initialize[T]`. Remember the usage of `apply` with a non-task setting looks like this:

```
name <<= (name, organization, version) { (n, o, v) => "project " + n + " from " + o +
" version " + v }
```

(`(name, organization, version)` has an apply method and is thus a function, taking the anonymous function in `{}` braces as a parameter.)

To create an `Initialize[Task[T]]` you need a `map` in there rather than `apply`:

```
// this WON'T compile because name (on the left of <<=) is not a task and we used map
name <<= (name, organization, version) map { (n, o, v) => "project " + n + " from " +
o + " version " + v }

// this WILL compile because packageBin is a task and we used map
packageBin in Compile <<= (name, organization, version) map { (n, o, v) => file(o +
"-" + n + "-" + v + ".jar") }

// this WILL compile because name is not a task and we used apply
name <<= (name, organization, version) { (n, o, v) => "project " + n + " from " + o +
" version " + v }

// this WON'T compile because packageBin is a task and we used apply
packageBin in Compile <<= (name, organization, version) { (n, o, v) => file(o + "-" +
n + "-" + v + ".jar") }
```

Bottom line: when converting a tuple of keys into an `Initialize[Task[T]]`, use `map`; when converting a tuple of keys into an `Initialize[T]` use `apply`; and you need the `Initialize[Task[T]]` if the key on the left side of `<<=` is a `TaskKey[T]` rather than a `SettingKey[T]`.

Remember, aliases use <<= not :=

If you want one key to be an alias for another, you might be tempted to use `:=` to create the following nonsense alias:

```
// doesn't work, and not useful
packageBin in Compile := packageDoc in Compile
```

The problem is that `:=`'s argument must be a value (or for tasks, a function returning a value). For `packageBin` which is a `TaskKey[File]`, it must be a `File` or a function `=> File`. `packageDoc` is not a `File`, it's a key.

The proper way to do this is with `<<=`, which takes a key (really an `Initialize`, but keys are instances of `Initialize`):

```
// works, still not useful
packageBin in Compile <<= packageDoc in Compile
```

Here, `<<=` expects an `Initialize[Task[File]]`, which is a computation that will return a file later, when sbt runs the task. Which is what you want: you want to alias a task by making it run another task, not by setting it one time when sbt loads the project.

(By the way: the `in Compile` scope is needed to avoid "undefined" errors, because the packaging tasks like `packageBin` are per-configuration, not global.)

Appending with dependencies: <+= and <++=

There are a couple more methods for appending to lists, which combine `+=` and `++=` with `<<=`. That is, they let you compute a new list element or new list to concatenate, using dependencies on other keys in order to do so.

These methods work exactly like `<<=`, but for `<++=`, the function you write to convert the dependencies' values into a new value should create a `Seq[T]` instead of a `T`.

Unlike `<<=` of course, `<+=` and `<++=` will append to the previous value of the key on the left, rather than replacing it.

For example, say you have a coverage report named after the project, and you want to add it to the files removed by `clean`:

```
cleanFiles <+= (name) { n => file("coverage-report-" + n + ".txt") }
```

Getting Started Library Dependencies

Getting Started Guide page 9 of 14. Next

This page assumes you've read the earlier Getting Started pages, in particular .sbt build definition, scopes, and more about settings.

Library dependencies can be added in two ways:

- *unmanaged dependencies* are jars dropped into the `lib` directory
- *managed dependencies* are configured in the build definition and downloaded automatically from repositories

Unmanaged dependencies

Most people use managed dependencies instead of unmanaged. But unmanaged can be simpler when starting out.

Unmanaged dependencies work like this: add jars to `lib` and they will be placed on the project classpath. Not much else to it!

You can place test jars such as ScalaCheck, specs, and ScalaTest in `lib` as well.

Dependencies in `lib` go on all the classpaths (for `compile`, `test`, `run`, and `console`). If you wanted to change the classpath for just one of those, you would adjust `dependencyClasspath in Compile` or `dependencyClasspath in Runtime` for example. You could use `~=` to get the previous classpath value, filter some entries out, and return a new classpath value. See more about settings for details of `~=`.

There's nothing to add to `build.sbt` to use unmanaged dependencies, though you could change the `unmanaged-base` key if you'd like to use a different directory rather than `lib`.

To use `custom_lib` instead of `lib`:

```
unmanagedBase <<= baseDirectory { base => base / "custom_lib" }
```

`baseDirectory` is the project's root directory, so here you're changing `unmanagedBase` depending on `baseDirectory`, using `<<=` as explained in more about settings.

There's also an `unmanaged-jars` task which lists the jars from the `unmanaged-base` directory. If you wanted to use multiple directories or do something else complex, you might need to replace the whole `unmanaged-jars` task with one that does something else.

Managed Dependencies

sbt uses Apache Ivy to implement managed dependencies, so if you're familiar with Maven or Ivy, you won't have much trouble.

The libraryDependencies key

Most of the time, you can simply list your dependencies in the setting `libraryDependencies`. It's also possible to write a Maven POM file or Ivy configuration file to externally configure your dependencies, and have sbt use those external configuration files. You can learn more about that here.

Declaring a dependency looks like this, where `groupId`, `artifactId`, and `revision` are strings:

```
libraryDependencies += groupID % artifactID % revision
```

or like this, where `configuration` is also a string:

```
libraryDependencies += groupID % artifactID % revision % configuration
```

`libraryDependencies` is declared in Keys like this:

```
val libraryDependencies = SettingKey[Seq[ModuleID]]("library-dependencies", "Declares
managed dependencies.")
```

The `%` methods create `ModuleID` objects from strings, then you add those `ModuleID` to `libraryDependencies`.

Of course, sbt (via Ivy) has to know where to download the module. If your module is in one of the default repositories sbt comes with, this will just work. For example, Apache Derby is in a default repository:

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

If you type that in `build.sbt` and then `update`, sbt should download Derby to `~/.ivy2/cache/org.apache.derby/`. (By the way, `update` is a dependency of `compile` so there's no need to manually type `update` most of the time.)

Of course, you can also use `++=` to add a list of dependencies all at once:

```
libraryDependencies ++= Seq(
      groupID % artifactID % revision,
      groupID % otherID % otherRevision
)
```

And in rare cases you might find reasons to use `:=`, `<<=`, `<+=`, etc. with `libraryDependencies` as well.

Getting the right Scala version with `%%`

If you use `groupID %% artifactID % revision` rather than `groupID % artifactID % revision` (the difference is the double `%%` after the groupID), sbt will add your project's Scala version to the artifact name. This is just a shortcut. You could write this without the `%%`:

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.9.1" % "0.3"
```

Assuming the `scalaVersion` for your build is `2.9.1`, the following is identical:

```
libraryDependencies += "org.scala-tools" %% "scala-stm" % "0.3"
```

The idea is that many dependencies are compiled for multiple Scala versions, and you'd like to get the one that matches your project.

The complexity in practice is that often a dependency will work with a slightly different Scala version; but `%%` is not smart about that. So if the dependency is available for `2.9.0` but you're using `scalaVersion := "2.9.1"`, you won't be able to use `%%` even though the `2.9.0` dependency likely works. If `%%` stops working just go see which versions the dependency is really built for, and hardcode the one you think will work (assuming there is one).

See Cross Build for some more detail on this.

Ivy revisions

The `revision` in `groupID % artifactID % revision` does not have to be a single fixed version. Ivy can select the latest revision of a module according to constraints you specify. Instead of a fixed revision like `"1.6.1"`, you specify `"latest.integration"`, `"2.9.+"`, or `"[1.0,)"`. See the Ivy revisions documentation for details.

Resolvers

Not all packages live on the same server; sbt uses the standard Maven2 repository by default. If your dependency isn't on one of the default repositories, you'll have to add a *resolver* to help Ivy find it.

To add an additional repository, use

