Covers Scala 2.12

SCALA for the Impatient

Second Edition

Cay S. Horstmann

Foreword by Martin Odersky

Scala for the Impatient

Second Edition

This page intentionally left blank

Scala for the Impatient

Second Edition

Cay S. Horstmann

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382–3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016954825

ISBN-13: 978-0-13-454056-6
ISBN-10: 0-13-454056-5

Text printed in the United States of America.
1 16

To my wife, who made writing this book possible,
and to my children, who made it necessary.

This page intentionally left blank

Contents

Foreword to the First Edition

When I met Cay Horstmann some years ago he told me that Scala needed a better introductory book. My own book had come out a little bit earlier, so of course I had to ask him what he thought was wrong with it. He responded that it was great but too long; his students would not have the patience to read through the eight hundred pages of *Programming in Scala*. I conceded that he had a point. And he set out to correct the situation by writing *Scala for the Impatient*.

I am very happy that his book has finally arrived because it really delivers on what the title says. It gives an eminently practical introduction to Scala, explains what's particular about it, how it differs from Java, how to overcome some common hurdles to learning it, and how to write good Scala code.

Scala is a highly expressive and flexible language. It lets library writers use highly sophisticated abstractions, so that library users can express themselves simply and intuitively. Therefore, depending on what kind of code you look at, it might seem very simple or very complex.

A year ago, I tried to provide some clarification by defining a set of levels for Scala and its standard library. There were three levels each for application programmers and for library designers. The junior levels could be learned quickly and would be sufficient to program productively. Intermediate levels would make programs more concise and more functional and would make libraries more flexible to use. The highest levels were for experts solving specialized tasks. At the time I wrote:

> I hope this will help newcomers to the language decide in what order to pick subjects to learn, and that it will give some advice to teachers and book authors in what order to present the material.

Cay's book is the first to have systematically applied this idea. Every chapter is tagged with a level that tells you how easy or hard it is and whether it's oriented towards library writers or application programmers.

As you would expect, the first chapters give a fast-paced introduction to the basic Scala capabilities. But the book does not stop there. It also covers many of the more "senior" concepts and finally progresses to very advanced material which is not commonly covered in a language introduction, such as how to write parser combinators or make use of delimited continuations. The level tags serve as a guideline for what to pick up when. And Cay manages admirably to make even the most advanced concepts simple to understand.

I liked the concept of *Scala for the Impatient* so much that I asked Cay and his editor, Greg Doench, whether we could get the first part of the book as a free download on the Typesafe web site. They have gracefully agreed to my request, and I would like to thank them for that. That way, everybody can quickly access what I believe is currently the best compact introduction to Scala.

Martin Odersky

January 2012

Preface

The evolution of traditional languages such as Java, C#, and C++ has slowed down considerably, and programmers who are eager to use more modern language features are looking elsewhere. Scala is an attractive choice; in fact, I think it is by far the most attractive choice for programmers who want to improve their productivity. Scala has a concise syntax that is refreshing after the Java boilerplate. It runs on the Java virtual machine, providing access to a huge set of libraries and tools. And Scala doesn't just target the Java virtual machine. The ScalaJS project emits JavaScript code, enabling you to write both the server-side and client-side parts of a web application in a language that isn't JavaScript. Scala embraces the functional programming style without abandoning object orientation, giving you an incremental learning path to a new paradigm. The Scala interpreter lets you run quick experiments, which makes learning Scala very enjoyable. Last but not least, Scala is statically typed, enabling the compiler to find errors, so that you don't waste time finding them—or not—later in the running program.

I wrote this book for *impatient* readers who want to start programming in Scala right away. I assume you know Java, C#, or C++, and I don't bore you with explaining variables, loops, or classes. I don't exhaustively list all the features of the language, I don't lecture you about the superiority of one paradigm over another, and I don't make you suffer through long and contrived examples. Instead, you will get the information that you need in compact chunks that you can read and review as needed.

Scala is a big language, but you can use it effectively without knowing all of its details intimately. Martin Odersky, the creator of Scala, has identified levels of expertise for application programmers and library designers—as shown in the following table.

