Classes and Objects

类的定义，基本概念。

Classes, fields, and methods

A class is a blueprint for objects. Once you define a class, you can create objects from the class blueprint with the keyword new. For example, given the class definition:

class ChecksumAccumulator {

// class definition goes here

}

You can create ChecksumAccumulator objects with:

new ChecksumAccumulator

Inside a class definition, you place fields and methods, which are collectively called **members**. **Fields**, which you define with either val or var, are variables that refer to objects. **Methods**, which you define with def, contain executable code. The fields hold the state, or data, of an object, whereas the methods use that data to do the computational work of the object. When you instantiate a class, the runtime sets aside some memory to hold the image of that object’s state--i.e., the content of its variables.

For example, if you defined a ChecksumAccumulator class and gave it a var field named sum:

class ChecksumAccumulator {

var sum = 0

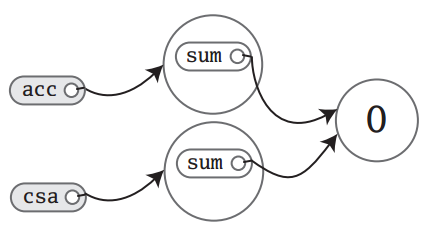
}

and you instantiated it twice with:

val acc = new ChecksumAccumulator

val csa = new ChecksumAccumulator

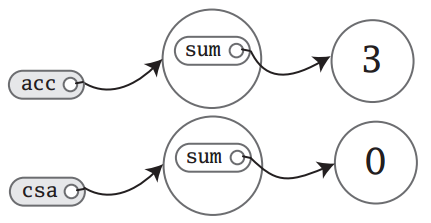
The image of the objects in memory might look like:



you can later reassign to sum a different Int value, like this:

acc.sum = 3

Now the picture would look like:



**private fields** can only be accessed by methods defined in the same class. To declare a field private, you place a private access modifier in front of the field, like this:

class ChecksumAccumulator {

private var sum = 0

}

The way you make members public in Scala is by not explicitly specifying any access modifier.

以下是函数的知识，和类没有直接关系。本来是说明类中方法和变量的定义，掺杂了太多不必要的函数知识。

One important characteristic of method parameters in Scala is that they are vals, not vars. If you attempt to reassign a parameter inside a method in Scala, therefore, it won’t compile. In the absence of any explicit return statement, a Scala method returns the last value computed by the method.

class ChecksumAccumulator {

private var sum = 0

def add(b: Byte): Unit = {

sum += b

}

def checksum(): Int = {

return ~(sum & 0xFF) + 1

}

}

If the result expression is short, it can even be placed on the same line as the def itself. With these changes, class ChecksumAccumulator looks like this:

class ChecksumAccumulator {

private var sum = 0

def add(b: Byte): Unit = sum += b

def checksum(): Int = ~(sum & 0xFF) + 1

}

In add’s case, for example, the side effect is that sum is reassigned. Another way to express such methods is to leave off the result type and the equals sign, and enclose the body of the method in curly braces. In this form, the method looks like a procedure, a method that is executed only for its side effects. The add method in Listing 4.1 illustrates this style:

class ChecksumAccumulator {

private var sum = 0

def add(b: Byte) { sum += b }

def checksum(): Int = ~(sum & 0xFF) + 1

}

Listing 4.1 · Final version of class ChecksumAccumulator.

One puzzler to watch out for is that whenever you leave off the equals sign before the body of a function, its result type will definitely be Unit. This is true no matter what the body contains, because the Scala compiler can convert any type to Unit. For example, if the last result of a method is a String, but the method’s result type is declared to be Unit, the String will be converted to Unit and its value lost. Here’s an example:

scala> def f(): Unit = "this String gets lost"

f: ()Unit

In this example, the String is converted to Unit because Unit is the declared result type of function f. The Scala compiler treats a function defined in the procedure style, *i.e.*, with curly braces but no equals sign, essentially the same as a function that explicitly declares its result type to be Unit:

scala> def g() { "this String gets lost too" }

g: ()Unit

Semicolon inference

In a Scala program, a semicolon at the end of a statement is usually optional. You can type one if you want but you don’t have to if the statement appears by itself on a single line. On the other hand, a semicolon is required if you write multiple statements on a single line:

val s = "hello"; println(s)

The rules of semicolon inference. In short, a line ending is treated as a semicolon unless one of the following conditions is true:

1. The line in question ends in a word that would not be legal as the end of a statement, such as a period or an infix operator.

2. The next line begins with a word that cannot start a statement.

3. The line ends while inside parentheses (...) or brackets [...], because these cannot contain multiple statements anyway.

Singleton objects

这一章节阐述的不知所云，替换掉。

A singleton object definition looks like a class definition, except instead of the keyword class you use the keyword object. When a singleton object shares the same name with a class, it is called that class’s **companion object**. You must define both the class and its companion object in the same source file. The class is called the **companion class** of the **singleton object**. A class and its companion object can access each other’s private members.