```
resolvers += name at location
```

For example:

```
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/
repositories/snapshots"
```

The `resolvers` key is defined in Keys like this:

```
val resolvers = SettingKey[Seq[Resolver]]("resolvers", "The user-defined additional
resolvers for automatically managed dependencies.")
```

The `at` method creates a `Resolver` object from two strings.

sbt can search your local Maven repository if you add it as a repository:

```
resolvers += "Local Maven Repository" at "file://"+Path.userHome.absolutePath+"/.m2/
repository"
```

See Resolvers for details on defining other types of repositories.

Overriding default resolvers

`resolvers` does not contain the default resolvers; only additional ones added by your build definition.

`sbt` combines `resolvers` with some default repositories to form `external-resolvers`.

Therefore, to change or remove the default resolvers, you would need to override `external-resolvers` instead of `resolvers`.

Per-configuration dependencies

Often a dependency is used by your test code (in `src/test/scala`, which is compiled by the `Test` configuration) but not your main code.

If you want a dependency to show up in the classpath only for the `Test` configuration and not the `Compile` configuration, add `% "test"` like this:

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3" % "test"
```

Now, if you type `show compile:dependency-classpath` at the sbt interactive prompt, you should not see derby. But if you type `show test:dependency-classpath`, you should see the derby jar in the list.

Typically, test-related dependencies such as ScalaCheck, specs, and ScalaTest would be defined with `% "test"`.

.scala Build Definition

Getting Started Guide page 10 of 14. Next

This page assumes you've read previous pages in the Getting Started Guide, *especially* .sbt build definition and more about settings.

sbt is recursive

`build.sbt` is so simple, it conceals how sbt really works. sbt builds are defined with Scala code. That code, itself, has to be built. What better way than with sbt?

The `project` directory *is another project inside your project* which knows how to build your project. The project inside `project` can (in theory) do anything any other project can do. *Your build definition is an sbt project.*

And the turtles go all the way down. If you like, you can tweak the build definition of the build definition project, by creating a `project/project/` directory.

Here's an illustration.

```
hello/                  # your project's base directory

    Hello.scala         # a source file in your project (could be in
                        #   src/main/scala too)

    build.sbt           # build.sbt is part of the source code for the
                        #   build definition project inside project/

    project/            # base directory of the build definition project

        Build.scala     # a source file in the project/ project,
                        #   that is, a source file in the build definition

        build.sbt       # this is part of a build definition for a project
                        #   in project/project ; build definition's build
                        #   definition


        project/        # base directory of the build definition project
                        #   for the build definition

            Build.scala # source file in the project/project/ project
```

Don't worry! Most of the time you are not going to need all that. But understanding the principle can be helpful.

By the way: any time files ending in `.scala` or `.sbt` are used, naming them `build.sbt` and `Build.scala` are conventions only. This also means that multiple files are allowed.

`.scala` source files in the build definition project

`.sbt` files are merged into their sibling `project` directory. Looking back at the project layout:

```
hello/                  # your project's base directory

    build.sbt           # build.sbt is part of the source code for the
                        #   build definition project inside project/

    project/            # base directory of the build definition project

        Build.scala     # a source file in the project/ project,
                        #   that is, a source file in the build definition
```

The Scala expressions in `build.sbt` are compiled alongside and merged with `Build.scala` (or any other `.scala` files in the `project/` directory).

`.sbt` files in the base directory for a project become part of the `project` build definition project also located in that base directory.

The `.sbt` file format is a convenient shorthand for adding settings to the build definition project.

Relating build.sbt to Build.scala

To mix `.sbt` and `.scala` files in your build definition, you need to understand how they relate.

The following two files illustrate. First, if your project is in `hello`, create `hello/project/Build.scala` as follows:

```
import sbt._
import Keys._

object HelloBuild extends Build {

    val sampleKeyA = SettingKey[String]("sample-a", "demo key A")
    val sampleKeyB = SettingKey[String]("sample-b", "demo key B")
    val sampleKeyC = SettingKey[String]("sample-c", "demo key C")
    val sampleKeyD = SettingKey[String]("sample-d", "demo key D")

    override lazy val settings = super.settings ++
        Seq(sampleKeyA := "A: in Build.settings in Build.scala", resolvers := Seq())

    lazy val root = Project(id = "hello",
                            base = file("."),
                            settings = Project.defaultSettings ++ Seq(sampleKeyB :=
"B: in the root project settings in Build.scala"))
}
```

Now, create `hello/build.sbt` as follows:

```
sampleKeyC in ThisBuild := "C: in build.sbt scoped to ThisBuild"

sampleKeyD := "D: in build.sbt"
```

Start up the sbt interactive prompt. Type `inspect sample-a` and you should see (among other things):

```
[info] Setting: java.lang.String = A: in Build.settings in Build.scala
[info] Provided by:
[info]      {file:/home/hp/checkout/hello/}/*:sample-a
```

and then `inspect sample-c` and you should see:

```
[info] Setting: java.lang.String = C: in build.sbt scoped to ThisBuild
[info] Provided by:
[info]      {file:/home/hp/checkout/hello/}/*:sample-c
```

Note that the "Provided by" shows the same scope for the two values. That is, `sampleKeyC in ThisBuild` in a `.sbt` file is equivalent to placing a setting in the `Build.settings` list in a `.scala` file. sbt takes build-scoped settings from both places to create the build definition.

Now, `inspect sample-b`:

```
[info] Setting: java.lang.String = B: in the root project settings in Build.scala
[info] Provided by:
[info]      {file:/home/hp/checkout/hello/}hello/*:sample-b
```

Note that `sample-b` is scoped to the project (`{file:/home/hp/checkout/hello/}hello`) rather than the entire build (`{file:/home/hp/checkout/hello/}`).

As you've probably guessed, `inspect sample-d` matches `sample-b`:

```
[info] Setting: java.lang.String = D: in build.sbt
[info] Provided by:
[info]      {file:/home/hp/checkout/hello/}hello/*:sample-d
```

sbt *appends* the settings from `.sbt` files to the settings from `Build.settings` and `Project.settings` which means `.sbt` settings take precedence. Try changing `Build.scala` so it sets key `sample-c` or `sample-d`, which are also set in `build.sbt`. The setting in `build.sbt` should "win" over the one in `Build.scala`.

One other thing you may have noticed: `sampleKeyC` and `sampleKeyD` were available inside `build.sbt`. That's because sbt imports the contents of your `Build` object into your `.sbt` files. In this case `import HelloBuild._` was implicitly done for the `build.sbt` file.

In summary:

- In `.scala` files, you can add settings to `Build.settings` for sbt to find, and they are automatically build-scoped.
- In `.scala` files, you can add settings to `Project.settings` for sbt to find, and they are automatically project-scoped.

- Any `Build` object you write in a `.scala` file will have its contents imported and available to `.sbt` files.
- The settings in `.sbt` files are *appended* to the settings in `.scala` files.
- The settings in `.sbt` files are project-scoped unless you explicitly specify another scope.

When to use `.scala` files

In `.scala` files, you are not limited to a series of settings expressions. You can write any Scala code including `val`, `object`, and method definitions.

One recommended approach is to define settings in `.sbt` files, using `.scala` files when you need to factor out a `val` or `object` or method definition.

