**THE SCALA BOOK**

**The Java Virtual Machine**

The Java virtual machine is a program whose purpose is to execute other programs.

*Technical definition:* The JVM is the specification for a software program that executes code and provides the runtime environment for that code.

*Everyday definition:* The JVM is how we run our Java programs. We configure the JVM's settings and then rely on it to manage program resources during execution.

The JVM has two primary functions: to allow Java programs to run on any device or operating system (known as the "Write once, run anywhere" principle), and to manage and optimize program memory. When Java was released in 1995, all computer programs were written to a specific operating system, and program memory was managed by the software developer. So the JVM was a revelation.

**Memory management – Garbage collector**

The most common interaction with a running JVM is to check the memory usage in the [heap and stack](https://www.javaworld.com/article/2077184/core-java/the-lean--mean--virtual-machine.html). The most common adjustment is [tuning the JVM's memory settings](https://www.javaworld.com/article/2078645/java-se/jvm-performance-optimization-part-3-garbage-collection.html).

Before Java, all program memory was managed by the programmer. In Java, program memory is managed by the JVM. The JVM manages memory through a process called ***garbage collection***, which continuously identifies and eliminates unused memory in Java programs. Garbage collection happens inside a running JVM.

In the early days, Java came under a lot of criticism for not being as "close to the metal" as C++, and therefore not as fast. The garbage collection process was especially controversial. Since then, a variety of algorithms and approaches have been proposed and used for garbage collection. With consistent development and optimization, garbage collection has vastly improved.

**The Basics**

A value declared with val is actually a constant – you can´t change its contents

To declare variables whose constants can vary, use a var.

var counter = 0

counter = 1

You can specify the type if necessary

Val greeting: String = null

Val greeting: Any = “Hello”

You can declare multiple values or variables together

Val xmax, ymax = 100

**Common used types**

Scala has seven numeric types (Byte, Char, Short, Int, Long, Float and Double) and a Boolean type. However, unlike java, these types are clases. There is no distinction between ***primitive types*** and class types in Scala. Cou can invoke methods on numbers:

1.toString()

1.to(10) // Yields Range(1,2,3,4,5,6,7,8,9,10)

In scala there is no need for ***wrapper types***. It is the job of the Scala compiler to convert between types and wrappers. For example if you make an array of Int, you get an int[] array in the virtual machine. Scala relies on the underlying java.lang.String class for strings. However it augments that class with well over a hundred operations in the StringOps class. For example, the intersect method yields the characters that are common between two strings.

“Hello”.intersect(“World”) // Yields “lo”

Similarity, there are clases RichInt, RichDouble, RichChar, and so on. Each of them has a small set of convinience methods for acting on their por cousins – Int, Double, or Char. Finally, there are clases BigInt and BigDecimal for computations with an arbitrary, but finite number of digits, which are more convinient because they can be used with the usual mathematical operators.mk´´

In scala, methods instead of casts are used to convert between numeric types.   
.toInt, .toDouble, .toChar, .toString.

**Arithmetic and operator overloading**

The + - \* / % operators do their usual job, as do the bit operatiors & | ^ >> <<. There is just one surprising aspect. Thes operators are actually methods. For example a + b is shorthand for a.+(b). Here, + is the name of the method. Scala has no silly prejudice agains non-alphanumeric characters in method names. In general you can write a method b as a shorthand for a.method(b)such as 1.to(10) to 1 to 10

**More about calling methods**

If the method has no parameters, you don’t have to use parentheses. The rule of thumb is that a parameterless method that does not modify the object has no parentheses.

Often, a class has a companion object whose methods act just like static methods do in Java. For example, the BigInt companion object to the scala.math.BigInt class has a method probablePrime tat generates a random prime number with given number of bits. Static methods in java cannot use any instance variable

In scala, it is common to use a syntax that looks like a function call. For example, if s is a string, then s(i) is the ith carácter of the string. (In java or C you would write s[i]).

You can think of this as an overloaded form of the () operator. It is implemented as a method with the name apply. For example, in the documentation of the StringOps class, you will find a method def apply(n: Int): Char . That is, s(4) is a shortcut for s.apply(4). The BigInt companionobject´s apply method, let you convert strings or numbers to BigInt objects. For examplle the call BigInt(“1234”) is a shortcut for BigInt.apply(“1234”). It yields a new BigInt object without having to use new, or also for example Array(1,2,3,4) returns an array, thanks to the apply method of the Array companion object.

Methods can have functions as parameters. For example, the count method in StringOps requires a function that returns true or false for a Char, specifying which characters should be counted:

def count(p: **(Char) => Boolean**) : Int

You supply a function, often in a very compact notation, when you call the method. As an example, the call s.count(\_.isUpper) counts the number of uppercase characters.

This programming style is called ***Higer-Order Functions***

Finally, some methods have an ***implicit*** parameter. F.E, the sorted method of StringOps: def sorted[B >: Char](implicit ord: math.Ordering[B]): String

Further details could be seen in Chapter 21 “Implicits”. And don´t worry if you run into into indecipherable incantation, such as the [B >: Char] in the declaration of sorted. The expression B >: Char means “ny supertype of Char” "Scala".sorted // "Saacl"

***Control Structures and Functions***

**Conditional Expressions**

In Scala, an if/else has a value that we could put in a variable:

val s = if (x > 0) 1 else -1 //== if (x > 0) s = 1 else s = -1

However the first form is better because it can be used to initialize a val, and the second form needs to be a var. As already mentioned, semicolons are mostly optional.

In java and C++ the operator ?: has the same purpose x > 0 ? 1 : -1.

In Scala, ***every expression has a type***. For example the last example has the type Int because both branches have the type Int. The type of a mixed-type expression, such as if (x > 0) "positive" else -1 is the common ***supertype*** of both branches, which is Any. In case the else part is omitted, for example if(x>0) 1, then it is posible that the if statement yields no value. However, in Scala, every expression is suposed to have some value. This is finessed by introducing a class Unit that has one value, written as (). The if statement without an else is equivalent to if (x > 0) 1 else ()

Think of () as a placeholder for “no useful value,” and of Unit as an analog of void.

Thechnically speaking, void has no value whereas Unit has one value that signifies “no value”. Scala has no switch statement, but it has a much more powerful pattern matching mechanism that we will discuss later.

**Statement Termination**

In Scala a semicolon is never required if it falls just before the end of the line. However, if you want to have more than one statement on a single line, you need to separate them with semicolons. if (n > 0) { r = r \* n; n -= 1 }

**Black expressions and assignments**

In Java or C++ a ***block statement*** is a sequence of statements enclosed in { }. You use a block statement whenever you need to put multiple actions in the body of a branch or loop statement. In Scala, a { } block contains a sequence of expressions, and the result is also an expression. The value of the block is the value of the expression, this feature can be useful if the initialization of a val takes more tan one step. For example:

val distance = { val dx = x - x0; val dy = y - y0; sqrt(dx \* dx + dy \* dy) }

In Scala, assignments have no value, or, strictly speaking, they have a value of type Unit, with a single value written as ().

A block that end with an assignment, such as { r = r \* n; n -= 1 } has a Unit value. This is not a problema, jist something to be aware of when defining functions. 2.7 Functions

Since assignments have Unit value, don’t chain them together. x = y = 1 // No

The value of y = 1 is (), and it’s highly unlikely that you wanted to assign a Unit to x.

(In contrast, in Java and C++, the value of an assignment is the value that is being assigned. In those languages, chained assignments are useful.)

**Input and output**

To print a value, use the print or println function. The latter adds a newline character after the printout. There is also a printf function with a C-style format string, or better, use string interpolation print(f”Hello, $name”)

**Formatted strings** are one of three predefined string interpolators in the Scala library. With a prefix of s, strings can contain expressions but not format directives. With a prefix of raw, escape sequences in a string are not evaluated. For example, raw"\n is a newline" starts with a backslash and the letter n, not a newline character. To include a $ sign in an interpolated string, double it. For example, s"$$$price" yields a dollar sign followed by the value of price.

You can read a line of input from the console with the readLine method of the scala.io.StdIn class. Use readInt, readDouble, readByte, readShort, readLong, readFloat, readBoolean, readChar, or readLine for strings.

**Loops**

Scala has the same while and do loops as Java and C++. For example,

while (n > 0) {  r = r \* n;  n -= 1}

Scala has no direct analog of the for (initialize; test; update) loop. If you need such a loop, you have two choices. You can use a while loop. Or, you can use a for statement like this: for (i <- 1 to n)  r = r \* i. You saw the to method of the RichInt class in Chapter 1. The call 1 to n returns a Range of the numbers from 1 to n (inclusive). The construct for (i <- expr) makes the variable i traverse all values of the expression to the right of the <-. When traversing a string, you can loop over the index values:

val s = "Hello"; var sum = 0;for (i <- 0 to s.length - 1){sum += s(i)}

In contrast in this example, there is actually no need to use indexes. You can directly loop over the characters: var sum = 0; for (ch <- "Hello") sum += ch

In Scala, loops are not used as often as in other languages. As you will see in Chapter 12, you can often process the values in a sequence by applying a function to all of them, which can be done with a single method call.

Scala has **no break** or continue statements to break out of a loop. What to do if you need a break? Here are a few options:

- Use a Boolean control variable.

- Use nested functions, you can return from the middle of a function.

- Use the break method in the Breaks object

import scala.util.control.Breaks.\_

breakable { for(...) { if(...) break;//Exits the breakable bloc }}

Here, the control transfer is done by throwing and catching an exception, so you should avoid this mechanism when time is of essence.ç+

**Advanced Loops**

You can have multiple generators of the form variable <- expression. Separate them by semicolons. For example,

for (i <- 1 to 3; j <- 1 to 3) print(f"${10 \* i + j}%3d")

/ / Prints 11 12 13 21 22 23 31 32 33

Each generator can have a **guard**, a Boolean condition preceded by if:

for (i <- 1 to 3; j <- 1 to 3 if i != j) print(f"${10 \* i + j}%3d")

// Prints 12 13 21 23 31 32

Note that there is no semicolon before the if. You can have any number of definitions, introducing variables that can be used inside the loop:

for (i <- 1 to 3; from = 4 - i; j <- from to 3) print(f"${10 \* i + j}%3d")  
// Prints 13 22 23 31 32 33

When the body of the for loop starts with yield, the loop constructs a collection of values, one for each iteration, this type of loop is called a **for comprehension**:

for (i <- 1 to 10) yield i % 3

// Yields Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)

The generated collection is compatible with the first generator.

for (c <- "Hello"; i <- 0 to 1) yield (c + i).toChar

// Yields "HIeflmlmop

"for (i <- 0 to 1; c <- "Hello") yield (c + i).toChar

// Yields Vector('H', 'e', 'l', 'l', 'o', 'I', 'f', 'm', 'm', 'p')

If you prefer, you can enclose the generators, guards, and definitions of a for loop in braces, and you can use newlines instead of semicolons to separate them:

for { i <- 1 to 3

from = 4 - i

  j <- from to 3 }

**Functions**

Scala has functions in addition to methods. A method operates on an object, but a function doesn’t. C++ has functions as well, but in Java, you have to imitate them with static methods. def abs(x: Double) = if (x >= 0) x else -x

You must specify the types of all parameters. However, as long as the function is not recursive, you need not specify the return type. The Scala compiler determines the return type from the type of the expression to the right of the = symbol. If the body of the function requires more than one expression, use a block. The last expression of the block becomes the value that the function returns. For example, the following function returns the value of r after the for loop.

def fac(n : Int) = {

var r = 1

for (i <- 1 to n) r = r \* i

r

}

While there is nothing wrong with using return *keystroke* in a named function (except the waste of seven keystrokes), it is a good idea to get used to life without return. Pretty soon, you will be using lots of ***anonymous functions***, and there, return doesn’t return a value to the caller but breaks out to the enclosing named function. Think of return as a kind of break statement for functions, and only use it when you want that breakout functionality. With a recursive function, you must specify the return type. For example,

def fac(n: Int): Int = if (n <= 0) 1 else n \* fac(n - 1)

Without the return type, the Scala compiler couldn’t verify that the type of n \* fac(n - 1) is an Int.