[Scala单例对象、伴生对象实战详解](http://www.cnblogs.com/nethk/p/5609320.html)

<http://www.cnblogs.com/nethk/p/5609320.html>

在Scala中，没有静态成员、静态方法这一说，但是object可以做到类似效果：

object University{

private var studentNo = 0

def newStudenNo = {

studentNo += 1

studentNo

}

}

object ObjecOps {

def main(args: Array[String]): Unit = {

println(University.newStudenNo) //第一次调用时，初始化成员变量

println(University.newStudenNo)

}

}

//结果：1、2

class University{

val id = University.newStudenNo //可以访问伴生对象的任何成员，即使是private

private var number =0

def aClass(number:Int){this.number += number}

}

* standalone object

A singleton object that does not share the same name with a companion class is called a standalone object. You can use standalone objects for many purposes, including collecting related utility methods together or defining an entry point to a Scala application. This use case is shown in the next section.

A Scala application

To run a Scala program, you must supply the name of a standalone singleton object with a main method that takes one parameter, an Array[String], and has a result type of Unit. Any standalone object with a main method of the proper signature can be used as the entry point into an application. An example is shown in Listing 4.3:

// In file Summer.scala

import ChecksumAccumulator.calculate

object Summer {

def main(args: Array[String]) {

for (arg <- args)

println(arg +": "+ calculate(arg))

}

}

Listing 4.3 · The Summer application.

Scala implicitly imports members of packages java.lang and scala, as well as the members of a singleton object named Predef, into every Scala source file. Predef, which resides in package scala, contains many useful methods. For example, when you say println in a Scala source file, you're actually invoking println on Predef. (Predef.println turns around and invokes Console.println, which does the real work.) When you say assert, you're invoking Predef.assert.

To run the Summer application, place the code from Listing 4.3 into a file named Summer.scala. Because Summer uses ChecksumAccumulator, place the code for ChecksumAccumulator, both the class shown in Listing 4.1 and its companion object shown in Listing 4.2, into a file named ChecksumAccumulator.scala.

you’ll need to actually compile these files with the Scala compiler, then run the resulting class files. One way to do this is to use scalac, which is the basic Scala compiler, like this:

$ scalac ChecksumAccumulator.scala Summer.scala

the Scala distribution also includes a Scala compiler daemon called fsc (for fast Scala compiler). You use it like this:

$ fsc ChecksumAccumulator.scala Summer.scala

You can run the Summer application, therefore, by typing:

$ scala Summer of love

You will see checksums printed for the two command line arguments:

of: -213

love: -182

The Application trait

Scala provides a trait, scala.Application, that can save you some finger typing. Listing 4.4 shows an example:

import ChecksumAccumulator.calculate

object FallWinterSpringSummer extends Application {

for (season <- List("fall", "winter", "spring"))

println(season +": "+ calculate(season))

}

Listing 4.4 · Using the Application trait.

To use the trait, you first write “extends Application” after the name of your singleton object. Then instead of writing a main method, you place the code you would have put in the main method directly between the curly braces of the singleton object. That’s it. You can compile and run this application just like any other.

Basic Types and Operations

Some basic types

Collectively, types Byte, Short, Int, Long, and Char are called integral types. The integral types plus Float and Double are called numeric types.

Table 5.1 · Some basic types

**Value type Range**

Byte 8-bit signed two’s complement integer (-27 to 27 - 1, inclusive)

Short 16-bit signed two’s complement integer (-215 to 215 - 1, inclusive)

Int 32-bit signed two’s complement integer (-231 to 231 - 1, inclusive)

Long 64-bit signed two’s complement integer (-263 to 263 - 1, inclusive)

Char 16-bit unsigned Unicode character (0 to 216 - 1, inclusive)

String a sequence of Chars

Float 32-bit IEEE 754 single-precision float

Double 64-bit IEEE 754 double-precision float

Boolean true or false

Functional Objects

A specification for class Rational

This chapter will give you a glimpse of some of the ways Scala enables you to write libraries that feel like native language support. For example, at the end of this chapter you’ll be able to do this with class Rational:

scala> val oneHalf = new Rational(1, 2)

oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)

twoThirds: Rational = 2/3

scala> (oneHalf / 7) + (1 - twoThirds)

res0: Rational = 17/42

Constructing a Rational

The Scala compiler will compile any code you place in the class body, which isn’t part of a field or a method definition, into the **primary constructor**. For example, you could print a debug message like this:

class Rational(n: Int, d: Int) {

println("Created "+ n +"/"+ d)

}

Given this code, the Scala compiler would place the call to println into Rational’s primary constructor. The println call will, therefore, print its debug message whenever you create a new Rational instance:

scala> new Rational(1, 2)

Created 1/2

res0: Rational = Rational@90110a

Reimplementing the toString method

When we created an instance of Rational in the previous example, the interpreter printed “Rational@90110a”. The interpreter obtained this somewhat funny looking string by calling toString on the Rational object. By default, class Rational inherits the implementation of toString defined in class java.lang.Object, which just prints the class name, an @ sign, and a hexadecimal number. You can *override* the default implementation by adding a method toString to class Rational, like this:

class Rational(n: Int, d: Int) {

override def toString = n +"/"+ d

}

You can test the new behavior of Rational in the interpreter:

scala> val x = new Rational(1, 3)

x: Rational = 1/3

scala> val y = new Rational(5, 7)

y: Rational = 5/7

Checking preconditions

A precondition is a constraint on values passed into a method or constructor, a requirement which callers must fulfill. One way to do that is to use require, like this:

class Rational(n: Int, d: Int) {

require(d != 0)

override def toString = n +"/"+ d

}

The require method takes one boolean parameter. If the passed value is true, require will return normally. Otherwise, require will prevent the object from being constructed by throwing an IllegalArgumentException.