Application Programmer	Library Designer	Overall Scala Level
Beginning A1		Beginning
Intermediate A2	Junior L1	Intermediate
Expert A3	Senior L2	Advanced
	Expert L3	Expert

For each chapter (and occasionally for individual sections), I indicate the experience level required. The chapters progress through levels A1, L1, A2, L2, A3, L3. Even if you don't want to design your own libraries, knowing about the tools that Scala provides for library designers can make you a more effective library user.

This is the second edition of this book, and I updated it thoroughly for Scala 2.12. I added coverage of recent Scala features such as string interpolation, dynamic invocation, implicit classes, and futures, and updated all chapters to reflect current Scala usage.

I hope you enjoy learning Scala with this book. If you find errors or have suggestions for improvement, please visit `http://horstmann.com/scala` and leave a comment. On that page, you will also find a link to an archive file containing all code examples from the book.

I am very grateful to Dmitry Kirsanov and Alina Kirsanova who turned my manuscript from XHTML into a beautiful book, allowing me to concentrate on the content instead of fussing with the format. Every author should have it so good!

Reviewers include Adrian Cumiskey, Mike Davis, Rob Dickens, Steve Haines, Susan Potter, Daniel Sobral, Craig Tataryn, David Walend, and William Wheeler. Thanks so much for your comments and suggestions!

Finally, as always, my gratitude goes to my editor, Greg Doench, for encouraging me to write this book, and for his insights during the development process.

Cay Horstmann

San Francisco, 2016

About the Author

Cay S. Horstmann is author of *Core Java™, Volumes I & II, Tenth Edition* (Prentice Hall, 2016), as well as a dozen other books for professional programmers and computer science students. He is a professor of computer science at San Jose State University and a Java Champion.

Classes

Topics in This Chapter A1

Chapter 5

In this chapter, you will learn how to implement classes in Scala. If you know classes in Java or C++, you won't find this difficult, and you will enjoy the much more concise notation of Scala.

The key points of this chapter are:

- Fields in classes automatically come with getters and setters.
- You can replace a field with a custom getter/setter without changing the client of a class—that is the "uniform access principle."
- Use the `@BeanProperty` annotation to generate the JavaBeans `get`*Xxx*/`set`*Xxx* methods.
- Every class has a primary constructor that is "interwoven" with the class definition. Its parameters turn into the fields of the class. The primary constructor executes all statements in the body of the class.
- Auxiliary constructors are optional. They are called `this`.

5.1 Simple Classes and Parameterless Methods

In its simplest form, a Scala class looks very much like its equivalent in Java or C++:

```
class Counter {
  private var value = 0 // You must initialize the field
  def increment() { value += 1 } // Methods are public by default
  def current() = value
}
```

In Scala, a class is not declared as `public`. A Scala source file can contain multiple classes, and all of them have public visibility.

To use this class, you construct objects and invoke methods in the usual way:

```
val myCounter = new Counter // Or new Counter()
myCounter.increment()
println(myCounter.current)
```

You can call a parameterless method (such as `current`) with or without parentheses:

```
myCounter.current // OK
myCounter.current() // Also OK
```

Which form should you use? It is considered good style to use `()` for a *mutator* method (a method that changes the object state), and to drop the `()` for an *accessor* method (a method that does not change the object state).

That's what we did in our example:

```
myCounter.increment() // Use () with mutator
println(myCounter.current) // Don't use () with accessor
```

You can enforce this style by declaring `current` without `()`:

```
class Counter {
  ...
  def current = value // No () in definition
}
```

Now class users must use `myCounter.current`, without parentheses.

5.2 Properties with Getters and Setters

When writing a Java class, we don't like to use public fields:

```
public class Person { // This is Java
  public int age; // Frowned upon in Java
}
```

With a public field, anyone could write to `fred.age`, making Fred younger or older. That's why we prefer to use getter and setter methods:

```
public class Person { // This is Java
  private int age;
  public int getAge() { return age; }
  public void setAge(int age) { this.age = age; }
}
```

A getter/setter pair such as this one is often called a *property*. We say that the class `Person` has an `age` property.

Why is this any better? By itself, it isn't. Anyone can call `fred.setAge(21)`, keeping him forever twenty-one.

But if that becomes a problem, we can guard against it:

```
public void setAge(int newValue) { if (newValue > age) age = newValue; }
  // Can't get younger
```

Getters and setters are better than public fields because they let you start with simple get/set semantics and evolve them as needed.