Note that there is a difference between **return keystroke** and **return type**.

**Private functions/members**

A private function is visible only inside the class or object that contains its definition.

**Protected functions/members**

A protected member is only accesible from subclases of the class in which the member is defined.

**Public functions/members**

For public members is not required to specify a keyword.

**Default and Named Arguments**

You can provide default arguments for functions that are used when you don’t specify explicit values. For example,

def decorate(str: String, left: String = "[", right: String = "]")

You can also specify the parameter names when you supply the arguments. decorate(left = "<<<", str = "Hello", right = ">>>")

Named arguments can make a function call more readable. They are also useful if a function has many default parameters.

**Variable Arguments**

Sometimes, it is convenient to implement a function that can take a variable number of arguments. The following example shows the syntax:

def sum(args: Int\*) = {…}

val s = sum(1, 4, 9, 16, 25)

The function receives a single parameter of type Seq, which we will discuss in Chapter 13. For now, all you need to know is that you can use a for loop to visit each element.

val s = sum(1 to 5) // Error

If the sum function is called with one argument, that must be a single integer, not a range of integers. The remedy is to tell the compiler that you want the parameter to be considered an ***argument sequence***. Append : **\_\***, like this:

This call syntax is needed in a recursive definition:

val s = sum(1 to 5: \_\*) // Consider 1 to 5 as an argument sequence

Here, the head of a sequence is its initial element, and tail is a sequence of all other elements. That’s again a Seq, and we have to use : **\_\*** to convert it to an argument sequence:

def recursiveSum(args: Int\*) : Int = {

if (args.length == 0) 0

else args.head + recursiveSum(args.tail : \_\*)

}

**Procedures**

Scala has a special notation for a function that returns no value. If the function body is enclosed in braces without a preceding = symbol, then the return type is Unit. Such a function is called a procedure.

def box(s : String)//no ‘=’{ print(f"$border%n|$s|%n$border%n")}

def box(s : String)**:** **Unit** = {  ...}

**Lazy values**

When a val is declared as ***lazy***, its initialization is deferred until it is accessed for the first time. lazy val words = scala.io.Source.fromFile("fileName").mkString

If the program never accesses words, the file is never opened.

Lazy values are useful to delay costly initialization statements. They can also deal with other initialization issues, such as **circular dependencies**. Moreover, they are essential for developing lazy data structures—see Section 13.12, “Streams,”.

You can think of lazy values as halfway between val and def. Compare

val words = scala.io.Source.fromFile("/words").mkString

//Evaluated as soon as words is definedlazy

val words = scala.io.Source.fromFile("/words").mkString

//Evaluated the first time words is used

def words = scala.io.Source.fromFile(" /words").mkString

//Evaluated every time words is used

Laziness is not cost-free. Every time a lazy value is accessed, a method is called that checks, in a threadsafe manner, whether the value has already been initialized.

**Exceptions**

Scala exceptions work the same way as in Java or C++. For example throw an exception, throw new IllegalArgumentException("x should not be negative")

The current computation is aborted, and the runtime system looks for an exception handler that can accept an IllegalArgumentException. Control resumes with the innermost such handler. If no *such handler* exists, the *program terminates*.

As in Java, the objects that you throw need to belong to a subclass of java.lang.Throwable. However, unlike Java, Scala has no “checked” exceptions—you never have to declare that a function or method might throw an exception.

In Java, **“checked” exceptions** are checked at compile time. If your method might throw an IOException, you must declare it. This forces programmers to think where those exceptions should be handled, which is a laudable goal. Many Java programmers detest this feature and end up defeating it by either catching exceptions too early or using excessively general exception classes. The Scala designers decided against checked exceptions, recognizing that thorough **compile-time checking** isn’t always a good thing.

A throw expression has the special type Nothing. That is useful in if/else expressions. If one branch has type Nothing, the type of the if/else expression is the type of the other branch. For example, consider

if (x >= 0) { sqrt(x)} else throw new IllegalArgumentException("x should not be negative")

The first branch has type Double, the second has type Nothing. Therefore, the if/else expression also has type Double.

The syntax for catching exceptions is modeled after the **pattern matching** syntax (see Chapter 14).

val url = new URL("http://horstmann.com/fred-tiny.gif")

try {

process(url)

} catch {

case \_: MalformedURLException => println(s"Bad URL: $url")

case ex: IOException => ex.printStackTrace()

}

The more general exception types should come after the more specific ones.

Note that you can use \_ for the variable name if you don’t need it.

The try/finally statement lets you dispose of a resource whether or not an exception has occurred. For example:

val in = new URL("http://horstmann.com/fred.gif").openStream()

try { process(in)}

finally { in.close()}

Note that try/catch and try/finally have complementary goals. The try/catch statement handles exceptions, and the try/finally statement takes some action (usually cleanup) when an exception is not handled. You can combine them into a single try/catch/finally statement:

try { try { ... } catch { ... } } finally { ... }

The Try class is designed to work with computations that may fail with exceptions. We will look at it more closely in Chapter 17. Here is a simple example:

import scala.io.\_

val result =

for ( a <- Try { StdIn.readLine("a: ").toInt };

b <- Try { StdIn.readLine("b: ").toInt })

yield a / b

If an exception occurs in either of the calls to toInt, or because of division by zero, then result is a Failure object, containing the exception that caused the computation to fail. Otherwise, result is a Success object holding the result of the computation.

***Arrays***

Use an Array if the lenght is fixed, and an ArrayBuffer if the lenght can vary.

Don’t use new when supplying initial values

Use () to acces eements

Use for (elem <- arr) to traverse the elements

Use for (elem <- arr if…) …yield… to transform into a new array

Scala an Java arrays are interoperable; with ArrayBuffer, use scala.collection.JavaConversions

**Fixed-Lenght Arrays**

If you need an array whose lenght doesn’t chane, use Array type in Scala. For example

val nums = new Array[Int](10) // An array of ten integers of 0

val a = new Array[String](10) ) // An array of ten null elements

val s = Array(“Hello, “World”) ) // An array of lenght 2

s(0) = “Goodbye” // Array(“Goodbye”, “World”), use () instead of []

Inside the JVM, a Scala Array is implemented as a Java array. An strings array in the would have the type java.lang.String[] inside the JVM, and an array of Int, Double, or another equivalent of the Java primitive types is a primitive type array. For example, Array(2,3,5,7,11) is an int[] in the JVM.

**Variable-Lenght Arrays: Array Buffers**

The variable array in Scala the ArrayBuffer.

import scala.collection.mutable.ArrayBuffer

val b = ArrayBuffer[Int]()  // new ArrayBuffer[Int] // empty array

b += 1  // ArrayBuffer(1) // Add an element at the end with +=

b += (1, 2)// ArrayBuffer(1, **1, 2**)// Add multiple elements

b ++= Array(8, 13)//ArrayBuffer(1, 1, 2, **8, 13**)//Append any Collection

b.trimEnd(2)// ArrayBuffer(1, 1, 2)// Removes the last two elements

Adding or removing elements at the end of an array buffer is an efficient (“amortized constant time”) operation.You can also insert and remove elements at an arbitrary location, but those operations are not as efficient—all elements after that location must be shifted. For example:

b.insert(2, 6)  // ArrayBuffer(1, 1, 6, 2)// Insert before index 2

b.insert(2, 7, 8, 9)  // ArrayBuffer(1, 1, **7, 8, 9**, 6, 2)

b.remove(2) // ArrayBuffer(1, 1, 8, 9, 6, 2)

b.remove(2, 3)// ArrayBuffer(1, 1, 2)

Sometimes, you want to build up an Array, but you don’t yet know how many elements you will need. First create an ArrayBuffer and then: b.toArray // Array(1, 1, 2)

Conversely, call a.toBuffer to convert the array a to an array buffer.

**Traversing arrays and array buffers**

In most of the cases we can use the same code for lists, arrays and vectors.

Here is how you traverse an array or array buffer with a for loop:

for (i <- 0 until a.length)  println(s"$i: ${a(i)}")

The until method is similar to the to method, except that it excludes the last value. Therefore, the variable i goes from 0 to a.length - 1.

In general, the construct for (i <- range) makes the variable i traverse all values of the range.

To visit every second element, let i traverse :

0 until a.length by 2 //Range(0, 2, 4, ...)

To visit the elements starting from the end of the array, traverse:

0 until a.length by -1  // Range(..., 2, 1, 0)

Instead of 0 until a.length or 0 until a.length by -1, you can use a.indices or a.indices.reverse.

If you don’t need the array index in the loop body, visit the array elements directly:

for (elem <- a)  println(elem)

**Transforming arrays**

In Scala is easy to take an array (or array buffer) and transform it in some way. Such transformations don’t modify the original array but yield a new one.

val a = Array(2, 3, 5, 7, 11)

val result = for (elem <- a) yield 2 \* elem // Array(4, 6, 10, 14, 22)

The for/yield loop creates a new collection of the same type as the original collection. Oftentimes, when you traverse a collection, you only want to process the elements that match a particular condition. This is achieved with a guard: an if inside the for. Here we double every even element, dropping the odd ones:

for (elem <- a **if elem % 2 == 0**) yield 2 \* elem

Keep in mind that the result is a new collection—the original collection is not affected.

Alternatively, you could write a.filter(\_ % 2 == 0).map(2 \* \_)

Or even a filter { \_ % 2 == 0 } map { 2 \* \_ }

Some programmers with experience in **functional programming** prefer **filter** and **map** to **guards** and **yield**. That’s just a matter of style—the for/yield loop does exactly the same work. Use whichever you find easier.

In the followig case we want to remove all negative elements from an array buffer of integers. A traditional sequential solution might traverse the array and remove the unwanted elements encountered:

var n = a.length

var i = 0

while (i < n) {

if (a(i) >= 0) i += 1

 else { a.remove(i); n -= 1 }

}

Thats a bit fussy, you have to remember not to increment i when you remove the element, and to decrement n instead. It is also not efficient to remove elements form the middle of the array buffer.

In Scala, best solution is to use for/yield loop or filter/map and keep the all non-negative elements: val result = for (elem <- a if elem >= 0) yield elem

The result is a new array buffer. Suppose that we want to modify the original array buffer instead, removing the unwanted elements. Then we can collect their positions:

val positionsToRemove = for (i <- a.indices if a(i) < 0) yield i

The key observation is that it is better to have all index values together instead of seeing them one by one.

**Common algorithms**

It is often said that a large percentage of business computations are nothing but computing sums and sorting. Fortunately, Scala has built-in functions for these tasks.

Array(1, 7, 2, 9).sum  // 19 // Works for ArrayBuffer too

In order to use the sum method, the element type must be a numeric type: either an integral or floating-point type or BigInteger/BigDecimal. Similarly, the min and max methods yield the smallest and largest element in an array or array buffer.