Adding fields

class Rational(n: Int, d: Int) {

require(d != 0)

val numer: Int = n

val denom: Int = d

override def toString = numer +"/"+ denom

def add(that: Rational): Rational =

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

scala> val oneHalf = new Rational(1, 2)

oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)

twoThirds: Rational = 2/3

scala> oneHalf add twoThirds

res3: Rational = 7/6

scala> val r = new Rational(1, 2)

r: Rational = 1/2

scala> r.numer

res4: Int = 1

scala> r.denom

res5: Int = 2

Self references

The keyword this refers to the object instance on which the currently executing method was invoked, or if used in a constructor, the object instance being constructed. As an example, consider adding a method, lessThan, which tests whether the given Rational is smaller than a parameter:

def lessThan(that: Rational) = this.numer \* that.denom < that.numer \* this.denom

Here, this.numer refers to the numerator of the object on which lessThan was invoked. You can also leave off the this prefix and write just numer; the two notations are equivalent.

Auxiliary constructors

In Scala, constructors other than the primary constructor are called auxiliary constructors. Auxiliary constructors in Scala start with def this(...). The body of Rational’s auxiliary constructor merely invokes the primary constructor, passing along its lone argument, n, as the numerator and 1 as the denominator.

class Rational(n: Int, d: Int) {

require(d != 0)

val numer: Int = n

val denom: Int = d

def this(n: Int) = this(n, 1) // auxiliary constructor

override def toString = numer +"/"+ denom

def add(that: Rational): Rational =

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

In Scala, every auxiliary constructor must invoke another constructor of the same class as its first action. In other words, the first statement in every auxiliary constructor in every Scala class will have the form “this(. . . )”. The invoked constructor is either the primary constructor (as in the Rational example), or another auxiliary constructor that comes textually before the calling constructor. The net effect of this rule is that every constructor invocation in Scala will end up eventually calling the primary constructor of the class. The primary constructor is thus the single point of entry of a class.

Private fields and methods

class Rational(n: Int, d: Int) {

require(d != 0)

private val g = gcd(n.abs, d.abs)

val numer = n / g

val denom = d / g

def this(n: Int) = this(n, 1)

def add(that: Rational): Rational =

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

override def toString = numer +"/"+ denom

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

}

scala> new Rational(66, 42)

res7: Rational = 11/7

Defining operators

class Rational(n: Int, d: Int) {

require(d != 0)

private val g = gcd(n.abs, d.abs)

val numer = n / g

val denom = d / g

def this(n: Int) = this(n, 1)

def + (that: Rational): Rational =

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

def \* (that: Rational): Rational =

new Rational(numer \* that.numer, denom \* that.denom)

override def toString = numer +"/"+ denom

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

}

With class Rational defined in this manner, you can now write:

scala> val x = new Rational(1, 2)

x: Rational = 1/2

scala> val y = new Rational(2, 3)

y: Rational = 2/3

scala> x + y

res8: Rational = 7/6

scala> x.+(y)

res9: Rational = 7/6

Built-in Control Structures

One thing you will notice is that almost all of Scala’s control structures result in some value. This is the approach taken by functional languages, in which programs are viewed as computing a value, thus the components of a program should also compute values.

If expressions

val filename =

if (!args.isEmpty) args(0)

else "default.txt"

While loops

def gcdLoop(x: Long, y: Long): Long = {

var a = x

var b = y

while (a != 0) {

val temp = a

a = b % a

b = temp

}

b

}

Listing 7.2 · Calculating greatest common divisor with a while loop.

Scala also has a do-while loop.

var line = ""

do {

line = readLine()

println("Read: "+ line)

} while (line != "")

Listing 7.3 · Reading from the standard input with do-while.

The while and do-while constructs are called “loops,” not expressions, because they don’t result in an interesting value. The type of the result is Unit. It is called the unit value and is written (). Try this in the interpreter:

scala> def greet() { println("hi") }

greet: ()Unit

scala> greet() == ()

hi

res0: Boolean = true

Because no equals sign precedes its body, greet is defined to be a procedure with a result type of Unit. Therefore, greet returns the unit value, ().

One other construct that results in the unit value, which is relevant here, is reassignment to vars.

var line = ""

while ((line = readLine()) != "") // This doesn’t work!

println("Read: "+ line)

When you compile this code, Scala will give you a warning that comparing values of type Unit and String using != will always yield true. in Scala assignment always results in the unit value, ().

For expressions

Iteration through collections

val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere)

println(file)

Listing 7.5 · Listing files in a directory with a for expression.

scala> for (i <- 1 to 4)

println("Iteration "+ i)

Iteration 1

Iteration 2

Iteration 3

Iteration 4

scala> for (i <- 1 until 4)

println("Iteration "+ i)

Iteration 1

Iteration 2

Iteration 3

Sometimes you do not want to iterate through a collection in its entirety. You want to filter it down to some subset. You can do this with a for expression by adding a filter: an if clause inside the for’s parentheses.

val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere if file.getName.endsWith(".scala"))

println(file)

Listing 7.6 · Finding .scala files using a for with a filter.

You could alternatively accomplish the same goal with this code:

for (file <- filesHere)

if (file.getName.endsWith(".scala"))

println(file)

You can include more filters if you want. Just keep adding if clauses.

for (

file <- filesHere

if file.isFile

if file.getName.endsWith(".scala")

) println(file)

Listing 7.7 · Using multiple filters in a for expression.