Because the `.sbt` format allows only single expressions, it doesn't give you a way to share code among expressions. When you need to share code, you need a `.scala` file so you can set common variables or define methods.

There's one build definition, which is a nested project inside your main project. `.sbt` and `.scala` files are compiled together to create that single definition.

`.scala` files are also required to define multiple projects in a single build. More on that is coming up in Multi-Project Builds.

(A disadvantage of using `.sbt` files in a multi-project build is that they'll be spread around in different directories; for that reason, some people prefer to put settings in their `.scala` files if they have sub-projects. This will be clearer after you see how multi-project builds work.)

The build definition project in interactive mode

You can switch the sbt interactive prompt to have the build definition project in `project/` as the current project. To do so, type `reload plugins`.

```
> reload plugins
[info] Set current project to default-a0e8e4 (in build file:/home/hp/checkout/hello/
project/)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/project/Build.scala)
> reload return
[info] Loading project definition from /home/hp/checkout/hello/project
[info] Set current project to hello (in build file:/home/hp/checkout/hello/)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/hw.scala)
>
```

As shown above, you use `reload return` to leave the build definition project and return to your regular project.

Reminder: it's all immutable

It would be wrong to think that the settings in `build.sbt` are added to the `settings` fields in `Build` and `Project` objects. Instead, the settings list from `Build` and `Project`, and the settings from `build.sbt`, are concatenated into another immutable list which is then used by sbt. The `Build` and `Project` objects are "immutable configuration" forming only part of the complete build definition.

In fact, there are other sources of settings as well. They are appended in this order:

- Settings from `Build.settings` and `Project.settings` in your `.scala` files.
- Your user-global settings; for example in `~/.sbt/build.sbt` you can define settings affecting *all* your projects.
- Settings injected by plugins, see using plugins coming up next.
- Settings from `.sbt` files in the project.
- Build definition projects (i.e. projects inside `project`) have settings from global plugins (`~/.sbt/plugins`) added. Using plugins explains this more.

Later settings override earlier ones. The entire list of settings forms the build definition.

Using Plugins

Getting Started Guide page 11 of 14. Next

Please read the earlier pages in the Getting Started Guide first, in particular you need to understand build.sbt, library dependencies, and .scala build definition before reading this page.

What is a plugin?

A plugin extends the build definition, most commonly by adding new settings. The new settings could be new tasks. For example, a plugin could add a `code-coverage` task which would generate a test coverage report.

Adding a plugin

The short answer

If your project is in directory `hello`, edit `hello/project/build.sbt` and add the plugin location as a resolver, then call `addSbtPlugin` with the plugin's Ivy module ID:

```
resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.0.0")
```

If the plugin were located on one of the default repositories, you wouldn't have to add a resolver, of course.

So that's how you do it... read on to understand what's going on.

How it works

Be sure you understand the recursive nature of sbt projects described earlier and how to add a managed dependency.

Dependencies for the build definition

Adding a plugin means *adding a library dependency to the build definition*. To do that, you edit the build definition for the build definition.

Recall that for a project `hello`, its build definition project lives in `hello/*.sbt` and `hello/project/*.scala`:

```
 hello/                     # your project's base directory

     build.sbt              # build.sbt is part of the source code for the
                            #   build definition project inside project/

     project/               # base directory of the build definition project

         Build.scala        # a source file in the project/ project,
                            #   that is, a source file in the build definition
```

If you wanted to add a managed dependency to project `hello`, you would add to the `libraryDependencies` setting either in `hello/*.sbt` or `hello/project/*.scala`.

You could add this in `hello/build.sbt`:

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3" % "test"
```

If you add that and start up the sbt interactive mode and type `show dependency-classpath`, you should see the derby jar on your classpath.

To add a plugin, do the same thing but recursed one level. We want the *build definition project* to have a new dependency. That means changing the `libraryDependencies` setting for the build definition of the build definition.

The build definition of the build definition, if your project is `hello`, would be in `hello/project/*.sbt` and `hello/project/project/*.scala`.

The simplest "plugin" has no special sbt support; it's just a jar file. For example, edit `hello/project/build.sbt` and add this line:

```
libraryDependencies += "net.liftweb" % "lift-json" % "2.0"
```

Now, at the sbt interactive prompt, `reload plugins` to enter the build definition project, and try `show dependency-classpath`. You should see the lift-json jar on the classpath. This means: you could use classes from lift-json in your `Build.scala` or `build.sbt` to implement a task. You could parse a JSON file and generate other files based on it, for example. Remember, use `reload return` to leave the build definition project and go back to the parent project.

(Stupid sbt trick: type `reload plugins` over and over. You'll find yourself in the project rooted in `project/project/project/project/project/project/`. Don't worry, it isn't useful. Also, it creates `target` directories all the way down, which you'll have to clean up.)

addSbtPlugin

`addSbtPlugin` is just a convenience method. Here's its definition:

```
def addSbtPlugin(dependency: ModuleID): Setting[Seq[ModuleID]] =
  libraryDependencies <+= (sbtVersion in update,scalaVersion) { (sbtV, scalaV) =>
    sbtPluginExtra(dependency, sbtV, scalaV)
  }
```

Remember from more about settings that `<+=` combines `<<=` and `+=`, so this builds a value based on other settings, and then appends it to `libraryDependencies`. The value is based on `sbtVersion in update` (sbt's version scoped to the `update` task) and `scalaVersion` (the version of scala used to compile the project, in this case used to compile the build definition). `sbtPluginExtra` adds the sbt and Scala version information to the module ID.

plugins.sbt

Some people like to list plugin dependencies (for a project `hello`) in `hello/project/plugins.sbt` to avoid confusion with `hello/build.sbt`. sbt does not care what `.sbt` files are called, so both `build.sbt` and `project/plugins.sbt` are conventions. sbt *does* of course care where the sbt files are *located*. `hello/*.sbt` would contain dependencies for `hello` and `hello/project/*.sbt` would contain dependencies for `hello`'s build definition.

Plugins can add settings and imports automatically

In one sense a plugin is just a jar added to `libraryDependencies` for the build definition; you can then use the jar from build definition code as in the lift-json example above.

However, jars intended for use as sbt plugins can do more.

If you download a plugin jar (here's one for sbteclipse) and unpack it with `jar xf`, you'll see that it contains a text file `sbt/sbt.plugins`. In `sbt/sbt.plugins` there's an object name on each line like this:

```
com.typesafe.sbteclipse.SbtEclipsePlugin
```
`com.typesafe.sbteclipse.SbtEclipsePlugin` is the name of an object that extends `sbt.Plugin`. The `sbt.Plugin` trait is very simple:

```
trait Plugin {
  def settings: Seq[Setting[_]] = Nil
}
```

sbt looks for objects listed in `sbt/sbt.plugins`. When it finds `com.typesafe.sbteclipse.SbtEclipsePlugin`, it adds `com.typesafe.sbteclipse.SbtEclipsePlugin.settings` to the settings for the project. It also does `import com.typesafe.sbteclipse.SbtEclipsePlugin._` for any `.sbt` files, allowing a plugin to provide values, objects, and methods to `.sbt` files in the build definition.

Adding settings manually from a plugin

If a plugin defines settings in the `settings` field of a `Plugin` object, you don't have to do anything to add them.

However, plugins often avoid this because you could not control which projects in a multi-project build would use the plugin.

sbt provides a method called `seq` which adds a whole batch of settings at once. So if a plugin has something like this:

```
object MyPlugin extends Plugin {
   val myPluginSettings = Seq(settings in here)
}
```

You could add all those settings in `build.sbt` with this syntax:

```
seq(myPluginSettings: _*)
```
If you aren't familiar with the `_*` syntax:

- `seq` is defined with a variable number of arguments: `def seq(settings: Setting[_]*)`
- `_*` converts a sequence into a variable argument list

Short version: `seq(myPluginSettings: _*)` in a `build.sbt` adds all the settings in `myPluginSettings` to the project.

Creating a plugin

After reading this far, you pretty much know how to *create* an sbt plugin as well. There's one trick to know; set `sbtPlugin := true` in `build.sbt`. If `sbtPlugin` is true, the project will scan its compiled classes for instances of `Plugin`, and list them in `sbt/sbt.plugins` when it packages a jar. `sbtPlugin := true` also adds sbt to the project's classpath, so you can use sbt APIs to implement your plugin.

Learn more about creating a plugin at Plugins and Plugins Best Practices.

Global plugins

Plugins can be installed for all your projects at once by dropping them in `~/.sbt/plugins/`. `~/.sbt/plugins/` is an sbt project whose classpath is exported to all sbt build definition projects. Roughly speaking, any `.sbt` files in `~/.sbt/plugins/` behave as if they were in the `project/` directory for all projects, and any `.scala` files in `~/.sbt/plugins/project/` behave as if they were in the `project/project/` directory for all projects.

You can create `~/.sbt/plugins/build.sbt` and put `addSbtPlugin()` expressions in there to add plugins to all your projects at once.

Available Plugins

There's a list of available plugins.

Some especially popular plugins are:

- those for IDEs (to import an sbt project into your IDE)
- those supporting web frameworks, such as xsbt-web-plugin.

Check out the list.

Multi-Project Builds

Getting Started Guide page 12 of 14. Next

This page introduces multiple projects in a single build.

Please read the earlier pages in the Getting Started Guide first, in particular you need to understand build.sbt and .scala build definition before reading this page.

Multiple projects

It can be useful to keep multiple related projects in a single build, especially if they depend on one another and you tend to modify them together.

Each sub-project in a build has its own `src/main/scala`, generates its own jar file when you run `package`, and in general works like any other project.

Defining projects in a `.scala` file

To have multiple projects, you must declare each project and how they relate in a `.scala` file; there's no way to do it in a `.sbt` file. However, you can define settings for each project in `.sbt` files. Here's an example of a `.scala` file which defines a root project `hello`, where the root project aggregates two sub-projects, `hello-foo` and `hello-bar`:

```
import sbt._
import Keys._

object HelloBuild extends Build {
    lazy val root = Project(id = "hello",
                            base = file(".")) aggregate(foo, bar)

    lazy val foo = Project(id = "hello-foo",
                           base = file("foo"))

    lazy val bar = Project(id = "hello-bar",
                           base = file("bar"))
}
```

sbt finds the list of `Project` objects using reflection, looking for fields with type `Project` in the `Build` object.

Because project `hello-foo` is defined with `base = file("foo")`, it will be contained in the subdirectory `foo`. Its sources could be directly under `foo`, like `foo/Foo.scala`, or in `foo/src/main/scala`. The usual sbt directory structure applies underneath `foo` with the exception of build definition files.

Any `.sbt` files in `foo`, say `foo/build.sbt`, will be merged with the build definition for the entire build, but scoped to the `hello-foo` project.

If your whole project is in `hello`, try defining a different version (`version := "0.6"`) in `hello/build.sbt`, `hello/foo/build.sbt`, and `hello/bar/build.sbt`. Now `show version` at the sbt interactive prompt. You should get something like this (with whatever versions you defined):

```
> show version
[info] hello-foo/*:version
[info]      0.7
[info] hello-bar/*:version
[info]      0.9
[info] hello/*:version
[info]      0.5
```

`hello-foo/*:version` was defined in `hello/foo/build.sbt`, `hello-bar/*:version` was defined in `hello/bar/build.sbt`, and `hello/*:version` was defined in `hello/build.sbt`. Remember the syntax for scoped keys. Each `version` key is scoped to a project, based on the location of the `build.sbt`. But all three `build.sbt` are part of the same build definition.

Each project's settings can go in `.sbt` files in the base directory of that project, while the `.scala` file can be as simple as the one shown above, listing the projects and base directories. *There is no need to put settings in the `.scala` file.*

You may find it cleaner to put everything including settings in `.scala` files in order to keep all build definition under a single `project` directory, however. It's up to you.

You cannot have a `project` subdirectory or `project/*.scala` files in the sub-projects. `foo/project/Build.scala` would be ignored.

Aggregation

Projects in the build can be completely independent of one another, if you want.

In the above example, however, you can see the method call `aggregate(foo, bar)`. This aggregates `hello-foo` and `hello-bar` underneath the root project.

Aggregation means that running a task on the aggregate project will also run it on the aggregated projects. Start up sbt with two subprojects as in the example, and try `compile`. You should see that all three projects are compiled.

In the project doing the aggregating, the root `hello` project in this case, you can control aggregation per-task. So for example in `hello/build.sbt` you could avoid aggregating the `update` task:

```
aggregate in update := false
```

`aggregate in update` is the `aggregate` key scoped to the `update` task, see scopes.

Note: aggregation will run the aggregated tasks in parallel and with no defined ordering.

Classpath dependencies

A project may depend on code in another project. This is done by adding a `dependsOn` method call. For example, if `hello-foo` needed `hello-bar` on its classpath, you would write in your `Build.scala`:

```
lazy val foo = Project(id = "hello-foo",
                       base = file("foo")) dependsOn(bar)
```

Now code in `hello-foo` can use classes from `hello-bar`. This also creates an ordering between the projects when compiling them; `hello-bar` must be updated and compiled before `hello-foo` can be compiled.

To depend on multiple projects, use multiple arguments to `dependsOn`, like `dependsOn(bar, baz)`.

Per-configuration classpath dependencies

`foo dependsOn(bar)` means that the `Compile` configuration in `foo` depends on the `Compile` configuration in `bar`. You could write this explicitly as `dependsOn(bar % "compile->compile")`.

The `->` in `"compile->compile"` means "depends on" so `"test->compile"` means the `Test` configuration in `foo` would depend on the `Compile` configuration in `bar`.

Omitting the `->config` part implies `->compile`, so `dependsOn(bar % "test")` means that the `Test` configuration in `foo` depends on the `Compile` configuration in `bar`.

A useful declaration is `"test->test"` which means `Test` depends on `Test`. This allows you to put utility code for testing in `bar/src/test/scala` and then use that code in `foo/src/test/scala`, for example.

You can have multiple configurations for a dependency, separated by semicolons. For example, `dependsOn (bar % "test->test;compile->compile")`.

Navigating projects interactively

At the sbt interactive prompt, type `projects` to list your projects and `project <projectname>` to select a current project. When you run a task like `compile`, it runs on the current project. So you don't necessarily have to compile the root project, you could compile only a subproject.

Custom Settings and Tasks

Getting Started Guide page 13 of 14. Next

This page gets you started creating your own settings and tasks.

To understand this page, be sure you've read earlier pages in the Getting Started Guide, especially build.sbt and more about settings.

Defining a key

Keys is packed with examples illustrating how to define keys. Most of the keys are implemented in Defaults.

Keys have one of three types. `SettingKey` and `TaskKey` are described in .sbt build definition. Read about `InputKey` on the Input Tasks page.

Some examples from Keys:

```
val scalaVersion = SettingKey[String]("scala-version", "The version of Scala used for
building.")
val clean = TaskKey[Unit]("clean", "Deletes files produced by the build, such as
generated sources, compiled classes, and task caches.")
```

The key constructors have two string parameters: the name of the key (`"scala-version"`) and a documentation string (`"The version of scala used for building."`).

Remember from .sbt build definition that the type parameter `T` in `SettingKey[T]` indicates the type of value a setting has. `T` in `TaskKey[T]` indicates the type of the task's result. Also remember from .sbt build definition that a setting has a fixed value until project reload, while a task is re-computed for every "task execution" (every time someone types a command at the sbt interactive prompt or in batch mode).

Keys may be defined in a `.scala` file (as described in .scala build definition), or in a plugin (as described in using plugins). Any `val` found in a `Build` object in your `.scala` build definition files, or any `val` found in a `Plugin` object from a plugin, will be imported automatically into your `.sbt` files.

Implementing a task

Once you've defined a key, you'll need to use it in some task. You could be defining your own task, or you could be planning to redefine an existing task. Either way looks the same; if the task has no dependencies on other settings or tasks, use `:=` to associate some code with the task key:

```
sampleStringTask := System.getProperty("user.home")

sampleIntTask := {
  val sum = 1 + 2
  println("sum: " + sum)
  sum
}
```

If the task has dependencies, you'd use `<<=` instead of course, as discussed in more about settings.

The hardest part about implementing tasks is often not sbt-specific; tasks are just Scala code. The hard part could be writing the "meat" of your task that does whatever you're trying to do. For example, maybe you're trying to format HTML in which case you might want to use an HTML library (you would add a library dependency to your build definition and write code based on the HTML library, perhaps).

sbt has some utility libraries and convenience functions, in particular you can often use the convenient APIs in IO to manipulate files and directories.

Extending but not replacing a task

If you want to run an existing task while also taking another action, use `~=` or `<<=` to take the existing task as input (which will imply running that task), and then do whatever else you like after the previous implementation completes.

```
// These two settings are equivalent
intTask <<= intTask map { (value: Int) => value + 1 }
intTask ~= { (value: Int) => value + 1 }
```

Use plugins!

If you find you have a lot of custom code in `.scala` files, consider moving it to a plugin for re-use across multiple projects.

It's very easy to create a plugin, as teased earlier and discussed at more length here.

Getting Started Summary

Getting Started Guide page 14 of 14.

This page wraps up the Getting Started Guide.

To use sbt, there are a small number of concepts you must understand. These have some learning curve, but on the positive side, there isn't much to sbt *except* these concepts. sbt uses a small core of powerful concepts to do everything it does.

If you've read the whole Getting Started series, now you know what you need to know.

sbt: The Core Concepts

- the basics of Scala. It's undeniably helpful to be familiar with Scala syntax. Programming in Scala written by the creator of Scala is a great introduction.
- .sbt build definition
  - your build definition is one big list of `Setting` objects, where a `Setting` transforms the set of key-value pairs sbt uses to perform tasks.
  - to create a `Setting`, call one of a few methods on a key (the `:=` and `<<=` methods are particularly important).
  - there is no mutable state, only transformation; for example, a `Setting` transforms sbt's collection of key-value pairs into a new collection. It doesn't change anything in-place.
  - each setting has a value of a particular type, determined by the key.
  - *tasks* are special settings where the computation to produce the key's value will be re-run each time you kick off a task. Non-tasks compute the value once, when first loading the build definition.
- Scopes
  - each key may have multiple values, in distinct scopes.
  - scoping may use three axes: configuration, project, and task.
  - scoping allows you to have different behaviors per-project, per-task, or per-configuration.
  - a configuration is a kind of build, such as the main one (`Compile`) or the test one (`Test`).
  - the per-project axis also supports "entire build" scope.
  - scopes fall back to or *delegate* to more general scopes.

- .sbt vs. .scala build definition
    - put most of your settings in `build.sbt`, but use `.scala` build definition files to define multiple subprojects, and to factor out common values, objects, and methods.
    - the build definition is an sbt project in its own right, rooted in the `project` directory.
- Plugins are extensions to the build definition
    - add plugins with the `addSbtPlugin` method in `project/build.sbt` (NOT `build.sbt` in the project's base directory).

If any of this leaves you wondering rather than nodding, please ask for help on the mailing list, go back and re-read, or try some experiments in sbt's interactive mode.

Frequently Asked Questions

Project Information

How do I get help?

Please use the mailing list for questions, comments, and discussions.

- Please state the problem or question clearly and provide enough context. Code examples and build transcripts are often useful when appropriately edited.
- Providing small, reproducible examples are a good way to get help quickly.
- Include relevant information such as the version of sbt and Scala being used.

How do I report a bug?

Please use the issue tracker to report confirmed bugs. Do not use it to ask questions. If you are uncertain whether something is a bug, please ask on the mailing list first.

How can I help?

- Fix mistakes that you notice on the wiki.
- Make bug reports that are clear and reproducible.
- Answer questions on the mailing list.
- Fix issues that affect you. Fork, fix, and submit a pull request.
- Implement features that are important to you. There is an Opportunities page for some ideas, but the most useful contributions are usually ones you want yourself.

For more details on developing sbt, see Developing.pdf

0.7 to 0.10+ Migration

How do I migrate from 0.7 to 0.10+?

See the migration page first and then the following questions.

Where has 0.7's `lib_managed` gone?

By default, sbt 0.11 loads managed libraries from your ivy cache without copying them to a `lib_managed` directory. This fixes some bugs with the previous solution and keeps your project directory small. If you want to insulate your builds from the ivy cache being cleared, set `retrieveManaged := true` and the dependencies will be copied to `lib_managed` as a build-local cache (while avoiding the issues of `lib_managed` in 0.7.x).

This does mean that existing solutions for sharing libraries with your favoured IDE may not work. There are 0.11.x plugins for IDEs being developed:

- IntelliJ IDEA: https://github.com/mpeltonen/sbt-idea
- Netbeans: https://github.com/remeniuk/sbt-netbeans-plugin
- Eclipse: https://github.com/typesafehub/sbteclipse

What are the commands I can use in 0.11 vs. 0.7?

For a list of commands, run `help`. For details on a specific command, run `help <command>`. To view a list of tasks defined on the current project, run `tasks`. Alternatively, see the Running page in the Getting Started Guide for descriptions of common commands and tasks.

If in doubt start by just trying the old command as it may just work. The built in TAB completion will also assist you, so you can just press TAB at the beginning of a line and see what you get.

The following commands work pretty much as in 0.7 out of the box:

```
reload
update
compile
test
test-only
publish-local
exit
```

Why have the resolved dependencies in a multi-module project changed since 0.7?

sbt 0.10 fixes a flaw in how dependencies get resolved in multi-module projects. This change ensures that only one version of a library appears on a classpath.

Use `last update` to view the debugging output for the last `update` run. Use `show update` to view a summary of files comprising managed classpaths.

My tests all run really fast but some are broken that weren't in 0.7!

Be aware that compilation and tests run in parallel by default in sbt 0.11. If your test code isn't thread-safe then you may want to change this behaviour by adding one of the following to your `build.sbt`:

```
// Execute tests in the current project serially.
// Tests from other projects may still run concurrently.
parallelExecution in Test := false

// Execute everything serially (including compilation and tests)
parallelExecution := false
```

How do I set log levels in 0.11 vs. 0.7?

`warn`, `info`, `debug` and `error` don't work any more.

The new syntax in the sbt 0.11.x shell is:

```
> set logLevel := Level.Warn
```

Or in your `build.sbt` file write:

```
logLevel := Level.Warn
```

What happened to the web development and Web Start support since 0.7?

Web application support was split out into a plugin. See the xsbt-web-plugin project.

For an early version of an xsbt Web Start plugin, visit the xsbt-webstart project.

How are inter-project dependencies different in 0.11 vs. 0.7?

In 0.11, there are three types of project dependencies (classpath, execution, and configuration) and they are independently defined. These were combined in a single dependency type in 0.7.x. A declaration like:

```
lazy val a = project("a", "A")
lazy val b = project("b", "B", a)
```

meant that the `B` project had a classpath and execution dependency on `A` and `A` had a configuration dependency on `B`. Specifically, in 0.7.x:

1. Classpath: Classpaths for `A` were available on the appropriate classpath for `B`.
2. Execution: A task executed on `B` would be executed on `A` first.
3. Configuration: For some settings, if they were not overridden in `A`, they would default to the value provided in `B`.

In 0.11, declare the specific type of dependency you want. Read about multi-project builds in the Getting Started Guide for details.

Where did class/object X go since 0.7?

0.7	0.11
FileUtilities	IO
Path class and object	Path object, `File`, RichFile
PathFinder class	`Seq[File]`, PathFinder class, PathFinder object

Where can I find plugins for 0.11?

See sbt 0.10 plugins list for a list of currently available plugins.

Usage

My last command didn't work but I can't see an explanation. Why?

sbt 0.11 by default suppresses most stack traces and debugging information. It has the nice side effect of giving you less noise on screen, but as a newcomer it can leave you lost for explanation. To see the previous output of a command at a higher verbosity, type `last <task>` where `<task>` is the task that failed or that you want to view detailed output for. For example, if you find that your `update` fails to load all the dependencies as you expect you can enter:

```
> last update
```

and it will display the full output from the last run of the `update` command.

How do I disable ansi codes in the output?

Sometimes sbt doesn't detect that ansi codes aren't supported and you get output that looks like:

```
[0m[ [0minfo [0m]  [0mSet current project to root
```

or ansi codes are supported but you want to disable colored output. To completely disable ansi codes, set the `sbt.log.noformat` system property to `true`. For example,

```
sbt -Dsbt.log.noformat=true
```

Build definitions

What are the `:=`, `~=`, `<<=`, `+=`, `++=`, `<+=`, and `<++=` methods?

These are methods on keys used to construct a `Setting`. The Getting Started Guide covers all these methods, see .sbt build definition and more about settings for example.

What is the `%` method?

It's used to create a `ModuleID` from strings, when specifying managed dependencies. Read the Getting Started Guide about library dependencies.

What is `ModuleID`, `Project`, ...?

To figure out an unknown type or method, have a look at the Getting Started Guide if you have not. Also try the Index of commonly used methods, values, and types, the API Documentation, and the hyperlinked sources.

How can one key depend on multiple other keys?

See More About Settings in the Getting Started Guide, scroll down to the discussion of `<<=` with multiple keys.

Briefly: You need to use a tuple rather than a single key by itself. Scala's syntax for a tuple is with parentheses, like `(a, b, c)`.

If you're creating a value for a task key, then you'll use `map`:

```
packageBin in Compile <<= (name, organization, version) map { (n, o, v) => file(o +
"-" + n + "-" + v + ".jar") }
```

If you're creating a value for a setting key, then you'll use `apply`:

```
name <<= (name, organization, version) apply { (n, o, v) => "project " + n + " from "
+ o + " version " + v }
```

Typing `apply` is optional in that code, since Scala treats any object with an `apply` method as a function. See More About Settings for a longer explanation.

To learn about task keys vs. setting keys, read .sbt build definition.

How do I add files to a jar package?

The files included in an artifact are configured by default by a task `mappings` that is scoped by the relevant package task. The `mappings` task returns a sequence `Seq[(File,String)]` of mappings from the file to include to the path within the jar. See Mapping Files for details on creating these mappings.

For example, to add generated sources to the packaged source artifact:

```
mappings in (Compile, packageSrc) <++=
  (sourceManaged in Compile, managedSources in Compile) map { (base, srcs) =>
      import Path.{flat, relativeTo}
    srcs x (relativeTo(base) | flat)
  }
```

This takes sources from the `managedSources` task and relativizes them against the `managedSource` base directory, falling back to a flattened mapping. If a source generation task doesn't write the sources to the `managedSource` directory, the mapping function would have to be adjusted to try relativizing against additional directories or something more appropriate for the generator.

How can I generate source code or resources?

sbt provides standard hooks for adding source or resource generation tasks. A generation task should generate sources in a subdirectory of `sourceManaged` for sources or `resourceManaged` for resources and return a sequence of files generated. The key to add the task to is called `sourceGenerators` for sources and `resourceGenerators` for resources. It should be scoped according to whether the generated files are main (`Compile`) or test (`Test`) sources or resources. This basic structure looks like:

```
sourceGenerators in Compile <+= <your Task[Seq[File]] here>
```

For example, assuming a method `def makeSomeSources(base: File): Seq[File]`,

```
sourceGenerators in Compile <+= sourceManaged in Compile map { outDir: File =>
  makeSomeSources(outDir / "demo")
}
```

As a specific example, the following generates a hello world source file:

```
sourceGenerators in Compile <+= sourceManaged in Compile map { dir =>
  val file = dir / "demo" / "Test.scala"
  IO.write(file, """object Test extends App { println("Hi") }""")
  Seq(file)
}
```

Executing 'run' will print "Hi". Change `Compile` to `Test` to make it a test source. To generate resources, change `sourceGenerators` to `resourceGenerators` and `sourceManaged` to `resourceManaged`. Normally, you would only want to generate sources when necessary and not every run.

By default, generated sources and resources are not included in the packaged source artifact. To do so, add them as you would other mappings. See the `Adding files to a package` section.

How can a task avoid redoing work if the input files are unchanged?

There is basic support for only doing work when input files have changed or when the outputs haven't been generated yet. This support is primitive and subject to change.

The relevant methods are two overloaded methods called FileFunction.cached. Each requires a directory in which to store cached data. Sample usage is:

```
// define a task that takes some inputs
//   and generates files in an output directory
myTask <<= (cacheDirectory, inputs, target) map {
  (cache: File, inFiles: Seq[File], outDir: File) =>
    // wraps a function taskImpl in an uptodate check
    //   taskImpl takes the input files, the output directory,
    //   generates the output files and returns the set of generated files
    val cachedFun = FileFunction.cached(cache / "my-task") { (in: Set[File]) =>
      taskImpl(in, outDir) : Set[File]
    }
    // Applies the cached function to the inputs files
    cachedFun(inFiles)
}
```

There are two additional arguments for the first parameter list that allow the file tracking style to be explicitly specified. By default, the input tracking style is `FilesInfo.lastModified`, based on a file's last modified time, and the output tracking style is `FilesInfo.exists`, based only on whether the file exists. The other available style is `FilesInfo.hash`, which tracks a file based on a hash of its contents. See the FilesInfo API for details.

A more advanced version of `FileFunction.cached` passes a data structure of type ChangeReport describing the changes to input and output files since the last evaluation. This version of `cached` also expects the set of files generated as output to be the result of the evaluated function.

Extending sbt

How can I add a new configuration?

The following example demonstrates adding a new set of compilation settings and tasks to a new configuration called `samples`. The sources for this configuration go in `src/samples/scala/`. Unspecified settings delegate to those defined for the `compile` configuration. For example, if `scalacOptions` are not overridden for `samples`, the options for the main sources are used.

Options specific to `samples` may be declared like:

```
scalacOptions in Samples += "-deprecation"
```

This uses the main options as base options because of `+=`. Use `:=` to ignore the main options:

```
scalacOptions in Samples := "-deprecation" :: Nil
```

The example adds all of the usual compilation related settings and tasks to `samples`:

```
samples:run
samples:run-main
samples:compile
samples:console
samples:console-quick
samples:scalac-options
samples:full-classpath
samples:package
samples:package-src
...
```

Example of adding a new configuration

`project/Sample.scala`

```
import sbt._
import Keys._

object Sample extends Build {
     // defines a new configuration "samples" that will delegate to "compile"
   lazy val Samples = config("samples") extend(Compile)

     // defines the project to have the "samples" configuration
   lazy val p = Project("p", file("."))
      .configs(Samples)
      .settings(sampleSettings : _*)

   def sampleSettings =
        // adds the default compile/run/... tasks in "samples"
      inConfig(Samples)(Defaults.configSettings) ++
      Seq(
        // (optional) makes "test:compile" depend on "samples:compile"
         compile in Test <<= compile in Test dependsOn (compile in Samples)
      ) ++
        // (optional) declare that the samples binary and
        // source jars should be published
      publishArtifact(packageBin) ++
      publishArtifact(packageSrc)

   def publishArtifact(task: TaskKey[File]): Seq[Setting[_]] =
      addArtifact(artifact in (Samples, task), task in Samples).settings
}
```

How do I add a test configuration?

See the `Additional test configurations` section of Testing.

How can I create a custom run task, in addition to `run`?

This answer is extracted from a mailing list discussion.

Read the Getting Started Guide up to custom settings for background.

A basic run task is created by:

```
// this lazy val has to go in a full configuration
lazy val myRunTask = TaskKey[Unit]("my-run-task")

// this can go either in a `build.sbt` or the settings member
//   of a Project in a full configuration
fullRunTask(myRunTask, Test, "foo.Foo", "arg1", "arg2")
```

or, if you really want to define it inline (as in a basic `build.sbt` file):

```
 fullRunTask(TaskKey[Unit]("my-run-task"), Test, "foo.Foo", "arg1", "arg2")
```

If you want to be able to supply arguments on the command line, replace `TaskKey` with `InputKey` and `fullRunTask` with `fullRunInputTask`. The `Test` part can be replaced with another configuration, such as `Compile`, to use that configuration's classpath.

This run task can be configured individually by specifying the task key in the scope. For example:

```
fork in myRunTask := true

javaOptions in myRunTask += "-Xmx6144m"
```

How can I delegate settings from one task to another task?

Settings scoped to one task can fall back to another task if undefined in the first task. This is called delegation.

The following key definitions specify that settings for `myRun` delegate to `aRun`

```
val aRun = TaskKey[Unit]("a-run", "A run task.")

//   The last parameter to TaskKey.apply here is a repeated one
val myRun = TaskKey[Unit]("my-run", "Custom run task.", aRun)
```

In use, this looks like:

```
// Make the run task as before.
fullRunTask(myRun, Compile, "pkg.Main", "arg1", "arg2")

// If fork in myRun is not explicitly set,
//   then this also configures myRun to fork.
// If fork in myRun is set, it overrides this setting
//   because it is more specific.
fork in aRun := true
```

```
// Appends "-Xmx2G" to the current options for myRun.
//   Because we haven't defined them explicitly,
//   the current options are delegated to aRun.
//   So, this says to use the same options as aRun
//   plus -Xmx2G.
javaOptions in myRun += "-Xmx2G"
```

How should I express a dependency on an outside tool such as proguard?

Tool dependencies are used to implement a task and are not needed by project source code. These dependencies can be declared in their own configuration and classpaths. These are the steps:

1. Define a new configuration.
2. Declare the tool dependencies in that configuration.
3. Define a classpath that pulls the dependencies from the Update Report produced by `update`.
4. Use the classpath to implement the task.

As an example, consider a `proguard` task. This task needs the ProGuard jars in order to run the tool. Assuming a new configuration defined in the full build definition (#1):

```
val ProguardConfig = config("proguard") hide
```

the following are settings that implement #2-#4:

```
// Add proguard as a dependency in the custom configuration.
//  This keeps it separate from project dependencies.
libraryDependencies +=
   "net.sf.proguard" % "proguard" % "4.4" % ProguardConfig.name

// Extract the dependencies from the UpdateReport.
managedClasspath in proguard <<=
   (classpathTypes in proguard, update) map { (ct, report) =>
     Classpaths.managedJars(proguardConfig, ct, report)
   }

// Use the dependencies in a task, typically by putting them
//  in a ClassLoader and reflectively calling an appropriate
//  method.
proguard <<= managedClasspath in proguard { (cp: Seq[File] =>
  // ... do something with 'cp', which includes proguard ...
}
```

How would I change sbt's classpath dynamically?

It is possible to register additional jars that will be placed on sbt's classpath (since version 0.10.1). Through State, it is possible to obtain a xsbti.ComponentProvider, which manages application components. Components are groups of files in the `~/.sbt/boot/` directory and, in this case, the application is sbt. In addition to the base classpath, components in the "extra" component are included on sbt's classpath.

(Note: the additional components on an application's classpath are declared by the `components` property in the `[main]` section of the launcher configuration file `boot.properties`.)

Because these components are added to the `~/.sbt/boot/` directory and `~/.sbt/boot/` may be read-only, this can fail. In this case, the user has generally intentionally set sbt up this way, so error recovery is not typically necessary (just a short error message explaining the situation.)

Example of dynamic classpath augmentation

The following code can be used where a `State => State` is required, such as in the `onLoad` setting (described below) or in a command. It adds some files to the "extra" component and reloads sbt if they were not already added. Note that reloading will drop the user's session state.

```
def augment(extra: Seq[File])(s: State): State =
{
    // Get the component provider
  val cs: xsbti.ComponentProvider = s.configuration.provider.components()

    // Adds the files in 'extra' to the "extra" component
    //   under an exclusive machine-wide lock.
    //   The returned value is 'true' if files were actually copied and 'false'
    //   if the target files already exists (based on name only).
  val copied: Boolean = s.locked(cs.lockFile, cs.addToComponent("extra",
extra.toArray))

    // If files were copied, reload so that we use the new classpath.
  if(copied) s.reload else s
}
```

How can I take action when the project is loaded or unloaded?

The single, global setting `onLoad` is of type `State => State` (see Build State) and is executed once, after all projects are built and loaded. There is a similar hook `onUnload` for when a project is unloaded. Project unloading typically occurs as a result of a `reload` command or a `set` command. Because the `onLoad` and `onUnload` hooks are global, modifying this setting typically involves composing a new function with the previous value. The following example shows the basic structure of defining `onLoad`:

```
// Compose our new function 'f' with the existing transformation.
{
  val f: State => State = ...
  onLoad in Global ~= (f compose _)
}
```

Example of project load/unload hooks

The following example maintains a count of the number of times a project has been loaded and prints that number:

```
{
  // the key for the current count
  val key = AttributeKey[Int]("load-count")
  // the State transformer
  val f = (s: State) => {
    val previous = s get key getOrElse 0
    println("Project load count: " + previous)
    s.put(key, previous + 1)
  }
  onLoad in Global ~= (f compose _)
}
```

Errors

Type error, found: Initialize[Task[String]], required: Initialize[String] or found: TaskKey[String] required: Initialize[String]

This means that you are trying to supply a task when defining a setting key. See .sbt build definition for the difference between task and setting keys, and more about settings for more on how to define one key in terms of other keys.

Setting keys are only evaluated once, on project load, while tasks are evaluated repeatedly. Defining a setting in terms of a task does not make sense because tasks must be re-evaluated every time.

One way to get a task when you didn't want one is to use the `map` method instead of the `apply` method. More about settings covers this topic as well.

Suppose we define these keys, in `./project/Build.scala` (For details, see .scala build definition).