ArrayBuffer("Mary", "had", "a", "little", "lamb").max  // "little"

The sorted method sorts an array or array buffer and returns the sorted array or array buffer, without modifying the original:

val b = ArrayBuffer(1, 7, 2, 9).sorted // ArrayBuffer(1, 2, 7, 9)

You can also supply a comparison function, but then you should use the sortWith method:

val bDescending = b.sortWith(\_ > \_) // ArrayBuffer(9, 7, 2, 1)

val a = Array(1, 7, 2, 9)

Finally, if you want to display the contents of an array or array buffer, the mkString method lets you specify the separator between elements. A second variant has parameters for the prefix and suffix. For example,

a.mkString(" and ")  // "1 and 2 and 7 and 9

"a.mkString("<", ",", ">")  // "<1,2,7,9>"

In contrast with toString:

a.toString //"[I@b73e5" //This is a useless toString method from Java

The toString method reports the type, which is useful for debugging

**Desciphering scaladoc**

def count(p: (A) => Boolean): Int // a.count(\_ > 0)

def append( elems: A\*): Unit // b.append(1,2,3,4)

def appendAll(xs: TraversableOnce[A]): Unit //

def containsSlice[B](that: GenSeq[B]): Boolean

def += (elem: A): ArraytBuffer.this.type

def copyToArray[B >: A](xd: Array[B]): Unit

def sorted[B >: A](implicit cmp: Ordering[B]): ArrayBuffer[A]

def ++:[B >: A, That](that: Collection.Traversable[B] (implicit bf: CanBuildFrom[ArrayBuffer[A], B, That]): That

**Multidimensional arrays**

Multidimensional arrays are implemented as arrays of arrays. For example, a two dimensoional array of Double values has the type Array[Array[Double]]. To construct such an array, use ofDim method: val matrix = Array.ofDim[Double](3, 4) // Three rows, four columns

To access an element, use two pairs of parentheses:

matrix(row)(column) = 42

You can make ragged arrays, with varying row lengths:

val triangle = new Array[Array[Int]](10)for (i <- triangle.indices)  triangle(i) = new Array[Int](i + 1)

**Interoperationg with java**

In Java, an array of a given type is automatically converted to an array of a supertype. For example, a Java String[] array can be passed to a method that expects a Java Object[] array. Scala does not permit this automatic conversion because it is unsafe. (See Chapter 18 for a detailed explanation.)

Suppose you want to invoke a Java method with an Object[] parameter, such as java.util.Arrays.binarySearch(Object[] a, Object key):

val a = Array("Mary", "a", "had", "lamb", "little")java.util.Arrays.binarySearch(a, "beef") // Does not work

This does not work because Scala will not convert an Array[String] into an Array[Object]. You can force the conversion like this:

java.util.Arrays.binarySearch(**a.asInstanceOf[Array[Object]]**, "beef")

This is just an example to show how to overcome element type differences. If you want to carry out binary search in Scala, do it like this:

import scala.collection.Searching.\_val result = a.search("beef")

If you call a Java method that receives or returns a java.util.List, you could, of course, use a Java ArrayList in your Scala code—but that is **unattractive**. Instead, import the implicit conversion methods in scala.collection.JavaConversions. Then you can use Scala buffers in your code, and they automatically get wrapped into Java lists when calling a Java method. Conversely, when a Java method returns a java.util.List, you can have it automatically converted into a Buffer. val listResult : Buffer[String]

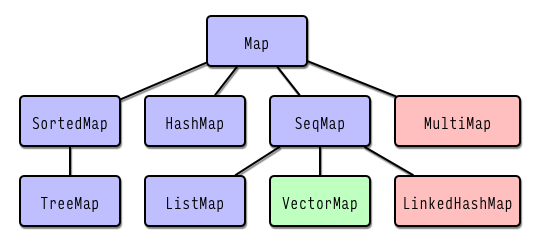
***Maps and Tuples***

You need to choose between mutable and immutable maps.

By defauld, you get a hash map, but you can also get a tree map

Tuples are useful for aggregating values

The class hierarchy for Map is represente don the following diagram:



**Construct a map**

This constructs an immutable Map[String, Int] whose contents can’t be changed.

val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)

If you want a mutable map, use:

val scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3)

If you want to start out with a blank map, you have to supply type parameters:

val scores = scala.collection.mutable.Map[String, Int]()

In Scala, a map is a **collection of pairs**. A pair is simply a grouping of two values, not necessarily of the same type, such as ("Alice", 10). The -> operator makes a pair.

You could have equally well defined the map as:

val scores = Map(("Alice", 10), ("Bob", 3), ("Cindy", 8))

The -> operator is just a little easier on the eyes than the parentheses.

**Accessing map values**

In Scala, the analogy between functions and maps is particularly close because you use the () notation to look up key values.

val bobsScore = scores("Bob") // Like scores.get("Bob") in Java

If the map doesn’t contain a value for the requested key, an exception is thrown.To check whether there is a key with the given value, call the contains method:

val bobsScore = if (scores.contains("Bob")) scores("Bob") else 0

Since this call combination is so common, there is a shortcut:

val bobsScore = scores.**getOrElse**("Bob", 0) //If the map contains the key "Bob", return the value; otherwise, return 0.

Finally, the call map.get(key) returns an Option object that is either Some (value for key) or None. We discuss the Option class in Chapter 14.

Given an immutable map, you can turn it into a map with a fixed default value for keys that are not present, or a function to compute such values.

val scores1 = scores.withDefaultValue(0)

val zeldasScore1 = scores1.get("Zelda")

//Yields 0 since "Zelda" is not present

val scores2 = scores.withDefault(\_.length)

val zeldasScore2 = scores2.get("Zelda")

//Yields 5, applying the length to the key that is not present

**Updating Map Values**

In a mutable map, you can update a map value, or add a new one, with a () to the left of an = sign: (assuming scores is mutable)

scores("Bob") = 10 // Updates the existing value for the key "Bob" scores("Fred") = 7  // Adds a new key/value pair to scores

Alternatively, you can use the += operation to add multiple associations:

scores += ("Bob" -> 10, "Fred" -> 7)

To remove a key and its associated value, use the -= operator:

scores -= "Alice" //for var

scores = scores - "Alice" //for val

You can’t update an immutable map, but you can do something that’s just as useful—obtain a new map that has the desired update:

val newScores = scores + ("Bob" -> 10, "Fred" -> 7)//New updated map

The newScores map contains the same associations as scores, except that "Bob" has been updated and "Fred" added.

You might think that it is **inefficient** **to keep constructing new maps**, but that is **not** the case. The old and new maps **share most of their structure**. (This is possible because they are immutable.)

**Iterationg over Maps**

The following amazingly simple loop iterates over all key/value pairs of a map:

for ((k, v) <- map) process k and v

The magic here is that you can use pattern matching in a Scala for loop. (Chapter 14 has all the details.) That way, you get the key and value of each pair in the map without tedious method calls.

If for some reason you want to visit only the keys or values, use the keySet and values methods, as you would in Java. The values method returns an Iterable that you can use in a for loop.

scores.keySet // A set such as Set("Bob", "Cindy", "Fred", "Alice")

for (v <- scores.values) println(v) // Prints 10 8 7 10 or some permutation thereof

To reverse a map—that is, switch keys and values—use

for ((k, v) <- map) yield (v, k)

**Sorted Maps**

There are two common implementation strategies for maps: hash tables ‘HashMap’ and balanced tres ‘TreeMap’. Hash tables use the hash codes of the keys to scramble entries, so iterating over the elements yields them in unpredictable order. By default, Scala gives you a map based on a hash table because it is usually more efficient. If you need to visit the keys in sorted order, use a SortedMap instead.

val scores = scala.collection.mutable.SortedMap("Alice" -> 10,  "Fred" -> 7, "Bob" -> 3, "Cindy" -> 8)

If you want to visit the keys in insertion order, use a LinkedHashMap.

val months = scala.collection.mutable.LinkedHashMap("January" -> 1,  "February" -> 2, "March" -> 3, "April" -> 4, "May" -> 5, ...)

**Interoperationg with Java**

If you get a Java map from calling a Java method, you may want to convert it to a Scala map so that you can use the pleasant Scala map API.

Simply add an import statement:

import scala.collection.JavaConversions.mapAsScalaMap

Then trigger the conversion by specifying the Scala map type:

val scores: scala.collection.mutable.Map[String, Int] =  new java.util.TreeMap[String, Int]

Conversely, to pass a Scala map to a method that expects a Java map, provide the opposite implicit conversion. For example:

import scala.collection.JavaConversions.mapAsJavaMapimport java.awt.font.TextAttribute.\_ // Import keys for map below

val attrs = Map(FAMILY -> "Serif", SIZE -> 12) // A Scala map

val font = new java.awt.Font(attrs) // Expects a Java map

**Tuples**

Maps are collections of key/value pairs. Pairs are the simplest case of tuples, aggregates of values of different types. A tuple value is formed by enclosing individual values in parentheses. (1, 3.14, "Fred")//tuple of type (Int, Double, String)

If you have a tuple, say, val t = (1, 3.14, "Fred")

then you can access its components with the methods \_1, \_2, \_3,:

val second = t.\_2 // Sets second to 3.14

Unlike array or string positions, the component positions of a tuple start with 1, not 0.

You can write t.\_2 as t \_2

Usually, it is better to use pattern matching to get at the components of a tuple:

val (first, second, third) = t // Sets first to 1, second to 3.14, third to "Fred"

You can use \_ if you don’t need all components:

val (first, second, \_) = t

Tuples are useful for functions that return more than one value. For example, the partition method of the StringOps class returns a pair of strings, containing the characters that fulfill a condition and those that don’t:

**Zipping**

One reason for using tuples is to bundle together values so that they can be processed together. This is commonly done with the zip method.

val symbols = Array("<", "-", ">")

val counts = Array(2, 10, 2)

val pairs = symbols.zip(counts)//Array(("<", 2), ("-", 10), (">", 2))

It yields an array of pairs thath can be processed together

for ((s, n) <- pairs) print(s \* n) // Prints <<---------->>

The toMap method turns a collection of pairs into a map.Then, if you have a collection of keys and a parallel collection of values, zip them up and turn them into a map: keys.zip(values).toMap

***CLASSES***

Fields in clases automatically come with getters an 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 getX/setX

Every class has a primary constructor that is “interwoven” with the class definition. Its parameters turn into the fields of the class.

Auxiliary constructors are optional. They are called this.

**Simple clases and parameterless methods**

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

class Counter {

private var value = 0 // You must initialize the field

def increment() { value += 1 } // Methods are public by default

def current() = value

}

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

val myCounter = new Counter // Or new Counter()

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

myCounter.current // OK

myCounter.current() // Also OK

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).

myCounter.**increment()** // Use () with mutator

println(myCounter.**current**) // Don't use () with accessor

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

def current = value // No () in definition

**Properties with getters and setters**

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

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;}

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\_=. :

println(fred.age) // Calls the method

fred.age()fred.age = 21 // Calls fred.age\_=(21)

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

As you can see, the compiler created methods private int age,public in age, public void age\_$eq and public Person() (The = symbol is translated to $eq because the JVM does not allow an = in a method name.)

At any time, you can redefine the getter and setter methods yourself:

class Person {

private var privateAge = 0 // Make private and rename

def age = privateAge

def age\_=(newVal: Int){if (newVal > privateAge) privateAge = newVal;}

}

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

**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

}

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.

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

**Properties with only getters**

In Scala, 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

}

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 other.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 called **object-private**.

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.

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.)