Nested iteration

If you add multiple <- clauses, you will get nested “loops.”

def fileLines(file: java.io.File) =

scala.io.Source.fromFile(file).getLines().toList

def grep(pattern: String) =

for (

file <- filesHere

if file.getName.endsWith(".scala");

line <- fileLines(file)

if line.trim.matches(pattern)

) println(file +": "+ line.trim)

grep(".\*gcd.\*")

Listing 7.8 · Using multiple generators in a for expression.

Mid-stream variable bindings

Note that the previous code repeats the expression line.trim. This is a non-trivial computation, so you might want to only compute it once. You can do this by binding the result to a new variable using an equals sign (=). The bound variable is introduced and used just like a val, only with the val keyword left out. Listing 7.9 shows an example.

In Listing 7.9, a variable named trimmed is introduced halfway through the for expression. That variable is initialized to the result of line.trim. The rest of the for expression then uses the new variable in two places, once in an if and once in println.

def grep(pattern: String) =

for {

file <- filesHere

if file.getName.endsWith(".scala")

line <- fileLines(file)

trimmed = line.trim

if trimmed.matches(pattern)

} println(file +": "+ trimmed)

grep(".\*gcd.\*")

Listing 7.9 · Mid-stream assignment in a for expression.

Producing a new collection

For example, here is a function that identifies the .scala files and stores them in an array:

def scalaFiles =

for {

file <- filesHere

if file.getName.endsWith(".scala")

} yield file

Each time the body of the for expression executes it produces one value, in this case simply file. When the for expression completes, the result will include all of the yielded values contained in a single collection. The type of the resulting collection is based on the kind of collections processed in the iteration clauses. In this case the result is an Array[File], because filesHere is an array and the type of the yielded expression is File.

The syntax of a for-yield expression is like this:

for clauses yield body

For example, the for expression shown in Listing 7.10 first transforms the Array[File] named filesHere, which contains all files in the current directory, to one that contains only .scala files. For each of these it generates an Iterator[String] (the result of the fileLines method, whose definition is shown in Listing 7.8). An Iterator offers methods next and hasNext that allow you to iterate over a collection of elements. This initial iterator is transformed into another Iterator[String] containing only trimmed lines that include the substring "for". Finally, for each of these, an integer length is yielded. The result of this for expression is an Array[Int] containing those lengths.

val forLineLengths =

for {

file <- filesHere

if file.getName.endsWith(".scala")

line <- fileLines(file)

trimmed = line.trim

if trimmed.matches(".\*for.\*")

} yield trimmed.length

Listing 7.10 · Transforming an Array[File] to Array[Int] with a for.

Exception handling with try expressions

You create an exception object and then you throw it with the throw keyword:

throw new IllegalArgumentException

in Scala, throw is an expression that has a result type.

val half =

if (n % 2 == 0)

n / 2

else

throw new RuntimeException("n must be even")

Technically, an exception throw has type Nothing.

You catch exceptions using the syntax shown in Listing 7.11 The syntax for catch clauses was chosen for its consistency with an important part of Scala: pattern matching.

import java.io.FileReader

import java.io.FileNotFoundException

import java.io.IOException

try {

val f = new FileReader("input.txt")

// Use and close file

} catch {

case ex: FileNotFoundException => // Handle missing file

case ex: IOException => // Handle other I/O error

}

Listing 7.11 · A try-catch clause in Scala.

The body is executed, and if it throws an exception, each catch clause is tried in turn. In this example, if the exception is of type FileNotFoundException, the first clause will execute. If it is of type IOException, the second clause will execute. If the exception is of neither type, the try-catch will terminate and the exception will propagate further.

You can wrap an expression with a finally clause if you want to cause some code to execute no matter how the expression terminates. For example, you might want to be sure an open file gets closed even if a method exits by throwing an exception. Listing 7.12 shows an example.

import java.io.FileReader

val file = new FileReader("input.txt")

try {

// Use the file

} finally {

file.close() // Be sure to close the file

}

Listing 7.12 · A try-finally clause in Scala.

If you’re familiar with Java, it’s worth noting that Scala’s behavior differs from Java only because Java’s try-finally does not result in a value. As in Java, if a finally clause includes an explicit return statement, or throws an exception, that return value or exception will “overrule” any previous one that originated in the try block or one of its catch clauses. For example, given this, rather contrived, function definition:

def f(): Int = try { return 1 } finally { return 2 }

calling f() results in 2. By contrast, given:

def g(): Int = try { 1 } finally { 2 }

calling g() results in 1.

Match expressions

val firstArg = if (!args.isEmpty) args(0) else ""

val friend =

firstArg match {

case "salt" => "pepper"

case "chips" => "salsa"

case "eggs" => "bacon"

case \_ => "huh?"

}

println(friend)

Listing 7.15 · A match expression that yields a value.