NOTE: Just because getters and setters are better than public fields doesn't mean they are always good. Often, it is plainly bad if every client can get or set bits and pieces of an object's state. In this section, I show you how to implement properties in Scala. It is up to you to choose wisely when a gettable/settable property is an appropriate design.

Scala provides getter and setter methods for every field. Here, we define a public field:

```
class Person {
  var age = 0
}
```

Scala generates a class for the JVM with a *private* `age` field and getter and setter methods. These methods are public because we did not declare `age` as `private`. (For a private field, the getter and setter methods are private.)

In Scala, the getter and setter methods are called `age` and `age_=`. For example,

```
println(fred.age) // Calls the method fred.age()
fred.age = 21 // Calls fred.age_=(21)
```

In Scala, the getters and setters are not named `get`*Xxx* and `set`*Xxx*, but they fulfill the same purpose. Section 5.5, "Bean Properties," on page 61 shows how to generate Java-style `get`*Xxx* and `set`*Xxx* methods, so that your Scala classes can interoperate with Java tools.

NOTE: To see these methods with your own eyes, compile the `Person` class and then look at the bytecode with `javap`:

```
$ scalac Person.scala
$ javap -private Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements scala.ScalaObject{
  private int age;
  public int age(); public void age_$eq(int);
  public Person();
}
```

As you can see, the compiler created methods `age` and `age_$eq`. (The `=` symbol is translated to `$eq` because the JVM does not allow an `=` in a method name.)

TIP: You can run the `javap` command inside the REPL as

```
:javap -private Person
```

At any time, you can redefine the getter and setter methods yourself. For example,

```
class Person {
  private var privateAge = 0 // Make private and rename

  def age = privateAge
  def age_=(newValue: Int) {
    if (newValue > privateAge) privateAge = newValue; // Can't get younger
  }
}
```

The user of your class still accesses `fred.age`, but now Fred can't get younger:

```
val fred = new Person
fred.age = 30
fred.age = 21
println(fred.age) // 30
```

NOTE: Bertrand Meyer, the inventor of the influential Eiffel language, formulated the *Uniform Access Principle* that states: "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation." In Scala, the caller of `fred.age` doesn't know whether `age` is implemented through a field or a method. (Of course, in the JVM, the service is *always* implemented through a method, either synthesized or programmer-supplied.)

TIP: It may sound scary that Scala generates getter and setter methods for every field. But you have some control over this process.

- If the field is private, the getter and setter are private.
- If the field is a `val`, only a getter is generated.
- If you don't want any getter or setter, declare the field as `private[this]` (see Section 5.4, "Object-Private Fields," on page 60).

5.3 Properties with Only Getters

Sometimes you want a *read-only property* with a getter but no setter. If the value of the property never changes after the object has been constructed, use a `val` field:

```
class Message {
  val timeStamp = java.time.Instant.now
  ...
}
```

The Scala compiler produces a Java class with a private `final` field and a public getter method, but no setter.

Sometimes, however, you want a property that a client can't set at will, but that is mutated in some other way. The `Counter` class from Section 5.1, "Simple Classes and Parameterless Methods," on page 55 is a good example. Conceptually, the counter has a `current` property that is updated when the `increment` method is called, but there is no setter for the property.

You can't implement such a property with a `val`—a `val` never changes. Instead, provide a private field and a property getter, like this:

```
class Counter {
  private var value = 0
  def increment() { value += 1 }
  def current = value // No () in declaration
}
```

Note that there are no `()` in the definition of the getter method. Therefore, you *must* call the method without parentheses:

```
val n = myCounter.current // Calling myCounter.current() is a syntax error
```

To summarize, you have four choices for implementing properties:

1. `var foo`: Scala synthesizes a getter and a setter.
2. `val foo`: Scala synthesizes a getter.

3. You define methods foo and foo_=.
4. You define a method foo.

NOTE: In Scala, you cannot have a write-only property (that is, a property with a setter and no getter).

TIP: When you see a field in a Scala class, remember that it is not the same as a field in Java or C++. It is a private field *together with* a getter (for a val field) or a getter and a setter (for a var field).

5.4 Object-Private Fields

In Scala (as well as in Java or C++), a method can access the private fields of *all* objects of its class. For example,

```
class Counter {
  private var value = 0
  def increment() { value += 1 }

  def isLess(other : Counter) = value < other.value
    // Can access private field of other object
}
```

Accessing other.value is legal because other is also a Counter object.