**Bean properties**

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

import scala.beans.BeanProperty

class Person {

  @BeanProperty var name: String = \_

}

That generates four methods:

1. name: String

2. name\_=(newValue: String): Unit

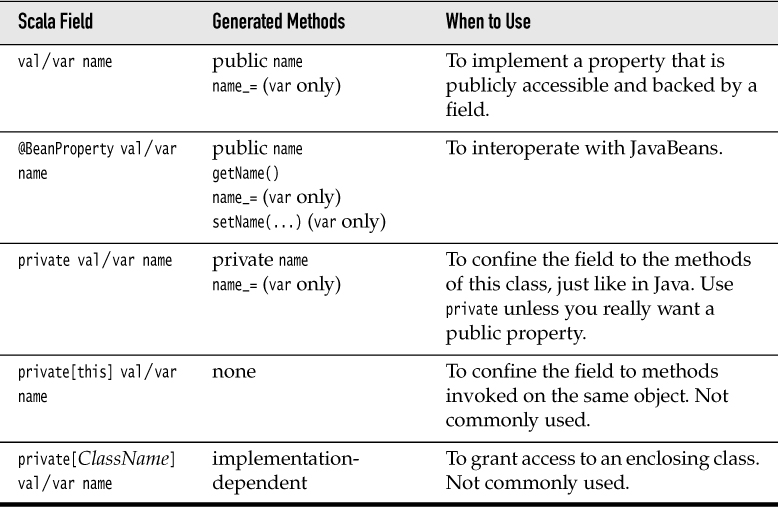
3. getName(): String

4. setName(newValue: String): Unit

You can define the field as a primary constructor parameter like:

class Person(@BeanProperty var name: String)

**Summary**



**Auxiliary constructors**

A Scala class can have as many constructors as you like. However, a Scala class has one constructor that is more imortant than all the others, called the **primary constructor**. In addition, a class may have any number of **auxiliary constructors**.

The auxiliary constructors are called this.

Each auxiliary constructor must start with a call to a previously

defined auxiliary constructor or the primary constructor.

Here is a class wit no 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 one

this(name) // Calls previous auxiliary constructor

this.age = age

}

}

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 ‘this()’.

You can construct objects of this class in three ways:

val p1 = new Person // Primary constructorval

p2 = new Person("**Fred**") // First auxiliary constructorval

p3 = new Person("**Fred**", **42**) // Second auxiliary constructor

**The primary costructor**

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

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 (...)  ...

}

In the 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.

The primary constructor executes all statements in the class definition, for example:

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.

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.

Primary constructor parameters can have any of the forms properties with only getters, properties with getter and setter, beanPorpertiesc, class-private and object-private.

Construction parameters can also be regular method parameters, without val or var.

In this case, such a field is equivalent of an object-private, private[this] val fieldName.

If you find the primary constructor notation confusng, you don need to use it. Just provide one more auxiliary constructors, but remember to call this().

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.

**The primary costructor**

In Scala, you can nest just about anything inside anything. You can define functions inside other functions, and clases inside other clases.

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 Networkval

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.

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

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.

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.”

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](https://learning.oreilly.com/library/view/Scala+for+the+Impatient,+Second+Edition/9780134540627/ch19.html#ch19)

In order to acces to variables of the outer class from the inner one, 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.

class Network(val name: String) { outer =>

  class Member(val name: String) {

    ...

    def description = s"$name inside ${outer.name}"

  }

}

***OBJECTS***

Use objects for singletons and utility methods.

A class can have a companion object with the same name

Object can extend clases or traits

The apply method of an object is usually used for constructing new instances of the companion class

To avoid the main method, use an object that extend the App trait.

You can implement enumerations by etending the Enumeration object

**Singletons**

Scala has no **static methods** or fields. Instead, you use the object construct. An object defines a single instance of a class with the features that you want. For example

object Accounts {

  private var lastNumber = 0

  def newUniqueNumber() = { lastNumber += 1; lastNumber }

}

When you need a new uique number just call Accounts.newUniqueNumber()

The constructor of an object is executed when the object is first used, in this case the first call to Accounts.newUniqueNumber(.

An object can have essentially all the features of a class, it can even extend other clases or traits. There is just one exception: You can not provide constructor parameters.

Use an object as a home for utility functions or constants, when a single immutable instance can be shared efficiently, when a single instance is required to coordinate some service (Singleton).

**Companion Object**

You often have a class with both instance methods and static methods. In Scala, you can achieve this by having a class and a **companion object** of the same.

class Account {

  val id = Account.newUniqueNumber()

  private var balance = 0.0

  def deposit(amount: Double) { balance += amount }

  ...

}

object Account { // The companion object

  private var lastNumber = 0

  private def newUniqueNumber() = { lastNumber += 1; lastNumber }

}

The class and its companion object can Access each other’s private features. They must be located in the same source file.

Note thet the companion object’s features are not in the scope of the class. For example, the Account class has to use Account.newUniqueNumber() and not just newUniqueNumber() to invoke the method of the companion object.

**Objects extending a class or trait**

An object can extend a class and/or one or more traits. The result is an object of a class that extends the given class and/or traits, and in addition has all of the features specified in the object definition.

One useful application is to specify default objects that can be shared. For example, consider a class for undoable actions in programs.

abstract class UndoableAction(val description: String) {

  def undo(): Unit

  def redo(): Unit

}

A useful default is the “do nothing” action. Of course, we only need one of them.

object DoNothingAction extends UndoableAction("Do nothing") {

  override def undo() {}

  override def redo() {}

}

The DoNothingAction object can be shared across all places that need this default.

val actions = Map("open" -> DoNothingAction, "save" -> DoNothingAction, ...)  // Open and save not yet implemented

**The apply method**

It is common to have objects with an apply method. The apply method is called for expressions of the form Object(arg1, ..., argN)

Typically, such an apply method returns an object of the companion class.For example, the Array object defines apply methods that allow array creation with expressions such as Array("Mary", "had", "a", "little", "lamb") Why doesn’t one just use a constructor? Not having the new keyword is handy for nested expressions, such as:

Array(Array(1, 7), Array(2, 9))

It is easy to confuse Array(100) and new Array(100). The first expression calls apply(100), yielding an Array[Int] with a single element, the integer 100. The second expression invokes the constructor this(100). The result is an Array[Nothing] with 100 null elements.

Here is an example of defining an apply method:

class Account private (val id: Int, initialBalance: Double) {

  private var balance = initialBalance

  ...

}

object Account { // The companion object

  def apply(initialBalance: Double) =

    new Account(newUniqueNumber(), initialBalance)

  ...

}

Now you can construct an account as val acct = Account(1000.0)

**Application objects**

An Scala program must start with an object’s method def main(args: Array[String])

Instead of providing a main method for your application, you can extend the App trait and place the program code into the constructor body: object Hello **extends** **App**

If you need the command-line arguments, you can get them from the args property.

If you invoke the application with the scala.time option set, then the elapsed time is displayed when the program exits.

scala -Dscala.time Hello Fred

**Ennumeration**

Scala does not have enumerated types. However, the standard library provides an Enumeration helper class that you can use to produce enumerations.

Define an object that extends the Enumeration class and initialize each value in your enumeration with a call to the Value method.

object TrafficLightColor extends Enumeration {

val Red, Yellow, Green = Value

}

Here we define three fields, Red, Yellow, and Green, and initialize each of them with a call to Value. This is a shortcut for:

val Red = Value

val Yellow = Value

val Green = Value

You can now refer to the enumeration values as TrafficLightColor.Red, TrafficLightColor.Yellow, and so on.

Remember that the type of the enumeration is TrafficLightColor.Value and not TrafficLightColor—that’s the type of the object holding the values. Some people recommend that you add a type alias.

object TrafficLightColor extends Enumeration {

**type TrafficLightColor = Value**

  val Red, Yellow, Green = Value

}

***PACKAGES & IMPORTS***

Package paths are not absolute

Package statements without braces at the top of the file extend to the entire file.

A chain x.y.z in a package clause leaves the intermediate packages x and x.y invisible.

A package object can hold functions and variables.

Import statements can import packages, classes, and objects.

Import statements can be anywhere.

java.lang, scala, and Predef are always imported.

Import statements can rename and hide members.

**Chained Package Clauses**

A package clause can contain a “chain,” or path segment, for example:

package com.horstmann.impatient {

  // Members of com and com.horstmann are not visible here

  package people {

    class Person

    ...

  }

}

Such a clause limits the visible members. Now a com.horstmann.collection package would no longer be accessible as collection.

If we want to acces all the package members we coud change it for:

package com {

  package horstmann {

    package collection {

      ...

    }

  }

}

**Package Objects**

A package can contain classes, objects, and traits, but not the definitions of functions or variables. That’s an unfortunate limitation of the Java virtual machine.

It would make more sense to add utility functions or constants to a package than to some Utils object. Package objects address this limitation.Every package can have one package object. You define it in the parent package, and it has the same name as the child package. For example,

package com.horstmann.impatient

package object people {

  val defaultName = "John Q. Public"

}

package people {

  class Person {

   var name = defaultName // A constant from the package

  }

 ...

}

Note that the defaultName value didn’t need to be qualified because it was in the same package. Elsewhere, it’s accessible com.horstmann.impatient.people.defaultName.

**Renaming and hiding names**

If you want to import more than one member from a package, use a selector like this: import java.awt.{Color, Font}

The selector syntax lets you rename members:

import java.util.{HashMap => JavaHashMap}

The selector HashMap => \_ hides a member instead of renaming it. This is only useful if you import others:

import java.util.{HashMap => \_, \_}import scala.collection.mutable.\_

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

**Implicit Imports**

Every Scala program implicitly starts with

import java.lang.\_

import scala.\_

import Predef.\_

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

collection.mutable.HashMap

***INHERITANCE***

The extends and final keywords are as in Java.

You must use override when you override a method.

Only the primary constructor can call the primary superclass constructor.

**Overriding**

To extend a class just use the extends keyword

class Employee **extends** Person {

  var salary = 0.0

  ...

}

Specify fields and methods that are new to the subclass or that override methods in the superclass. Declaring a class as final so that it cannot be extended. You can also declare individual methods or fields final so that they cannot be overridden.

Note that this is different from Java, where a final field is immutable, similar to val in Scala.

You must use the override modifier when you override a method that isn’t abstract.

override def toString = s"${getClass.getName}[name=$name]"

Invoking a superclass method in Scala works exactly like in Java, with the keyword super:

override def toString = s"${**super**.toString}[salary=$salary]"

Note the following restrictions

- A def can only override another def.

- A val can only override another val or a parameterless def.

override val toString = "secret"

- A var can only override an abstract var.

**Type checks and casts**

To test whether an object belongs to a given class, use the isInstanceOf method. If the test succeeds, you can use the asInstanceOf method to convert a reference to a subclass reference:

if (p.isInstanceOf[Employee]) {

  val s = p.asInstanceOf[Employee] // s has type Employee

  ...

}

The p.isInstanceOf[Employee] test succeeds if p refers to an object of class Employee or its subclass (such as Manager).

If p is null, then p.isInstanceOf[Employee] returns false and p.asInstanceOf[Employee] returns null.

If p is not an Employee, then p.asInstanceOf[Employee] throws an exception.

If you want to test whether p refers to an Employee object, but not a subclass, use:

if (p.getClass == classOf[Employee])

The classOf method is defined in the scala.Predef object that is always imported.

However, pattern matching is usually a better alternative to using type checks and casts. For example,

p match {

  case s: Employee => ... // Process s as an Employee

  case \_ => ... // p wasn't an Employee

}

**Protected fields and methods**

A member declared as protected, is accessible from any subclass, but not from other locations.

A protected member is not visible throughout the package to which the class belongs.

There is also a protected[this] variant that restricts access to the current object, similar to the private[this]

**Superclass construction**

An auxiliary constructor can never invoke a superclass constructor directly.

The auxiliary constructors of the subclass eventually call the primary constructor of the subclass. Only the primary constructor can call a superclass constructor:

class Employee(name: String, age: Int, val salary : Double) extends  
Person(name, age)

Intertwining the class and the constructor makes for very concise code. You may find it helpful to think of the primary constructor parameters as parameters of the class. Here, the Employee class has three parameters: name, age, and salary, two of which it “passes” to the superclass. In Java, the equivalent code is quite a bit more verbose:

public class Employee extends Person { // Java

  private double salary;

  public Employee(String name, int age, double salary) {

    super(name, age);

    this.salary = salary;

  }

}

In a Scala constructor, you can never call super(params), as you would in Java, to call the superclass constructor.