Functions and Closures

Methods

import scala.io.Source

object LongLines {

def processFile(filename: String, width: Int) {

val source = Source.fromFile(filename)

for (line <- source.getLines())

processLine(filename, width, line)

}

private def processLine(filename: String, width: Int, line: String) {

if (line.length > width)

println(filename +": "+ line.trim)

}

}

object FindLongLines {

def main(args: Array[String]) {

val width = args(0).toInt

for (arg <- args.drop(1))

LongLines.processFile(arg, width)

}

}

$ scala FindLongLines 45 LongLines.scala

LongLines.scala: def processFile(filename: String, width: Int) {

Local functions

you can define functions inside other functions. Just like local variables, such local functions are visible only in their enclosing block.

import scala.io.Source

object LongLines {

def processFile(filename: String, width: Int) {

def processLine(line: String) {

if (line.length > width)

println(filename +": "+ line)

}

val source = Source.fromFile(filename)

for (line <- source.getLines())

processLine(line)

}

}

Listing 8.2 · LongLines with a local processLine function.

First-class functions

Scala has first-class functions. Not only can you define functions and call them, but you can write down functions as **unnamed literals** and then pass them around as values.

A function literal is compiled into a class that when instantiated at runtime is a function value. Thus the distinction between function literals and values is that function literals exist in the source code, whereas function values exist as objects at runtime. The distinction is much like that between classes (source code) and objects (runtime). Here is a simple example of a function literal that adds one to a number:

(x: Int) => x + 1

The => designates that this function converts the thing on the left (any integer x) to the thing on the right (x + 1). So, this is a function mapping any integer x to x + 1.

scala> var increase = (x: Int) => x + 1

increase: (Int) => Int = <function1>

scala> increase(10)

res0: Int = 11

scala> increase = (x: Int) => {

println("We")

println("are")

println("here!")

x + 1

}

increase: (Int) => Int = <function1>

scala> increase(10)

We

Are

here!

res2: Int = 11

scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x: Int) => println(x))

-11

-10

-5

0 5

10

scala> someNumbers.filter((x: Int) => x > 0)

res4: List[Int] = List(5, 10)

One way to make a function literal more brief is to leave off the parameter types. Thus, the previous example with filter could be written like this:

scala> someNumbers.filter((x) => x > 0)

res5: List[Int] = List(5, 10)

A second way to remove useless characters is to leave out parentheses around a parameter whose type is inferred. In the previous example, the parentheses around x are unnecessary:

scala> someNumbers.filter(x => x > 0)

res6: List[Int] = List(5, 10)

Placeholder syntax

To make a function literal even more concise, you can use underscores as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal. For example, \_ > 0 is very short notation for a function that checks whether a value is greater than zero:

scala> someNumbers.filter(\_ > 0)

res7: List[Int] = List(5, 10)

Sometimes when you use underscores as placeholders for parameters, the compiler might not have enough information to infer missing parameter types. For example, suppose you write \_ + \_ by itself:

scala> val f = \_ + \_

<console>:4: error: missing parameter type for expanded

function ((x$1, x$2) => x$1.$plus(x$2))

val f = \_ + \_

ˆ

In such cases, you can specify the types using a colon, like this:

scala> val f = (\_: Int) + (\_: Int)

f: (Int, Int) => Int = <function2>

scala> f(5, 10)

res9: Int = 15

Note that \_ + \_ expands into a literal for a function that takes two parameters. This is why you can use this short form only if each parameter appears in the function literal at most once. Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly. The first underscore represents the first parameter, the second underscore the second parameter, the third underscore the third parameter, and so on.

Partially applied functions

Although the previous examples substitute underscores in place of individual parameters, you can also replace an entire parameter list with an underscore. For example, rather than writing println(\_), you could write println \_. Here’s an example:

someNumbers.foreach(println \_)

Scala treats this short form exactly as if you had written the following:

someNumbers.foreach(x => println(x))

Thus, the underscore in this case is not a placeholder for a single parameter. It is a placeholder for an entire parameter list.

In Scala, when you invoke a function, passing in any needed arguments, you apply that function to the arguments. For example, given the following function:

scala> def sum(a: Int, b: Int, c: Int) = a + b + c

sum: (a: Int,b: Int,c: Int)Int

You could apply the function sum to the arguments 1, 2, and 3 like this:

scala> sum(1, 2, 3)

res10: Int = 6

A **partially applied function** is an expression in which you don’t supply all of the arguments needed by the function. Instead, you supply some, or none, of the needed arguments. For example, to create a partially applied function expression involving sum, in which you supply none of the three required arguments, you just place an underscore after “sum”. The resulting function can then be stored in a variable. Here’s an example:

scala> val a = sum \_

a: (Int, Int, Int) => Int = <function3>

scala> a(1, 2, 3)

res11: Int = 6

scala> val b = sum(1, \_: Int, 3)

b: (Int) => Int = <function1>

scala> b(2)

res13: Int = 6 // In this case, b.apply invoked sum(1, 2, 3).

scala> b(5)

res14: Int = 9 // And in this case, b.apply invoked sum(1, 5, 3).

Closures

The function value (the object) that’s created at runtime from this function literal is called a closure. The name arises from the act of “closing” the function literal by “capturing” the bindings of its free variables. A function literal with no free variables, such as (x: Int) => x + 1, is called a **closed term**, where a term is a bit of source code. Thus a function value created at runtime from this function literal is not a closure in the strictest sense, because (x: Int) => x + 1 is already closed as written. But any function literal with **free variables**, such as (x: Int) => x + more, is an **open term**. Therefore, any function value created at runtime from (x: Int) => x + more will by definition require that a binding for its free variable, more, be captured. The resulting function value, which will contain a reference to the captured more variable, is called a **closure**, therefore, because the function value is the end product of the act of closing the open term, (x: Int) => x + more.

