Covers Scala 3

SCALA for the Impatient

Third Edition

Cay S. Horstmann

Foreword by Martin Odersky

Scala for the Impatient

Third Edition

Now HashMap unambiguously refers to scala.collection.mutable.HashMap since java.util.HashMap is hidden.

7.10 Implicit Imports

Every Scala program implicitly starts with

```
import java.lang.*
import scala.*
import Predef.*
```

The java.lang package is always imported. Next, the scala package is imported, but in a special way. Unlike all other imports, this one is allowed to override the preceding import. For example, scala.StringBuilder overrides java.lang.StringBuilder instead of conflicting with it.

Finally, the Predef object is imported. It contains commonly used types, implicit conversions, and utility methods. (The methods could equally well have been placed into the scala package, but Predef was introduced long before Scala had package-level functions.)

Since the scala package is imported by default, you never need to write package names that start with scala. For example,

```
collection.mutable.HashMap
```

is just as good as

```
scala.collection.mutable.HashMap
```

7.11 Exports

This chapter ends with the export statement which, somewhat similar to import, provides aliases for certain features. However, import can be used with packages, classes, or objects, whereas export is only used with objects.

Here's a concrete example. A ColoredPoint has a color and a point. Now we'd like to get the color components and the point coordinates. Of course, we can delegate to the color methods and point fields:

```
import java.awt.*
class ColoredPoint(val color: Color, val point: Point) :
  def red = color.getRed()
  def green = color.getRed()
  def blue = color.getBlue()
  val x = point.x
  val y = point.y
```

That delegation can get tedious, and it sometimes encourages programmers to reach for inheritance. Then you automatically inherit everything and don't have to implement delegations.

However, inheritance may not be appropriate. Is a `ColoredPoint` a special kind of `Color`? A special kind of `Point`? There are philosophical and practical reasons why the answer might be "no."

A general software engineering principle is to favor composition over inheritance. Here, the `export` feature can help. Instead of manually delegating features, declare which ones you want:

```
class ColoredPoint(val color: Color, val point: Point) :
  export color.{getRed as red, getGreen as green, getBlue as blue}
  export point.{x, y}
```

The syntax is just like with `import` statements. Enclose the selected features in braces and use arrows for renaming.

As with imports, you can export all but a subset of features:

```
export point.{
  setLocation as _, translate as _, toString as _,
  hashCode as _, equals as _, clone as _, *}
```

Exercises

1. Write an example program to demonstrate that

   ```
   package com.horstmann.impatient
   ```

   is not the same as

   ```
   package com
   package horstmann
   package impatient
   ```

2. Write a puzzler that baffles your Scala friends, using a package `com` that isn't at the top level.
3. Write a package `random` with functions `nextInt(): Int`, `nextDouble(): Double`, and `setSeed(seed: Int): Unit`. To generate random numbers, use the linear congruential generator

   $next = (previous \times a + b) \bmod 2^n$,

   where $a = 1664525$, $b = 1013904223$, $n = 32$, and the initial value of *previous* is `seed`.
4. Make two source files, each contributing two classes and a top-level function to a given package. What class files are generated? In which directories?

What happens if you move the source files to different directories? What happens if you rename the source files?

5. What is the meaning of `private[com] def giveRaise(rate: Double)`? Is it useful?
6. Write a program that copies all elements from a Java hash map into a Scala hash map. Use imports to rename both classes.
7. In the preceding exercise, move all imports into the innermost scope possible.
8. What is the effect of

```
import java.*
import javax.*
```

   Is this a good idea?

9. Write a program that imports the `java.lang.System` class, reads the user name from the `user.name` system property, reads a password from the `StdIn` object, and prints a message to the standard error stream if the password is not `"secret"`. Otherwise, print a greeting to the standard output stream. Do not use any other imports, and do not use any qualified names (with dots).
10. Apart from `StringBuilder`, what other members of `java.lang` does the `scala` package override?
11. One common example of improper inheritance is a stack class that inherits from an `ArrayBuffer`. This is bad because the stack then inherits many methods that are not permitted on stacks. Use composition and `export` statements to define a `Stack` class that stores strings.
12. Pick an example where composition is preferred over inheritance and implement it in Scala, using the `export` syntax for method delegations.
13. An `export` statement can appear inside a class, object, or trait, or at the top level. If it appears in a class, how does it differ from an `import`? What if it appears at the top level?

This page intentionally left blank

Inheritance

Topics in This Chapter A1

- 8.1 Extending a Class — page 98
- 8.2 Overriding Methods — page 98
- 8.3 Type Checks and Casts — page 99
- 8.4 Superclass Construction — page 100
- 8.5 Anonymous Subclasses — page 101
- 8.6 Abstract Classes — page 101
- 8.7 Abstract Fields — page 102
- 8.8 Overriding Fields — page 102
- 8.9 Open and Sealed Classes — page 104
- 8.10 Protected Fields and Methods — page 105
- 8.11 Construction Order — page 105
- 8.12 The Scala Inheritance Hierarchy — page 106
- 8.13 Object Equality L1 — page 109
- 8.14 Multiversal Equality L2 — page 110
- 8.15 Value Classes L2 — page 111
- Exercises — page 112

Chapter 8

In this chapter, you will learn the most important ways in which inheritance in Scala differs from inheritance in other programming languages. (I assume that you are familiar with the general concept of inheritance.) The highlights are:

- The `extends` keyword denotes inheritance.
- You must use `override` when you override a method.
- A `final` class cannot be extended. A `final` method cannot be overridden.
- An `open` class is explicitly designed for being extended.
- Only the primary constructor can call the primary superclass constructor.
- You can override fields.
- You can define classes whose instances can only be compared with each other or other suitable types.
- A value class wraps a single value without the overhead of a separate object.

In this chapter, we only discuss the case in which a class inherits from another class. See Chapter 10 for inheriting *traits*—the Scala concept that generalizes Java interfaces.