**Anonymous subclasses**

An instance of an **anonymous** subclass if you include a block with definitions or overrides, such as:

**val** alien = **new** Person(**"Fred"**) {  
 **def** greeting = **"Greetings, Earthling! My name is Fred."**}

Technically, this creates an object of a structural type. The type is denoted as Person{def greeting: String}. You can use this type as a parameter type:

**def** meet(p: Person{**def** greeting: String}) {  
 println(**s"$**{p.name} **says: $**{p.greeting}**"**)  
}

**Abstract clases**

Use can use the abstract keyword to denote a class that cannot be instantiated, usually because one or more of its methods are not defined. For example,

**abstract class** Person(**val** name: String) {  
 **def** id: Int *// No method body—this is an abstract method* **var** name: String  
}

Each concrete subclass of Person needs to specify an id method. In Scala you do not use the abstract keyword for an abstract method or fields. You simply omit its body.

In a subclass, you need not use the override keyword when you define a method that was abstract in the superclass.

**class** Employee(name: String) **extends** Person(name) {  
 **def** id = name.hashCode *// override keyword not require* **var** name = **""**}

**Abstract fields**

In addition to abstract methods, a class can also have abstract fields. An abstract field is simply a field without an initial value. For example:

**abstract class** Person {  
 **val** id: Int *// Abstract field with an abstract getter method* **var** name: String *//Another abstract field with getter and setter*}

Concrete subclasses must provide concrete fieldsfe, for example:

**class** Employee(**val** id: Int) **extends** Person {*//Subclass has concrete id property***var** *name* = **""** *//and concrete name property*}

As with methods, no override keyword is required in the subclass when you define a field that was abstract in the superclass.

**Construction order and early definitions**

When you override a val in a subclass and use the value in a superclass constructor, the resulting behavior is unintuitive. Here is an example

**class** Creature {  
 **val** *range*: Int = 10  
 **val** *env*: Array[Int] = **new** Array[Int](*range*)  
}

**class** Ant **extends** Creature {  
 **override val** *range* = 2  
}

Unfortunately, we now have a problem. The range value is used in the superclass constructor, and the superclass constructor runs before the subclass constructor. Specifically, here is what happens:

1. The Ant constructor calls the Creature constructor before doing its own construction.

2. The Creature constructor sets its range field to 10.

3. The Creature constructor, in order to initialize the env array, calls the range() getter.

4. That method is overridden to yield the (as yet uninitialized) range field of the Ant class.

5. The range method returns 0. (That is the initial value of all integer fields when an object is allocated.)

6. env is set to an array of length 0.

7. The Ant constructor continues, setting its range field to 2.

Even though it appears as if range is either 10 or 2, env has been set to an array of length 0. The moral is that you should not rely on the value of a val in the body of a constructor.

There are several remedies.

-Declare the val as final. This is safe but not very flexible.

-Declare the val as lazy in the superclass.

-This is safe but a bit inefficient.

-Use the **early definition** syntax in the subclass—see below.

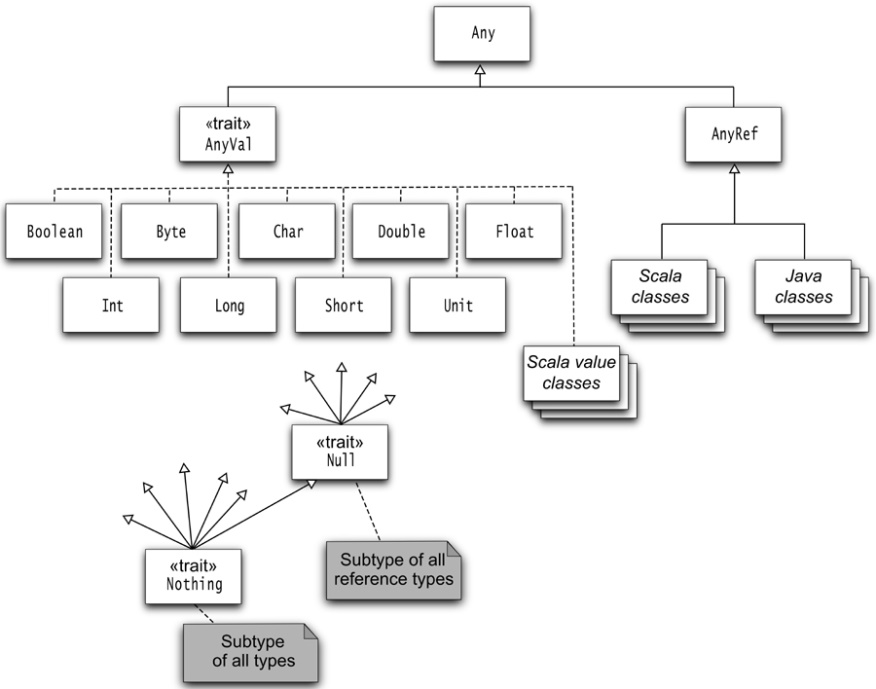
The “early definition” syntax lets you initialize val fields of a subclass *before* the superclass is executed. The syntax is so ugly that only a mother could love it. You place the val fields in a block after the extends keyword, like this:

**class** Ant **extends** { **override val** *range* = 2 } **with** Creature

Note the with keyword before the superclass name. This keyword is normally used with traits.

**Construction order and early definitions**

The below figure shows the inheritance hierarchy of Scala classes. The classes that correspond to the primitive types in Java, as well as the type Unit, extend AnyVal. You can also define your own value clases.



All other clases are subclases of the AnyRef class. When compiling to the Java virtual machine, this is a synonym for the java.lang.Object class.

Both AnyVal and AnyRef extend the Any class, the root of the hierarchy.

The Any class defines methods isInstanceOf, asInstanceOf, and the methods for equality and hash codes (see in next sub section).

AnyVal does not add any methods. It is just a marker for value types.

The AnyRef class adds the monitor methods wait and notify/notifyAll and synchronized from the Object class. (Stay away from these methods unless have a good reason)

All Scala classes implement the marker interface ScalaObject, which has no methods.

At the other end of the hierarchy are the Nothing and Null types.

Null is the type whose sole instance is the value null. You can assign null to any reference, but not to one of the value types. For example, setting an Int to null is not possible. This is better than in Java, where it would be possible to set an Integer wrapper to null.

The Nothing type has no instances. It is occasionally useful for generic constructs. For example, the empty list Nil has type List[Nothing], which is a subtype of List[T] for any T.

The ??? method is declared with return type Nothing. It never returns but instead throws a NotImplementedError when invoked. You can use it for methods that you still need to implement:

**class** Person(**val** name: String) {  
 **def** description = *???*}

The Person class compiles since Nothing is a subtype of every type. You can start using the class, so long as you don’t call the description method.

The Nothing type is not at all the same as the Unit type, the type with the sole value (). Note that Unit is not a supertype of any other type. However, the compiler still allows any value to be replaced by a (). Consider

**def** printAny(x: Any) { c(x) }  
**def** printUnit(x: Unit) { println(x) }  
printAny(**"Hello"**) *// Prints Hello*printUnit(**"Hello"**) *// Replaces "Hello" with () and calls printUnit(()), which prints ()*

When a method has a parameter of type Any or AnyRef, and it is called with multiple arguments, then they are placed in a tuple:

**def** show(o: Any) { println(**s"$**{o.getClass}**: $**o**"**) }  
show(3) *// Prints class java.lang.Integer: 3*show(3, 4, 5) *// Prints class scala.Tuple3: (3,4,5)*

**Object equality**

In Scala, the eq method of the AnyRef class checks whether two references refer to the same object. The equals method in AnyRef calls eq. When you implement a class, you should consider overriding the equals method to provide a natural notion of equality for your situation.For example, if you define, **class** Item(**val** description: String, **val** price: Double), might want to consider two items equal if they have the same description and price. Here is an appropriate equals method:

**final override def** equals(other: Any) = {  
 other.isInstanceOf[Item] && {  
 **val** that = other.asInstanceOf[Item]  
 description == that.description && price == that.price  
 }  
}

Or even better, use pattern matching:

**final override def** equals(other: Any) = other **match** {  
 **case** that: Item => description == that.description && price == that.price  
 **case** \_ => **false**}

We defined the method as final because it is generally very difficult to correctly extend equality in a subclass. The problem is symmetry. You want a.equals(b) to have the same result as b.equals(a), even when b belongs to a subclass.

**final def** equals(other: Item) = { ... } *// Don't!*

**final def** ==(other: Item) = { ... } *// Don't!*

When you define equals, remember to define hashCode as well. The hash code should be computed only from the fields that you use in the equality check, so that equal objects have the same hash code. In the Item example, combine the hash codes of the fields.

**final override def** hashCode = (description, price).##

The ## method is a null-safe version of the hashCode method that yields 0 for null instead of throwing an exception.

In an application program, you don’t generally call eq or equals. Simply use the == operator. For reference types, it calls equals after doing the appropriate check for null operands.

**Value classes**

Some classes have a single field, such as the wrapper classes for primitive types, and the “rich” or “ops” wrappers that Scala uses to add methods to existing types. It is inefficient to allocate a new object that holds just one value. *Value classes* allow you to define classes that are “inlined,” so that the single field is used directly.

A value class has these properties:

**1.** The class extends AnyVal.

**2.** Its primary constructor has exactly one parameter, which is a val, and no body.

**3.** The class has no other fields or constructors.

**4.** The automatically provided equals and hashCode methods compare and hash the underlying value.

As an example, let us define a value class that wraps a “military time” value:

**class** MilTime(**val** time: Int) **extends** AnyVal {  
 **def** minutes = time % 100  
 **def** hours = time / 100  
 **override def** toString = **f"$**time04d**"**}

When you construct a new MilTime(1230), the compiler doesn’t allocate a new object. Instead, it uses the underlying value, the integer 1230. You can invoke the minutes and hours methods on the value but, just as importantly, you cannot invoke Int methods.

MilTime lunch = **new** MilTime(1230)  
println(lunch.hours) *// OK*println(lunch \* 2) *// Error*

To guarantee proper initialization, make the primary constructor private and provide a factory method in the companion object:

**class** MilTime **private**(**val** time: Int) **extends** AnyVal ...

**object** MilTime {  
 **def** apply(t: Int) =  
 **if** (0 <= t && t < 2400 && t % 100 < 60) **new** MilTime(t)  
 **else throw new** IllegalArgumentException  
}

In some programming languages, value types are any types that are allocated on the runtime stack, including structured types with multiple fields. In Scala, a value class can only have one field. If you want a value class to implement a trait, the trait must explicitly extend Any, and it may not have fields. Such traits are called universal traits.

Value types were designed to make implicit conversions efficient, but you can use them for your own overhead-free “tiny types.” For example, instead of a **class** Book(**val** author: String, **val** title: String), you can wrap each string into a separate value class Author and Title. When the class is defined as class Book(**val** author: **Author**, val title: **Title**), programmers who construct Book objects can’t accidentally switch the author and the title.

**Traits**

-A class can implement any number of traits.

-Traits can require implementing clases to have certain fields, methods, or subclases.

-Unlike Java interfaces, a Scala trait can provide implementations of methods and fields.

-When you layer multiple traits, the order matters, the trait whose methods execute first goes to the back.

**Traits as interfaces**

Scala does not allow a class to inherit from multiple superclases. At first, this seems like an unfortunate restriction. That’s because multiple inheritance works fine when you combine clases that have nothing in common. But if these clases have common methods or fields, thorny issues come up. So, in Scala, a class can extend only one superclass; and it has *traits* instead of interfaces. A trait can have abstract and concrete methods, as well as state, and a class can implement multiple traits.