This example brings up a question: what happens if more changes after the closure is created? In Scala, the answer is that the closure sees the change. For example:

scala> more = 9999

more: Int = 9999

scala> addMore(10)

res18: Int = 10009

Intuitively, Scala’s closures capture variables themselves, not the value to which variables refer. As the previous example demonstrates, the closure created for (x: Int) => x + more sees the change to more made outside the closure. The same is true in the opposite direction. Changes made by a closure to a captured variable are visible outside the closure. Here’s an example:

scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> var sum = 0

sum: Int = 0

scala> someNumbers.foreach(sum += \_)

scala> sum

res20: Int = -11

Even though it is the closure modifying sum at runtime, the resulting total, -11, is still visible outside the closure.

What if a closure accesses some variable that has several different copies as the program runs? For example, what if a closure uses a local variable of some function, and the function is invoked many times? Which instance of that variable gets used at each access?

Only one answer is consistent with the rest of the language: the instance used is the one that was active at the time the closure was created. For example, here is a function that creates and returns “increase” closures:

def makeIncreaser(more: Int) = (x: Int) => x + more

Each time this function is called it will create a new closure. Each closure will access the more variable that was active when the closure was created.

scala> val inc1 = makeIncreaser(1)

inc1: (Int) => Int = <function1>

scala> val inc9999 = makeIncreaser(9999)

inc9999: (Int) => Int = <function1>

When you call makeIncreaser(1), a closure is created and returned that captures the value 1 as the binding for more. Similarly, when you call makeIncreaser(9999), a closure that captures the value 9999 for more is returned. When you apply these closures to arguments (in this case, there’s just one argument, x, which must be passed in), the result that comes back depends on how more was defined when the closure was created:

scala> inc1(10)

res21: Int = 11

scala> inc9999(10)

res22: Int = 10009

It makes no difference that the more in this case is a parameter to a method call that has already returned. The Scala compiler rearranges things in cases like this so that the captured parameter lives out on the heap, instead of the stack, and thus can outlive the method call that created it. This rearrangement is all taken care of automatically, so you don’t have to worry about it. Capture any variable you like: val, var, or parameter.

Special function call forms

Repeated parameters

To denote a repeated parameter, place an asterisk after the type of the parameter. For example:

scala> def echo(args: String\*) =

for (arg <- args) println(arg)

echo: (args: String\*)Unit

Defined this way, echo can be called with zero to many String arguments:

scala> echo()

scala> echo("one")

one

scala> echo("hello", "world!")

hello

world!

Inside the function, the type of the repeated parameter is an Array of the declared type of the parameter. Thus, the type of args inside the echo function, which is declared as type “String\*” is actually Array[String].

scala> val arr = Array("What's", "up", "doc?")

arr: Array[java.lang.String] = Array(What's, up, doc?)

scala> echo(arr)

<console>:7: error: type mismatch;

found : Array[java.lang.String]

required: String

echo(arr)

ˆ

To accomplish this, you’ll need to append the array argument with a colon and an \_\* symbol, like this:

scala> echo(arr: \_\*)

What's

Up

doc?

This notation tells the compiler to pass each element of arr as its own argument to echo, rather than all of it as a single argument.

Named arguments

In a normal function call, the arguments in the call are matched one by one in the order of the parameters of the called function:

scala> def speed(distance: Float, time: Float): Float =

distance / time

speed: (distance: Float,time: Float)Float

scala> speed(100, 10)

res28: Float = 10.0

Named arguments allow you to pass arguments to a function in a different order.

scala> speed(time = 10, distance = 100)

res30: Float = 10.0

Default parameter values

def printTime2(out: java.io.PrintStream = Console.out, divisor: Int = 1) =

out.println("time = "+ System.currentTimeMillis()/divisor)

Listing 8.4 · A function with two parameters that have defaults.

To specify the output stream, call it like this:

printTime2(out = Console.err)

To specify the time divisor, call it like this:

printTime2(divisor = 1000)

Tail recursion

Control Abstraction

Reducing code duplication

object FileMatcher {

private def filesHere = (new java.io.File(".")).listFiles

def filesMatching(query: String, matcher: (String, String) => Boolean) = {

for (file <- filesHere; if matcher(file.getName, query))

yield file

}

def filesEnding(query: String) =

filesMatching(query, \_.endsWith(\_))

def filesContaining(query: String) =

filesMatching(query, \_.contains(\_))

def filesRegex(query: String) =

filesMatching(query, \_.matches(\_))

}

This code is already simplified, but it can actually be even shorter.

object FileMatcher {

private def filesHere = (new java.io.File(".")).listFiles

private def filesMatching(matcher: String => Boolean) =

for (file <- filesHere; if matcher(file.getName))

yield file

def filesEnding(query: String) =

filesMatching(\_.endsWith(query))

def filesContaining(query: String) =

filesMatching(\_.contains(query))

def filesRegex(query: String) =

filesMatching(\_.matches(query))

}

Listing 9.1 · Using closures to reduce code duplication.

The function literals used in the previous example, such as \_.endsWith(\_) and \_.contains(\_), are instantiated at runtime into function values that are *not* closures, because they don’t capture any free variables. Both variables used in the expression, \_.endsWith(\_), for example, are represented by underscores, which means they are taken from arguments to the function. Thus, \_.endsWith(\_) uses two bound variables, and no free variables. By contrast, the function literal \_.endsWith(query), used in the most recent example, contains one bound variable, the argument represented by the underscore, and one free variable named query. It is only because Scala supports closures that you were able to remove the query parameter from filesMatching in the most recent example, thereby simplifying the code even further.