Scala allows an even more severe access restriction with the private[this] qualifier:

```
private[this] var value = 0 // Accessing someObject.value is not allowed
```

Now, the methods of the Counter class can only access the value field of the current object, not of other objects of type Counter. This access is sometimes called *object-private*, and it is common in some OO languages such as SmallTalk.

With a class-private field, Scala generates private getter and setter methods. However, for an object-private field, no getters and setters are generated at all.

NOTE: Scala allows you to grant access rights to specific classes. The `private[`*ClassName*`]` qualifier states that only methods of the given class can access the given field. Here, the *ClassName* must be the name of the class being defined or an enclosing class. (See Section 5.8, "Nested Classes," on page 66 for a discussion of inner classes.)

In this case, the implementation will generate auxiliary getter and setter methods that allow the enclosing class to access the field. These methods will be public because the JVM does not have a fine-grained access control system, and they will have implementation-dependent names.

5.5 Bean Properties L1

As you saw in the preceding sections, Scala provides getter and setter methods for the fields that you define. However, the names of these methods are not what Java tools expect. The JavaBeans specification (`www.oracle.com/technetwork/articles/javaee/spec-136004.html`) defines a Java property as a pair of `getFoo`/`setFoo` methods (or just a `getFoo` method for a read-only property). Many Java tools rely on this naming convention.

When you annotate a Scala field with `@BeanProperty`, then such methods are automatically generated. For example,

```
import scala.beans.BeanProperty

class Person {
  @BeanProperty var name: String = _
}
```

generates *four* methods:

1. `name: String`
2. `name_=(newValue: String): Unit`
3. `getName(): String`
4. `setName(newValue: String): Unit`

Table 5–1 shows which methods are generated in all cases.

Table 5–1 Generated Methods for Fields

Scala Field	Generated Methods	When to Use
`val/var name`	public `name` `name_=` (`var` only)	To implement a property that is publicly accessible and backed by a field.
`@BeanProperty val/var name`	public `name` `getName()` `name_=` (`var` only) `setName(...)` (`var` only)	To interoperate with JavaBeans.
`private val/var name`	private `name` `name_=` (`var` only)	To confine the field to the methods of this class, just like in Java. Use `private` unless you really want a public property.
`private[this] val/var name`	none	To confine the field to methods invoked on the same object. Not commonly used.
`private[`*ClassName*`] val/var name`	implementation-dependent	To grant access to an enclosing class. Not commonly used.

NOTE: If you define a field as a primary constructor parameter (see Section 5.7, "The Primary Constructor," on page 63), and you want JavaBeans getters and setters, annotate the constructor parameter like this:

```
class Person(@BeanProperty var name: String)
```

5.6 Auxiliary Constructors

As in Java or C++, a Scala class can have as many constructors as you like. However, a Scala class has one constructor that is more important than all the others, called the *primary constructor*. In addition, a class may have any number of *auxiliary constructors*.

We discuss auxiliary constructors first because they are easier to understand. They are similar to constructors in Java or C++, with just two differences.

1. The auxiliary constructors are called `this`. (In Java or C++, constructors have the same name as the class—which is not so convenient if you rename the class.)
2. Each auxiliary constructor *must* start with a call to a previously defined auxiliary constructor or the primary constructor.

Here is a class with two auxiliary constructors:

```
class Person {
  private var name = ""
  private var age = 0

  def this(name: String) { // An auxiliary constructor
    this() // Calls primary constructor
    this.name = name
  }

  def this(name: String, age: Int) { // Another auxiliary constructor
    this(name) // Calls previous auxiliary constructor
    this.age = age
  }
}
```

We will look at the primary constructor in the next section. For now, it is sufficient to know that a class for which you don't define a primary constructor has a primary constructor with no arguments.

You can construct objects of this class in three ways:

```
val p1 = new Person // Primary constructor
val p2 = new Person("Fred") // First auxiliary constructor
val p3 = new Person("Fred", 42) // Second auxiliary constructor
```

5.7 The Primary Constructor

In Scala, every class has a primary constructor. The primary constructor is not defined with a `this` method. Instead, it is interwoven with the class definition.