**trait** Logger {  
 **def** log(msg: String) *// An abstract method* **def** printString(s: String) { println(s) }

}

Note that you need not declare the method as abstract—an unimplemented method in a trait is automatically abstract. A subclass can provide an implementation:

**class** ConsoleLogger **extends** Logger { *// Use extends, not implements* **def** log(msg: String) { *println*(msg) } *// No override needed*}

You need not supply the override keyword when overriding an abstract method of a trait.

If you need more than one trait, add the others using the with keyword:

**class** ConsoleLogger **extends** Logger **with** Cloneable **with** Serializable

Here we use the Cloneable and Serializable interfaces from the Java library, just for the sake of showing the syntax. All Java interfaces can be used as Scala traits.

The methods oof a trait need not be abstract, as **def** printString(s: String) { println(s)}. We say that the Logger functionality is “mixed in” whith ConsoleLogger.

**Objects with traits**

You can add a trait to an individual object when you construct it. Let’s first define this class:

**abstract class** SavingsAccount **extends** Account **with** Logger {  
 **def** withdraw(amount: Double) {  
 **if** (amount > balance) log(**"Insufficient funds"**)  
 **else** ...  
 }  
 ...  
}

This class is abstract since it can’t yet do any logging, which might seem pointless. But you can “mix in” a concrete logger when constructing an object.

**trait** ConsoleLogger **extends** Logger {  
 **def** log(msg: String) { *println*(msg) }  
}

**val** acct = **new** SavingsAccount **with** ConsoleLogger

Of course, another object can add in a different trait:

**val** acct2 = **new** SavingsAccount **with** FileLogger

**Layered traits**

You can add, to a class ora n object, multiple traits that invoke each other starting with the last one. This is useful when you need tto transform a value in stages. Here is a simple example. We may want to add a timestamp to all logging messages.

**trait** TimestampLogger **extends** ConsoleLogger {  
 **override def** log(msg: String) {  
 **super**.log(**s"$**{java.time.Instant.*now*()} **$**msg**"**)  
 }  
}

Also, suppose we want to truncate overly chatty log messages like this:

**trait** ShortLogger **extends** ConsoleLogger {  
 **override def** log(msg: String) {  
 **super**.log(  
 **if** (msg.length <= 15) msg **else s"$**{msg.substring(0, 12)}**..."**)  
 }  
}

Note that each of the log methods passes a modified message to super.log.With traits, super.log does not have the same meaning as it does with classes. Instead, super.log calls the log method of another trait, which depends on the order in which the traits are added.

To see how the order matters, compare the following two examples:

**val** acct1 = **new** SavingsAccount **with** TimestampLogger **with** ShortLogger

Sun Feb 06 17:45:45 ICT 2011 Insufficient...  
**val** acct2 = **new** SavingsAccount **with** ShortLogger **with** TimestampLogger

Sun Feb 06 1...

With traits, you cannot tell from the source code which method is invoked by super.someMethod. The exact method depends on the ordering of the traits in the object or class that uses them. This makes super far more flexible than in plain old inheritance.

If you want to control which trait’s method is invoked, you can specify it in brackets: super[ConsoleLogger].log(...). The specified type must be an immediate supertype; you can’t access traits or classes that are further away in the inheritance hierarchy.

**Overriding abstracct methods in traits**

In the preceding section, the TimestampLogger and ShortLogger traits extended ConsoleLogger. Let’s make them extend our Logger trait instead, where we provide *no implementation* to the log method.

**trait** Logger {  
 **def** log(msg: String) *// This method is abstract*}

Then, the TimestampLogger class no longer compiles.

**trait** TimestampLogger **extends** Logger {  
 **override def** log(msg: String) { *// Overrides an abstract method* **super**.log(**s"$**{java.time.Instant.*now*()} **$**msg**"**) *// Is super.log defined?* }  
}

Scala takes the position that TimestampLogger.log is still abstract—it requires a concrete log method to be mixed in. You therefore need to tag the method with the abstract keyword and the override keyword, like this:  
**abstract override def** log(msg: String) {  
 **super**.log(**s"$**{java.time.Instant.now()} **$**msg**"**)  
}

**Traits for rich interfaces**

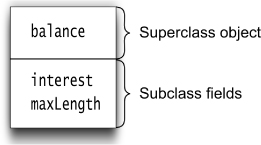
A trait can have many utility methods that depend on a few abstract ones. One example is the Scala Iterator trait that defines dozens of methods in terms of the abstract next and hasNext methods. Let us enrich our rather anemic logging API. Usually, a logging API lets you specify a level for each log message to distinguish informational messages from warnings or errors. We can easily add this capability without forcing any policy for the destination of logging messages.

**trait** Logger {  
 **def** log(msg: String)  
 **def** info(msg: String) { log(**s"INFO: $**msg**"**) }  
 **def** warn(msg: String) { log(**s"WARN: $**msg**"**) }  
 **def** severe(msg: String) { log(**s"SEVERE: $**msg**"**) }  
}

**Concrete fields in traits**

A field in a trait can be concrete or abstract. If you supply an initial value, the field is concrete.

**trait** ShortLogger **extends** Logger {  
 **val** *maxLength* = 15 *// A concrete field* **abstract override def** log(msg: String) {  
 **super**.log(  
 **if** (msg.length <= *maxLength*) msg  
 **else s"$**{msg.substring(0, *maxLength* - 3)}**..."**)  
 }  
}

You can think of concrete trait fields as “assembly instructions” for the classes that use the trait. Any such fields become fields of the class.

**class** Account {  
 **var** *balance* = 0.0  
}

**class** SavingsAccount **extends** Account **with** ShortLogger {  
 **var** *interest* = 0.0  
}

**Abstract fields in traits**

An uninitialized field in a trait is abstract and must be overridden in a concrete subclass.

**trait** ShortLogger **extends** Logger {  
 **val** maxLength: Int *// An abstract field* **abstract override def** log(msg: String) { ... }  
 **super**.log(  
 **if** (msg.length <= **maxLength**) msg else msg.substr(0,maxLenght -3)  
 *// The maxLength field is used in the implementation*}

When you use this trait in a concrete class, you must supply the maxLength field:

**class** SavingsAccount **extends** Account **with** ShortLogger {  
 **val** *maxLength* = 20 *// No override necessary*

}

**Trait construction order**

Just like classes, traits can have constructors, made up of field initializations and other statements in the trait’s body. For example,

**trait** FileLogger **extends** Logger {  
 **val** *out* = **new** PrintWriter(**"app.log"**)*//Part of the trait's constructor  
 out*.println(**s"# $**{java.time.Instant.now()}**"**) *//Also constructor* **def** log(msg: String) { out.println(msg); out.flush() }  
}

These statements are executed during construction of any object incorporating the trait.

Constructors execute in the following order:

1. The superclass constructor is called first.

2. Trait constructors are executed after the superclass constructor but before the class constructor.

3. Traits are constructed left-to-right.

4. Within each trait, the parents get constructed first.

5. If multiple traits share a common parent, and that parent has already been constructed, it is not constructed again.

6. After all traits are constructed, the subclass is constructed.

For example, consider this class:

**class** SavingsAccount **extends** Account **with** FileLogger **with** ShortLogger

The constructors execute in the following order:

1. Account (the superclass).

2. Logger (the parent of the first trait).

3. FileLogger (the first trait).

4. ShortLogger (the second trait). Note that its Logger parent has already been constructed.

5. SavingsAccount (the class).

**Initializing trait fields**

Traits cannot have constructor parameters. Every trait has a single parameterless constructor.

Interestingly, the absence of constructor parameters is the only technical difference between traits and classes. Otherwise, traits can have all the features of classes, such as concrete and abstract fields and superclasses.

**val** acct = **new** SavingsAccount **with** FileLogger(**"myapp.log"**)   
 *// Error: Can't have constructor parameters for traits*

**trait** FileLogger **extends** Logger {  
 **val** filename: String  
 **def** log(msg: String) { *out*.println(msg); *out*.flush() }  
}

A class using this trait can override the filename field. Unfortunately, there is a pitfall. The straightforward approach does not work:

**class** SavingsAccount **extends** Account **with** FileLogger { **val** filename = **"myapp.log"**} *// Does not work*

{...}

The problem is the construction order. The FileLogger constructor runs before the subclass constructor. One remedy is an obscure feature that we described in Chapter `Inheritance´: **early definition**. Here is the correct version:

**class** SavingsAccount **extends** {**val** filename = **"myapp.log"**} **with** Account **with** FileLogger {...}

or

**val** savingsAccount= **new** {**val** filename = **"myapp.log"**} **with** Account **with** FileLogger *// AnonymousClass*

Another alternative is to use a *lazy value* in the FileLogger constructor, like the lazy value for getting the table fileds in the ingestion\_framework, since the tables were not being created yet, then without the lazy statement these would cause the fail.

**trait** FileLogger **extends** Logger {  
 **val** filename: String  
 **lazy val** *out* = **new** PrintStream(filename)  
 **def** log(msg: String) { *out*.println(msg) } *// No override needed*}

Then the out field is initialized when it is first used. At that time, the filename field will have been set. However, lazy values are somewhat inefficient because they are checked for initialization before every use.

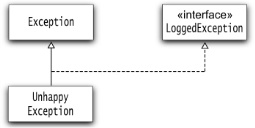
**Traits extending clases**

As you have seen, a trait can extend aother trait, and it is common to have a hierarchi of traits. Less commonly, a trait can also extend a class. That class becomes a superclass of any class mixing the trait. The LoggedException trait extends the Exception class:

**trait** LoggedException **extends** Exception **with** ConsoleLogger {  
 **def** log() { log(getMessage()) }  
}

A LoggedException has a log method to log the exception’s message. Note that the log method calls the getMessage method that is inherited from the Exception superclass.

**class** UnhappyException **extends** LoggedException{*//Class extending trait* **override def** getMessage() = **"arggh!"**}

The superclass of the trait becomes the superclass of our class

What if our class already extends another class? That’s OK, as long as it’s a subclass of the trait’s superclass:

**class** UnhappyException **extends** IOException **with** LoggedException

Here UnhappyException extends IOException, which already extends Exception. When mixing in the trait, its superclass is already present, and there is no need to add it.However, if our class extends an unrelated class, then it is not possible to mix in the trait. For example, you cannot form the following class:

**class** UnhappyFrame **extends** JFrame **with** LoggedException

*// Error: Unrelated superclases*

It would be impossible to add both JFrame and Exception as superclasses.

**Self types**

When a trait extends a class, there is a guarantee that the superclass is present in any class mixing in the trait. Scala has an alternate mechanism for guaranteeing this: *self types*.

When a trait starts out with **this**: Type => then it can only be mixed into a subclass of the given type. Let’s use this feature for our LoggedException:

**trait** LoggedException **extends** ConsoleLogger {  
 **this**: Exception =>  
 **def** log() { log(getMessage()) }  
}

Note that the trait does *not* extend the Exception class. Instead, it has a self type of Exception. That means it can only be mixed into subclasses of *Exception*.

10.14 What Happens under the Hood

In the trait’s methods, we can call any methods of the self type. For example, the call to getMessage() in the log method is valid, since we know that this must be an Exception.

If you try to mix the trait into a class that doesn’t conform to the self type, an error occurs.

**val** f = **new** JFrame **with** LoggedException

*// Error: JFrame isn't a subtype of Exception, the self type of LoggedException*

There are a few situations where the self type notation is more flexible than traits with supertypes. Self types can handle circular dependencies between traits. This can happen if you have two traits that need each other.Self types can also handle ***structural types***—types that merely specify the methods that a class must have, without naming the class. Here is the LoggedException using a structural type:

**trait** LoggedException **extends** ConsoleLogger {  
 **this**: { **def** getMessage() : String } =>  
 **def** log() { log(getMessage()) }  
}

The trait can be mixed into any class that has a getMessage method.We discuss self types and structural types in more detail in Chapter 19.