Currying

A curried function is applied to multiple argument lists, instead of just one. Listing 9.2 shows a regular, non-curried function, which adds two Int parameters, x and y.

scala> def plainOldSum(x: Int, y: Int) = x + y

plainOldSum: (x: Int,y: Int)Int

scala> plainOldSum(1, 2)

res4: Int = 3

Listing 9.2 · Defining and invoking a “plain old” function.

By contrast, Listing 9.3 shows a similar function that’s curried. Instead of one list of two Int parameters, you apply this function to two lists of one Int parameter each.

scala> def curriedSum(x: Int)(y: Int) = x + y

curriedSum: (x: Int)(y: Int)Int

scala> curriedSum(1)(2)

res5: Int = 3

Listing 9.3 · Defining and invoking a curried function.

What’s happening here is that when you invoke curriedSum, you actually get two traditional function invocations back to back. The first function invocation takes a single Int parameter named x, and returns a function value for the second function. This second function takes the Int parameter y. You can use the placeholder notation to use curriedSum in a partially applied function expression, like this:

scala> val onePlus = curriedSum(1)\_

onePlus: (Int) => Int = <function1>

The underscore in curriedSum(1)\_ is a placeholder for the second parameter list. The result is a reference to a function that, when invoked, adds one to its sole Int argument and returns the result:

scala> onePlus(2)

res7: Int = 3

scala> val twoPlus = curriedSum(2)\_

twoPlus: (Int) => Int = <function1>

scala> twoPlus(2)

res8: Int = 4

Composition and Inheritance

Composition means one class holds a reference to another, using the referenced class to help it fulfill its mission. Inheritance is the superclass/subclass relationship.

A two-dimensional layout library

As a running example in this chapter, we’ll create a library for building and rendering two-dimensional layout elements. Each element will represent a rectangle filled with text. For convenience, the library will provide factory methods named “elem” that construct new elements from passed data. For example, you’ll be able to create a layout element containing a string using a factory method with the following signature:

elem(s: String): Element

As you can see, elements will be modeled with a type named Element. You’ll be able to call above or beside on an element, passing in a second element, to get a new element that combines the two. For example, the following expression would construct a larger element consisting of two columns, each with a height of two:

val column1 = elem("hello") above elem("\*\*\*")

val column2 = elem("\*\*\*") above elem("world")

column1 beside column2

Printing the result of this expression would give:

hello \*\*\*

\*\*\* world

Abstract classes

abstract class Element {

def contents: Array[String]

}

Listing 10.1 · Defining an abstract method and class.

Note that the contents method in class Element does not carry an abstract modifier. A method is abstract if it does not have an implementation (i.e., no equals sign or body). A class with **abstract members** must itself be declared abstract, which is done by writing an abstract modifier in front of the class keyword:

abstract class Element ...

The abstract modifier signifies that the class may have abstract members that do not have an implementation. As a result, you cannot instantiate an abstract class. If you try to do so, you’ll get a compiler error:

scala> new Element

<console>:5: error: class Element is abstract;

cannot be instantiated

new Element

ˆ

Defining parameterless methods

abstract class Element {

def contents: Array[String]

def height: Int = contents.length

def width: Int = if (height == 0) 0 else contents(0).length

}

Listing 10.2 · Defining parameterless methods width and height.

Note that none of Element’s three methods has a parameter list, not even an empty one. For example, instead of:

def width(): Int

the method is defined without parentheses:

def width: Int

Such **parameterless methods** are quite common in Scala. By contrast, methods defined with empty parentheses, such as def height(): Int, are called **empty-paren methods**.

"hello".length // no () because no side-effect

println() // better to not drop the ()

To summarize, it is encouraged style in Scala to define methods that take no parameters and have no side effects as parameterless methods, i.e., leaving off the empty parentheses. On the other hand, you should never define a method that has side-effects without parentheses, because then invocations of that method would look like a field selection. So your clients might be surprised to see the side effects. Similarly, whenever you invoke a function that has side effects, be sure to include the empty parentheses when you write the invocation. Another way to think about this is if the function you’re calling performs an operation, use the parentheses, but if it merely provides access to a property, leave the parentheses off.

Extending classes

To instantiate an element, therefore, we will need to create a subclass that extends Element and implements the abstract contents method. Listing 10.3 shows one possible way to do that:

class ArrayElement(conts: Array[String]) extends Element {

def contents: Array[String] = conts

}

Listing 10.3 · Defining ArrayElement as a subclass of Element.

Class ArrayElement is defined to extend class Element. Just like in Java, you use an extends clause after the class name to express this:

... extends Element ...

If you leave out an extends clause, the Scala compiler implicitly assumes your class extends from scala.AnyRef, which on the Java platform is the same as class java.lang.Object. Thus, class Element implicitly extends class AnyRef.

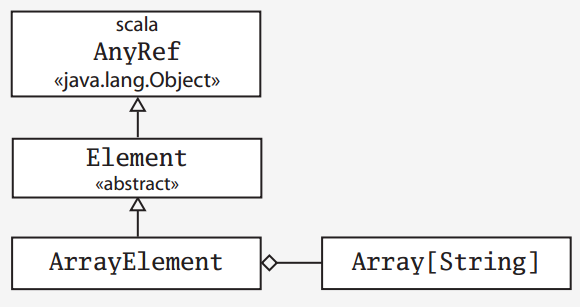


Figure 10.1 · Class diagram for ArrayElement.