1. The parameters of the primary constructor are placed *immediately after the class name*.

   ```
   class Person(val name: String, val age: Int) {
     // Parameters of primary constructor in (...)
     ...
   }
   ```

   Parameters of the primary constructor turn into fields that are initialized with the construction parameters. In our example, `name` and `age` become fields of the `Person` class. A constructor call such as `new Person("Fred", 42)` sets the `name` and `age` fields.

   Half a line of Scala is the equivalent of seven lines of Java:

```
public class Person { // This is Java
  private String name; private int age; public Person(String name, int age) {
    this.name = name; this.age = age;
  }
  public String name() { return this.name; } public int age() { return this.age; }
  ...
}
```

2. The primary constructor executes *all statements in the class definition*. For example, in the following class

```
class Person(val name: String, val age: Int) {
  println("Just constructed another person")
  def description = s"$name is $age years old"
}
```

the `println` statement is a part of the primary constructor. It is executed whenever an object is constructed.

This is useful when you need to configure a field during construction. For example:

```
class MyProg {
  private val props = new Properties
  props.load(new FileReader("myprog.properties"))
    // The statement above is a part of the primary constructor
  ...
}
```

NOTE: If there are no parameters after the class name, then the class has a primary constructor with no parameters. That constructor simply executes all statements in the body of the class.

TIP: You can often eliminate auxiliary constructors by using default arguments in the primary constructor. For example:

```
class Person(val name: String = "", val age: Int = 0)
```

Primary constructor parameters can have any of the forms in Table 5–1. For example,

```
class Person(val name: String, private var age: Int)
```

declares and initializes fields

```
val name: String
private var age: Int
```

Construction parameters can also be regular method parameters, without `val` or `var`. How these parameters are processed depends on their usage inside the class.

- If a parameter without `val` or `var` is used inside at least one method, it becomes a field. For example,

  ```
  class Person(name: String, age: Int) {
    def description = s"$name is $age years old"
  }
  ```

  declares and initializes immutable fields `name` and `age` that are object-private.

  Such a field is the equivalent of a `private[this] val` field (see Section 5.4, "Object-Private Fields," on page 60).
- Otherwise, the parameter is not saved as a field. It's just a regular parameter that can be accessed in the code of the primary constructor. (Strictly speaking, this is an implementation-specific optimization.)

Table 5–2 summarizes the fields and methods that are generated for different kinds of primary constructor parameters.

Table 5–2 Fields and Methods Generated for Primary Constructor Parameters

Primary Constructor Parameter	Generated Field/Methods
`name: String`	object-private field, or no field if no method uses `name`
`private val/var name: String`	private field, private getter/setter
`val/var name: String`	private field, public getter/setter
`@BeanProperty val/var name: String`	private field, public Scala and JavaBeans getters/setters

If you find the primary constructor notation confusing, you don't need to use it. Just provide one or more auxiliary constructors in the usual way, but remember to call `this()` if you don't chain to another auxiliary constructor.

However, many programmers like the concise syntax. Martin Odersky suggests to think about it this way: In Scala, classes take parameters, just like methods do.

NOTE: When you think of the primary constructor's parameters as class parameters, parameters without `val` or `var` become easier to understand. The scope of such a parameter is the entire class. Therefore, you can use the parameter in methods. If you do, it is the compiler's job to save it in a field.

TIP: The Scala designers think that *every keystroke is precious*, so they let you combine a class with its primary constructor. When reading a Scala class, you need to disentangle the two. For example, when you see

```
class Person(val name: String) {
  var age = 0
  def description = s"$name is $age years old"
}
```

take this definition apart into a class definition:

```
class Person(val name: String) {
  var age = 0
  def description = s"$name is $age years old"
}
```

and a constructor definition:

```
class Person(val name: String) {
  var age = 0
  def description = s"$name is $age years old"
}
```

NOTE: To make the primary constructor private, place the keyword `private` like this:

```
class Person private(val id: Int) { ... }
```

A class user must then use an auxiliary constructor to construct a `Person` object.