**What happens under the Hood**

Scala needs to translate traits into clases and interfaces of the JVM. You are not required to know how is one, but you may find it helpful for understanding how traits work.

A trait that has only abstract memthods is simply turned into a Java interface.

|  |  |
| --- | --- |
| **trait** Logger {  **def** log(msg: String) } | **public** **interface** Logger {  **void** log(String msg); }*//Generated Java interface* |
| **class** SavingsAccount **extends** Account **with** ConsoleLogger **with** ShortLogger | **public** **class** SavingsAccount **extends** Account **implements** ConsoleLogger, ShortLogger {  **public** SavingsAccount() {  **super**();  ConsoleLogger.$init$();  ShortLogger.$init$()} } |
|  |  |

**Operators**

This chapter covers in detail implementing your own *operators*—methods with the same syntax as the familiar mathematical operators. Operators are often used to build *domain-specific languages*—minilanguages embedded inside Scala. *Implicit conversions* (type conversion functions that are applied automatically) are another tool facilitating the creation of domain-specific languages. This chapter also discusses the special methods apply, update, and unapply. We end the chapter with a discussion of *dynamic invocations*—method calls that can be intercepted at runtime, so that arbitrary actions can occur depending on the method names and arguments.

-Identifiers contain either alphanumeric or operator characters.

Unary and binary operators are method calls.

• Operator precedence depends on the first character, associativity on the last.

• The apply and update methods are called when evaluating expr(args).

• Extractors extract tuples or sequences of values from an input .

• Types extending the Dynamic trait can inspect the names of methods and arguments at runtime.

**Identifiers**

The name of variables, functions, classes and so on are collectively called ***identifiers***. In Scala, you have more choices for forming identifiers than in other languages. Of course, you can follow the time-honored pattern: sequences of alphanumeric characters, starting with an alphabetic carácter ora n underscore, such as input1 or next\_token.

As in Java, Unicode characters are allowed. For example, quantité or ποσó are valid identifiers, In addition, you can use operator characters in identifiers:

-The ASCII characters ! # % & \* + - / : < = > ? @ \ ^ | ~ that are not letters, digits, underscore, the .,; punctuation marks, parentheses () [] {}, or quotation marks ' ` ".

For example, \*\* and √ are valid identifiers. With the definition

val √ = scala.math.sqrt \_

-You can write √(2) to compute a square root. This may be a good idea, provided one’s programming environment makes it easy to type the symbol.

The identifiers @ # : = \_ => <- <: <% >: ⇒ ← are reserved in the specification, and you cannot redefine them.

If the carácter ‘=’ is reserved as a operator, it shoud be used as =(), shoudn’t it?

Finally, you can include just about any sequence of characters in backquotes. For example, val `val` = 42

That example is silly, but backquotes can sometimes be an “escape hatch.” For example, in Scala, yield is a reserved word, but you may need to access a Java method of the same name. Backquotes to the rescue: Thread.`yield`().

**Infix operators**

You can write a identifier b where identifier denotes a method with two parameters (one implicit, one explicit). For example, the expresión 1 to 10 is actually a method call 1.to(10). This is called an infix expression because the operator is between the arguments. The operator can contain letters, as in to, or it can contain operator characters—for example, 1 -> 10 is a method call 1 .->(10). To define an operator in your own class, simply define a method whose name is that of the desired operator. For example, here is a Fraction class that multiplies two fractions according to the law:

(n1 / d1) × (n2 / d2) = (n1n2 / d1d2)

**class** Fraction(n: Int, d: Int) {  
 **private val** *num* = ...  
 **private val** *den* = ...  
 ...  
 **def** \*(other: Fraction) = **new** Fraction(*num* \* other.*num*, *den* \* other.*den*)  
}

**Unary operators**

Infix operators are binary operators—they have two parameters. An operator with one parameter is called a ***unary operator***.

The four operators +, -, !, ~ are allowed as prefix operators, appearing before their arguments. They are converted into calls to methods with the name *unary\_operator*.

+a means the same as a.unary\_+ .

If a unary operator follows its argument, it is a **postfix** operator. The expression a identifier

is the same as the method call a.identifier() for instance 42 toString is the same as 42.toString()

Postfix operators can lead to parsing errors. For example, the code:

val result = 42 toString

println(result)

yields the error message “too many arguments for method toString”. Since parsing precedes type inference and overload resolution, the compiler does not yet know that toString is a unary method. Instead, the code is parsed as val result = 42.toString(println(result)).

For that reason, the compiler warns you if you use postfix operators.You can turn the warning off with the compiler option -language:postfixOps, or by adding the clause:

import scala.language.postfixOps

**Assignment operators**

An assignment operator has the form operator=, and the expresión a operator= b

means the same as a = a operator b. For example, a += b is equivalent to a = a + b.

There are a few technical details.

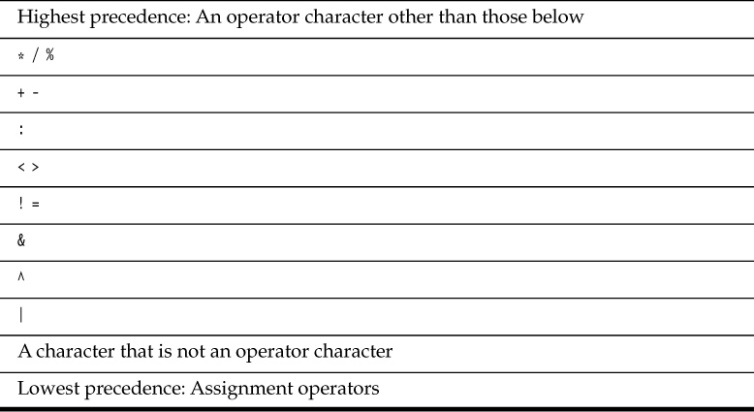
<=, >=, and != are not assignment operators.

An operator starting with an = is never an assignment operator (==, ===, =/=, and so on).

If a has a method called operator=, then that method is called directly.

**Precedence**

When you have two or more operators in a row without parentheses, the ones with higher precedence are executed first. For example, in the expresión 1 + 2 \* 3 the \* operator is evaluated first. Scala can have arbitrary operators instead of Java or C++ which it is fixed, so it uses a scheme that works for all operators, while also giving the familiar precedence order to the standard ones. Except for assignment operators, the precedence is determined by the first character of the operator.



Characters in the same row yield operators with the same precedence. For example, + and -> have the same precedence.

*Postfix* operators have lower precedence than *infix* operators:

a *infixOp* b*postfixOp* is the same as(a *infixOp* b)*postfixOp*

**Associativity**

When you have a sequence of operators of the same precedence, the associativity determines whether they are evaluated left-to-right or right-to-left. For example, in the expression *17 – 2 – 9*, one computes *(17 – 2) – 9*. The *–* operator is left-associative.

In Scala, all operators are left-associative except for

- operators that end in a colon (:)

- assignment operators

In particular, the :: operator for constructing lists is right-associative. For example,

*1 :: 2 :: Nil* means *1 :: (2 :: Nil)*

This is as it should be—we first need to form the list containing *2*, and that list becomes the tail of the list whose head is *1*.

A right-associative binary operator is a method of its second argument. For example,

*2 :: Nil* means *Nil.::(2)*

**The apply and update methods**

Scala lets you extend the function call syntax f(arg1, arg2, ...) to values other than functions. If f is not a function or method, then this expression is equivalent to the call f.apply(arg1, arg2, ...).

Unless it occurs to the left of an assignment. The expresión f(arg1, arg2, ...) = value

corresponds to the call f.update(arg1, arg2, ..., value)

This mechanism is used in arrays and maps. For example,

**val** scores = **new** scala.collection.mutable.HashMap[String, Int]  
scores(**"Bob"**) = 100 *// Calls scores.update("Bob", 100)***val** bobsScore = scores(**"Bob"**) *// Calls scores.apply("Bob")*

The apply method is also commonly used in companion objects to construct objects without calling new. For example, consider a Fraction class.

**class** Fraction(n: Int, d: Int) {  
 ...  
}  
**object** Fraction {  
 **def** apply(n: Int, d: Int) = **new** Fraction(n, d)  
}

Because of the apply method, we can construct a fraction as Fraction(3, 4) instead of new Fraction(3, 4). That sounds like a small thing, but if you have many Fraction values, it is a welcome improvement:

**val** result = Fraction(3, 4) \* Fraction(2, 5)

**Extractors**

An extractor is an object with an unapply method. You can think of the unapply method as the opposite of the apply method of a companion object. An apply method takes construction parameters and turns them into an object. An unapply method takes an object and extracts values from it—usually the values from which the object was constructed.

For example, suppose you want to extract first and last names from a string:

**val** author = **"Cay Horstmann"**

**val** Name(first, last) = author *// Calls Name.unapply(author)*

Provide an object Name with an unapply method that returns an Option[(String, String)]. If the match succeeds, return a pair with the first and last name. The components of the pair will be bound to the variables in the pattern. Otherwise, return None.

**object** Name {  
 **def** unapply(input: String) = {  
 **val** pos = input.indexOf(**" "**)  
 **if** (pos == -1) None  
 **else** *Some*((input.substring(0, pos), input.substring(pos + 1)))  
 }  
}

In this example, there is no Name class. The Name object is an extractor for String objects.

As shown in the preceding example, the extracors can be used to extract information from an object of any type, in that case from an String. Now, lets see how to use it in an example when the apply and unapply methods are inverses of one another.

Consider the Fraction class from the preceding section. The apply method makes a fraction from a numerator and denominator. An unapply method retrieves the numerator and denominator. You can use it in a variable definition

**var** Fraction(a, b) = Fraction(3, 4) \* Fraction(2, 5)  
*// a, b are initialized with the numerator and denominator of the result*

**case** Fraction(a, b) => ... *// a, b are bound to the numerator and denominator*

In general, a pattern match can fail. Therefore, the unapply method returns an Option. It contains a tuple with one value for each matched variable. In our case, we return an Option[(Int, Int)].

**object** Fraction {  
 **def** unapply(input: Fraction) =  
 **if** (input.den == 0) None **else** *Some*((input.num, input.den))  
}

Just to show the possibility, this method returns None when the denominator is zero, indicating no match.

**val** Fraction(a, b) = f;

becomes

**val** tupleOption = Fraction.unapply(f)  
**if** (tupleOption == None) **throw new** MatchError  
*// tupleOption is Some((t1, t2))***val** a = t1  
**val** b = t2

Note that in the declaration **val** Fraction(a, b) = f; neither the Fraction.apply method nor the Fraction constructor are called. Instead, the statement means: “Initialize a and b so that if they would be passed to Fraction.apply, the result would be f.”

Every case class automatically has apply and unapply methods. (Case classes are discussed in Chapter 14.) For example, consider:

**case class** Currency(value: Double, unit: String)

You can construct a Currency instance as

Currency(29.95, **"EUR"**) *// Calls Currency.apply*

You can extract values from a Currency object:

**case** Currency(amount, **"USD"**) => *println*(**s"$$$**amount**"**) *// Calls Currency.unapply*

**Extractors with one or no arguments**

In Scala, there are no tuples with one component. If the unapply method extracts a single value, it should just return an Option of the target type. For example,

**object** Number {  
 **def** unapply(input: String): Option[Int] =  
 **try** {  
 *Some*(input.trim.toInt)  
 } **catch** {  
 **case** ex: NumberFormatException => None  
 }  
}

With this extractor, you can extract a number from a string:

**val** Number(n) = **"1729"**

An extractor can just test its input without extracting any value. In that case, the unapply method should return a Boolean. For example,

**object** IsCompound {  
 **def** unapply(input: String) = input.contains(**" "**)  
}

You can use this extractor to add a test to a pattern, for example

author **match** {  
 **case** *Name*(first, **IsCompound()**) => ...  
 *// Matches if the last name is compound, such as van der Linden* **case** *Name*(first, last) => ...  
}

**The unapplyseq method**

To extracta n arbitrary sequence of values, the method needs to be called unapplySeq. It returns an Option[Seq[A]], where A is the type of the extracted values. For example, a Name extractor can produce a sequence of the names’s components:

**object** Name {  
 **def** unapplySeq(input: String): Option[Seq[String]] =  
 **if** (input.trim == **""**) None **else** *Some*(input.trim.split(**"\\s+"**))  
}

Now you can match for any number of variables

author **match** {  
 **case** Name(first, last) => ...  
 **case** Name(first, middle, last) => ...  
 **case** Name(first, **"van"**, **"der"**, last) => ...  
 ...  
}

Do not supply both an unapply and an unapplySeq methods with the same argument types.