```
val baseSetting = SettingKey[String]("base-setting")
val derivedSetting = SettingKey[String]("derived-setting")
val baseTask = TaskKey[Long]("base-task")
val derivedTask = TaskKey[String]("derived-task")
```

Let's define an initialization for `base-setting` and `base-task`. We will then use these as inputs to other setting and task initializations.

```
baseSetting := "base setting"

baseTask := { System.currentTimeMillis() }
```

Then this will not work:

```
// error: found: Initialize[Task[String]], required: Initialize[String]
derivedSetting <<= baseSetting.map(_.toString),
derivedSetting <<= baseTask.map(_.toString),
derivedSetting <<= (baseSetting, baseTask).map((a, b) => a.toString + b.toString),
```

One or more settings can be used as inputs to initialize another setting, using the `apply` method.

```
derivedSetting <<= baseSetting.apply(_.toString)
derivedSetting <<= baseSetting(_.toString)
derivedSetting <<= (baseSetting, baseSetting)((a, b) => a.toString + b.toString)
```

Both settings and tasks can be used to initialize a task, using the `map` method.

```
derivedTask <<= baseSetting.map(_.toString)
derivedTask <<= baseTask.map(_.toString)
derivedTask <<= (baseSetting, baseTask).map((a, b) => a.toString + b.toString)
```

But, it is a compile time error to use `map` to initialize a setting:

```
// error: found: Initialize[Task[String]], required: Initialize[String]
derivedSetting <<= baseSetting.map(_.toString),
derivedSetting <<= baseTask.map(_.toString),
derivedSetting <<= (baseSetting, baseTask).map((a, b) => a.toString + b.toString),
```

It is not allowed to use a task as input to a settings initialization with `apply`:

```
// error: value apply is not a member of TaskKey[Long]
derivedSetting <<= baseTask.apply(_.toString)

// error: value apply is not a member of TaskKey[Long]
derivedTask <<= baseTask.apply(_.toString)

// error: value apply is not a member of (sbt.SettingKey[String], sbt.TaskKey[Long])
derivedTask <<= (baseSetting, baseTask).apply((a, b) => a.toString + b.toString)
```

Finally, it is not directly possible to use `apply` to initialize a task.

```
// error: found String, required Task[String]
derivedTask <<= baseSetting.apply(_.toString)
```

On project load, "Reference to uninitialized setting"

Setting initializers are executed in order. If the initialization of a setting depends on other settings that has not been initialized, sbt will stop loading. This can happen using `+=`, `++=`, `<<=`, `<+=`, `<++=`, and `~=`. (To understand those methods, read this.)

In this example, we try to append a library to `libraryDependencies` before it is initialized with an empty sequence.

```
object MyBuild extends Build {
  val root = Project(id = "root", base = file("."),
    settings = Seq(
      libraryDependencies += "commons-io" % "commons-io" % "1.4" % "test"
    )
  )
}
```

To correct this, include the default settings, which includes `libraryDependencies := Seq()`.

```
settings = Defaults.defaultSettings ++ Seq(
  libraryDependencies += "commons-io" % "commons-io" % "1.4" % "test"
)
```

A more subtle variation of this error occurs when using scoped settings.

```
// error: Reference to uninitialized setting
settings = Defaults.defaultSettings ++ Seq(
  libraryDependencies += "commons-io" % "commons-io" % "1.2" % "test",
  fullClasspath ~= (_.filterNot(_.data.name.contains("commons-io")))
)
```

Generally, all of the update operators can be expressed in terms of <<=. To better understand the error, we can rewrite the setting as:

```
// error: Reference to uninitialized setting
fullClasspath <<= (fullClasspath).map(_.filterNot(_.data.name.contains("commons-io")))
```

This setting varies between the test and compile scopes. The solution is use the scoped setting, both as the input to the initializer, and the setting that we update.

```
fullClasspath in Compile <<= (fullClasspath in Compile).map(_.filterNot
(_.data.name.contains("commons-io")))

// or equivalently
fullClasspath in Compile ~= (_.filterNot(_.data.name.contains("commons-io")))
```

Dependency Management

How do I resolve a checksum error?

This error occurs when the published checksum, such as a sha1 or md5 hash, differs from the checksum computed for a downloaded artifact, such as a jar or pom.xml. An example of such an error is:

```
[warn]  problem while downloading module descriptor:
http://repo1.maven.org/maven2/commons-fileupload/commons-fileupload/1.2.2/commons-
fileupload-1.2.2.pom:
invalid sha1: expected=ad3fda4adc95eb0d061341228cc94845ddb9a6fe
computed=0ce5d4a03b07c8b00ab60252e5cacdc708a4e6d8 (1070ms)
```

The invalid checksum should generally be reported to the repository owner (as was done for the above error). In the meantime, you can temporarily disable checking with the following setting:

```
checksums in update := Nil
```

See Library Management for details.

I've added a plugin, and now my cross-compilations fail!

This problem crops up frequently. Plugins are only published for the Scala version that SBT uses (currently, 2.9.1). You can still *use* plugins during cross-compilation, because SBT only looks for a 2.9.1 version of the plugin.

... unless you specify the plugin in the wrong place!

A typical mistake is to put global plugin definitions in `~/.sbt/plugins.sbt`. **THIS IS WRONG.** `.sbt` files in `~/.sbt` are loaded for *each* build--that is, for *each* cross-compilation. So, if you build for Scala 2.9.0, SBT will try to find a version of the plugin that's compiled for 2.9.0--and it usually won't. That's because it doesn't *know* the dependency is a plugin.

To tell SBT that the dependency is an SBT plugin, make sure you define your global plugins in a `.sbt` file in `~/.sbt/plugins/`. SBT knows that files in `~/.sbt/plugins` are only to be used by SBT itself, not as part of the general build definition. If you define your plugins in a file under *that* directory, they won't foul up your cross-compilations. Any file name ending in `.sbt` will do, but most people use `~/.sbt/plugins/build.sbt` or `~/.sbt/plugins/plugins.sbt`.

Miscellaneous

How do I use the Scala interpreter in my code?

sbt runs tests in the same JVM as sbt itself and Scala classes are not in the same class loader as the application classes. Therefore, when using the Scala interpreter, it is important to set it up properly to avoid an error message like:

```
Failed to initialize compiler: class scala.runtime.VolatileBooleanRef not found.
** Note that as of 2.8 scala does not assume use of the java classpath.
** For the old behavior pass -usejavacp to scala, or if using a Settings
** object programmatically, settings.usejavacp.value = true.
```

The key is to initialize the Settings for the interpreter using *embeddedDefaults*. For example:

```
val settings = new Settings
settings.embeddedDefaults[MyType]
val interpreter = new Interpreter(settings, ...)
```

Here, MyType is a representative class that should be included on the interpreter's classpath and in its application class loader. For more background, see the original proposal that resulted in *embeddedDefaults* being added.

Similarly, use a representative class as the type argument when using the *break* and *breakIf* methods of *ILoop*, as in the following example:

```
def x(a: Int, b: Int) = {
  import scala.tools.nsc.interpreter.ILoop
  ILoop.breakIf[MyType](a != b, "a" -> a, "b" -> b )
}
```

Note

This version of the online SBT documentation was manually converted into PDF format by Alvin Alexander of devdaily.com.