5.8 Nested Classes L1

In Scala, you can nest just about anything inside anything. You can define functions inside other functions, and classes inside other classes. Here is a simple example of the latter:

```
import scala.collection.mutable.ArrayBuffer
class Network {
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Member]
  }

  private val members = new ArrayBuffer[Member]

  def join(name: String) = {
    val m = new Member(name)
    members += m
```

```
      m
    }
  }
```

Consider two networks:

```
val chatter = new Network
val myFace = new Network
```

In Scala, each *instance* has its own class `Member`, just like each instance has its own field `members`. That is, `chatter.Member` and `myFace.Member` are *different classes*.

NOTE: This is different from Java, where an inner class belongs to the outer class.

The Scala approach is more regular. For example, to make a new inner object, you simply use `new` with the type name: `new chatter.Member`. In Java, you need to use a special syntax, `chatter.new Member()`.

In our network example, you can add a member within its own network, but not across networks.

```
val fred = chatter.join("Fred")
val wilma = chatter.join("Wilma")
fred.contacts += wilma // OK
val barney = myFace.join("Barney") // Has type myFace.Member
fred.contacts += barney
  // No—can't add a myFace.Member to a buffer of chatter.Member elements
```

For networks of people, this behavior probably makes sense. If you don't want it, there are two solutions.

First, you can move the `Member` type somewhere else. A good place would be the `Network` companion object. (Companion objects are described in Chapter 6.)

```
object Network {
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Member]
  }
}

class Network {
  private val members = new ArrayBuffer[Network.Member]
  ...
}
```

Alternatively, you can use a *type projection* `Network#Member`, which means "a `Member` of *any* `Network`." For example,

```
class Network {
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Network#Member]
  }
  ...
}
```

You would do that if you want the fine-grained "inner class per object" feature in some places of your program, but not everywhere. See Chapter 19 for more information about type projections.

NOTE: In a nested class, you can access the `this` reference of the enclosing class as *EnclosingClass*`.this`, like in Java. If you like, you can establish an alias for that reference with the following syntax:

```
class Network(val name: String) { outer =>
  class Member(val name: String) {
    ...
    def description = s"$name inside ${outer.name}"
  }
}
```

The `class Network { outer =>` syntax makes the variable `outer` refer to `Network.this`. You can choose any name for this variable. The name `self` is common, but perhaps confusing when used with nested classes.

This syntax is related to the "self type" syntax that you will see in Chapter 19.

Exercises

1. Improve the `Counter` class in Section 5.1, "Simple Classes and Parameterless Methods," on page 55 so that it doesn't turn negative at `Int.MaxValue`.
2. Write a class `BankAccount` with methods `deposit` and `withdraw`, and a read-only property `balance`.
3. Write a class `Time` with read-only properties `hours` and `minutes` and a method `before(other: Time): Boolean` that checks whether this time comes before the other. A `Time` object should be constructed as `new Time(hrs, min)`, where `hrs` is in military time format (between 0 and 23).
4. Reimplement the `Time` class from the preceding exercise so that the internal representation is the number of minutes since midnight (between 0 and $24 \times 60 - 1$). *Do not* change the public interface. That is, client code should be unaffected by your change.

5. Make a class `Student` with read-write JavaBeans properties `name` (of type `String`) and `id` (of type `Long`). What methods are generated? (Use `javap` to check.) Can you call the JavaBeans getters and setters in Scala? Should you?
6. In the `Person` class of Section 5.1, "Simple Classes and Parameterless Methods," on page 55, provide a primary constructor that turns negative ages to 0.
7. Write a class `Person` with a primary constructor that accepts a string containing a first name, a space, and a last name, such as `new Person("Fred Smith")`. Supply read-only properties `firstName` and `lastName`. Should the primary constructor parameter be a `var`, a `val`, or a plain parameter? Why?
8. Make a class `Car` with read-only properties for manufacturer, model name, and model year, and a read-write property for the license plate. Supply four constructors. All require the manufacturer and model name. Optionally, model year and license plate can also be specified in the constructor. If not, the model year is set to `-1` and the license plate to the empty string. Which constructor are you choosing as the primary constructor? Why?
9. Reimplement the class of the preceding exercise in Java, C#, or C++ (your choice). How much shorter is the Scala class?
10. Consider the class

```
class Employee(val name: String, var salary: Double) {
  def this() { this("John Q. Public", 0.0) }
}
```

Rewrite it to use explicit fields and a default primary constructor. Which form do you prefer? Why?

Index

Symbols and Numbers

B

D

E

F

G

H

I

J

N

O

P

S

T

U

V