**Dynamic invocation**

Scala is a strongly typed language that reports type errors at compile time rathere than at runtime. If you have an expresión x.f(args), and your program compiles, then you know for sure that x has a method f that can accept the given arguments. However, there are situations where it is desirable to define methods in a running program. This is common with object-relational mappers in Dynamic languages such as Ruby or JavaScript. Objects that represent database tables have methods findByName, findById, and so on, with the method names matching the table columns. For database entities, the columna names can be used to get and set fields, such as person.lastName = “Doe”. In Scala, you can do this too. If a type extends the trait scala.Dynamic, then method calls, getters, and setters are rewritten as calls to special methods that can inspect the name of the original call and the parameters, and then take arbitrary actions. Dynamic types are an “exotic” feature, and the compiler wants your explicit consent when you implement such a type. You do that by adding the import statement import scala.language.dynamics

Users of such types do not need to provide the import statement.

Here are the details of the rewriting. Consider obj.name, where obj belongs to a class that’s a subtype of Dynamic. Here is what the Scala compiler does with it.

**1.** If name is a known method or field of obj, it is processed in the usual way.

**2.** If obj.name is followed by (arg1, arg2, ...),

**a.** If none of the arguments are named (of the form *name*=*arg*), pass the arguments on to applyDynamic: obj.applyDynamic(**"name"**)(arg1, arg2, ...)

**b.** If at least one of the arguments is named, pass the name/value pairs on to applyDynamicNamed:

obj.applyDynamicNamed(**"name"**)((name1, arg1), (name2, arg2), ...)

Here, name1, name2, and so on are strings with the argument names, or "" for unnamed arguments.

**3.** If obj.name is to the left of an =, call obj.updateDynamic(**"name"**)(rightHandSide)

**4.** Otherwise call obj.selectDynamic(**"sel"**)

The calls to updateDynamic, applyDynamic, and applyDynamicNamed have two sets of parentheses, one for the selector name and one for the arguments. This construct is explained in Chapter 12.

Let’s look at a few examples. Suppose person is an instance of a type extending Dynamic.

The stat statement person.lastName = **"Doe"** is replaced with a call:

person.updateDynamic(**"lastName"**)(**"Doe"**)

The Person class must have such a method:

**class** Person {  
 ...  
 **def** updateDynamic(field: String)(newValue: String) { ... }  
}

It is then up to you to implement the updateDynamic method. For example, if you are implementing an object-relational mapper, you might update the cached entity and mark it as changed, so that it can be persisted in the database.

Conversely, a statement **val** name = person.lastName turns into:

**val** name = name.selectDynamic(**"lastName"**)

The selectDynamic method would simply look up the field value.

Method calls that don’t involve named parameters are translated to applyDynamic calls.

For example, **val** does = people.findByLastName(**"Doe"**) becomes:

**val** does = people.applyDynamic(**"findByLastName"**)(**"Doe"**)

and **val** johnDoes = people.find(lastName = **"Doe"**, firstName = **"John"**)

becomes **val** johnDoes = people.applyDynamicNamed(**"find"**)  ((**"lastName"**, **"Doe"**), (**"firstName"**, **"John"**))

It is then up to you to implement applyDynamic and applyDynamicNamed as calls that retrieve the matching objects.

Here is a concrete example. Suppose we want to be able to dynamically look up and set elements of a java.util.Properties instance, using the dot notation:

**val** sysProps = **new** DynamicProps(System.getProperties)  
sysProps.username = **"Fred"** *// Sets the "username" property to "Fred"***val** home = sysProps.java\_home *// Gets the "java.home" property*

The DynamicProps class extends the Dynamic trait and implements the updateDynamic and selectDynamic methods:

**class** DynamicProps(**val** props: java.util.Properties) **extends** Dynamic {  
 **def** updateDynamic(name: String)(value: String) {  
 props.setProperty(name.replaceAll(**"\_"**, **"."**), value)  
 }  
 **def** selectDynamic(name: String) =  
 props.getProperty(name.replaceAll(**"\_"**, **"."**))  
}

As an additional enhancement, let us use the add method to add key/value pairs in bulk, using named arguments: sysProps.add(username=**"Fred"**, password=**"Secret"**)

Then we need to supply the applyDynamicNamed method in the DynamicProps class. Note that the name of the method is fixed. We are only interested in arbitrary parameter names.

**class** DynamicProps(**val** props: java.util.Properties) **extends** Dynamic {  
 ...  
 **def** applyDynamicNamed(name: String)(args: (String, String)\*) {  
 **if** (name != **"add"**) **throw new** IllegalArgumentException  
 **for** ((k, v) <- args)  
 props.setProperty(k.replaceAll(**"\_"**, **"."**), v)  
 }  
}

These examples are only meant to illustrate the mechanism—I don’t think that it is a good idea to use the dot notation for map access. Like operator overloading, dynamic invocation is a feature that is best used with restraint.

**Higher-Order functions**

Scala mixes object orientation with functional features. In a functional programming language, functions are first-class citizens that can be passed around and manipulated just like any other data types. This is very useful whenever you want to pass some action detail to an algorithm. In a functional language, you just wrap that detail into a function that you pass as a parameter. In this chapter, you will see how to be productive with functions that use or return functions.

- Functions are “first-class citizens” in Scala, just like numbers.

- You can create anonymous functions, usually to give them to other functions.

- A function argument specifies behavior that should be executed later.

- Many collection methods take function parameters, applying a function to the values of the collection.

- There are syntax shortcuts that allow you to express function parameters in a way that is short and easy to read.

-You can create functions that operate on blocks of code and look much like the built-in control statements.

**Functions as values**

In Scala, a function is a first-class citizen, just like a number. You can store a function in a variable:

**import** scala.math.\_  
  
**val** num = 3.14  
  
**val** fun = ceil \_

This code sets num to 3.14 and fun to the ceil function.The \_ behind the ceil function indicates that you really meant the function, and you didn’t just forget to supply the arguments.When you try this code in the REPL, the type of num is, not surprisingly, Double. The type of fun is reported as (Double) => Double—that is, a function receiving and returning a Double.

Technically, the \_ turns the ceil method into a function. In Scala, you cannot manipulate methods, only functions. The type of the function is (Double) => Double, with an arrow. In contrast, the type of the ceil method is (Double)Double, without an arrow. There is no way for you to work with such a type, but you will find it in compiler and REPL messages.

The \_ suffix is not necessary when you use a method name in a context where a function is expected. For example, the following is legal:

**val** f: (Double) => Double = ceil *// No underscore needed*

The ceil method is a method of the scala.math package object. If you have a method from a class, the syntax for turning it into a function is slightly different:

**val** f = (\_: String).charAt(\_: Int)  *// A function(String, Int) => Char*

Alternatively, you can specify the type of the function instead of the parameter types:

**val** f: (String, Int) => Char = \_.charAt(\_)

What can you do with a function? Two things:

- Call it.

- Pass it around, by storing it in a variable or giving it to a function as a parameter.

Here is how to call the function stored in fun: fun(num) *// 4.0*

As you can see, the normal function call syntax is used. The only difference is that fun is a ***variable containing a function***, not a fixed function.Here is how you can give fun to another function: Array(3.14, 1.42, 2.0).map(fun) *// Array(4.0, 2.0, 2.0)*

**Anonymous functions**

In Scala, you don’t have to give a name to each function, just like you don’t have to give a name to each number. Here is an ***anonymous function***: (x: Double) => 3 \* x

This function multiplies its argument by 3.

Of course, you can store this function in a variable: **val** triple = (x: Double) => 3 \* x

That’s just as if you had used a def: **def** triple(x: Double) = 3 \* x

But you don’t have to name the function. You can just pass it to another function:

Array(3.14, 1.42, 2.0).map(**(x: Double) => 3 \* x**)  
*// Array(9.42, 4.26, 6.0)*

Here, we tell the map method: “Multiply each element by 3.”

If you prefer, you can enclose the function argument in braces instead of parentheses, for example: Array(3.14, 1.42, 2.0).map{ (x: Double) => 3 \* x }

This is more common when a method is used in infix notation (without the dot).

Array(3.14, 1.42, 2.0) map { (x: Double) => 3 \* x }

Anything defined with def (in the REPL or a class or object) is a method, not a function:

scala> **def** triple(x: Double) = 3 \* xtriple: (x: Double)Double

triple: (x: Double)Double

Note the method type (x: Double)Double.

In contrast, a function definition has a function type:

triple: Double => Double

**Functions with function parameters**

In this section, you will see how to implement a function that takes another function as a parameter. Here is an example:

**def** valueAtOneQuarter(f: (Double) => Double) = f(0.25)

Note that the parameter can be *any* function receiving and returning a Double. The valueAtOneQuarter function computes the value of that function at 0.25.

valueAtOneQuarter(ceil \_) *// 1.0*valueAtOneQuarter(sqrt \_) *// 0.5 (because 0.5 × 0.5 = 0.25)*

What is the type of valueAtOneQuarter? It is a function with one parameter, so its type is written as(parameterType) => resultTypeThe resultType is clearly Double, and the parameterType is already given in the function header as (Double) => Double. Therefore, the type of valueAtOneQuarter is ((Double) => Double) => Double

Since valueAtOneQuarter is a function that ***receives a function***, it is called a ***higher-order function****.* A higher-order function can also ***produce a function***. Here is a simple example:

def mulBy(factor : Double) = **(x : Double) => factor \* x**

The power of mulBy is that it can deliver functions that multiply by any amount:

**val** quintuple = mulBy(5)  
  
quintuple(20) *// 100*

The mulBy function has a parameter of type Double, and it returns a function of type (Double) => Double. Therefore, its type is:

(Double) => ((Double) => Double)

**Parameter inference**

When you pass an anonymous function to another function or method, Scala helps you out by deducing types when possible. For example, you don’t have to write

valueAtOneQuarter((x: **Double**) => 3 \* x) *// 0.75*

Since the valueAtOneQuarter method knows that you will pass in a (Double) => Double function, you can just write valueAtOneQuarter((x) => 3 \* x)

It gets better. If a parameter occurs only once on the right-hand side of the =>, you can replace it with an underscore: valueAtOneQuarter(3 \* \_). This is the ultimate in comfort, and it is also pretty easy to read: a function that multiplies something by 3, but keep in mind that these shortcuts only work when the parameter types are known.

**val** fun = 3 \* \_ *// Error: Can't infer types***val** fun = 3 \* (\_: Double) *// OK***val** fun: (Double) => Double = 3 \* \_*// OK because we specified the type for fun*

Of course, the last definition is contrived. But it shows what happens when a function is passed to a parameter (which has just such a type). Specifying the type of \_ is useful for turning methods into functions. For example, (\_: String).length is a function String => Int, and (\_: String).substring(\_:Int, \_: Int) is a function

(String, Int, Int) => String.