**Inheritance** means that all members of the superclass are also members of the subclass, with two exceptions. First, private members of the super-class are not inherited in a subclass. Second, a member of a superclass is not inherited if a member with the same name and parameters is already implemented in the subclass. In that case we say the member of the subclass overrides the member of the superclass. If the member in the subclass is concrete and the member of the superclass is abstract, we also say that the concrete member implements the abstract one. For example, the contents method in ArrayElement overrides (or, alternatively: implements) abstract method contents in class Element. By contrast, class ArrayElement inherits the width and height methods from class Element.

scala> val ae = new ArrayElement(Array("hello", "world"))

ae: ArrayElement = ArrayElement@d94e60

scala> ae.width

res1: Int = 5

**Subtyping** means that a value of the subclass can be used wherever a value of the superclass is required. For example:

val e: Element = new ArrayElement(Array("hello"))

Variable e is defined to be of type Element, so its initializing value should also be an Element. In fact, the initializing value’s type is ArrayElement. This is OK, because class ArrayElement extends class Element, and as a result, the type ArrayElement is compatible with the type Element.

Overriding methods and fields

in Scala, fields and methods belong to the same namespace. This makes it possible for a field to override a parameterless method. For instance, you could change the implementation of contents in class ArrayElement from a method to a field without having to modify the abstract method definition of contents in class Element, as shown in Listing 10.4:

class ArrayElement(conts: Array[String]) extends Element {

val contents: Array[String] = conts

}

Listing 10.4 · Overriding a parameterless method with a field.

class WontCompile {

private var f = 0 // Won’t compile, because a field

def f = 1 // and method have the same name

}

Defining parametric fields

class ArrayElement(

val contents: Array[String]

) extends Element

Listing 10.5 · Defining contents as a parametric field.

Note that now the contents parameter is prefixed by val. This is a shorthand that defines at the same time a parameter and field with the same name. Specifically, class ArrayElement now has an (unreassignable) field contents, which can be accessed from outside the class. The field is initialized with the value of the parameter. It’s as if the class had been written as follows, where x123 is an arbitrary fresh name for the parameter:

class ArrayElement(x123: Array[String]) extends Element {

val contents: Array[String] = x123

}

You can also prefix a class parameter with var, in which case the corresponding field would be reassignable. Finally, it is possible to add modifiers such as private, protected, or override to these parametric fields, just as you can do for any other class member. Consider, for instance, the following class definitions:

class Cat {

val dangerous = false

}

class Tiger(

override val dangerous: Boolean,

private var age: Int

) extends Cat

Tiger’s definition is a shorthand for the following alternate class definition with an overriding member dangerous and a private member age:

class Tiger(param1: Boolean, param2: Int) extends Cat {

override val dangerous = param1

private var age = param2

}

Invoking superclass constructors

class LineElement(s: String) extends ArrayElement(Array(s)) {

override def width = s.length

override def height = 1

}

Listing 10.6 · Invoking a superclass constructor.

Since LineElement extends ArrayElement, and ArrayElement’s constructor takes a parameter (an Array[String]), LineElement needs to pass an argument to the primary constructor of its superclass. To invoke a superclass constructor, you simply place the argument or arguments you want to pass in parentheses following the name of the superclass. For example, class LineElement passes Array(s) to ArrayElement’s primary constructor by placing it in parentheses after the superclass ArrayElement’s name:

... extends ArrayElement(Array(s)) ...

Polymorphism and dynamic binding

You saw in Section 10.4 that a variable of type Element could refer to an object of type ArrayElement. The name for this phenomenon is **polymorphism**, which means “many shapes” or “many forms.” In this case, Element objects can have many forms. So far, you’ve seen two such forms: ArrayElement and LineElement.

method invocations on variables and expressions are **dynamically bound**. This means that the actual method implementation invoked is determined at run time based on the class of the object, not the type of the variable or expression.

abstract class Element {

def demo() {

println("Element's implementation invoked")

}

}

class ArrayElement extends Element {

override def demo() {

println("ArrayElement's implementation invoked")

}

}

class LineElement extends ArrayElement {

override def demo() {

println("LineElement's implementation invoked")

}

}

// UniformElement inherits Element’s demo

class UniformElement extends Element

If you enter this code into the interpreter, you can then define this method that takes an Element and invokes demo on it:

def invokeDemo(e: Element) {

e.demo()

}

scala> invokeDemo(new ArrayElement)

ArrayElement's implementation invoked

scala> invokeDemo(new LineElement)

LineElement's implementation invoked

scala> invokeDemo(new UniformElement)

Element's implementation invoked

Because UniformElement does not override demo, it inherits the implementation of demo from its superclass, Element. Thus, Element’s implementation is the correct implementation of demo to invoke when the class of the object is UniformElement.

Declaring final members

Sometimes when designing an Inheritance hierarchy, you want to ensure that a member cannot be overridden by subclasses. you could place a final modifier on ArrayElement’s demo method, as shown in Listing 10.7.

class ArrayElement extends Element {

final override def demo() {

println("ArrayElement's implementation invoked")

}

}

Listing 10.7 · Declaring a final method.

You may also at times want to ensure that an entire class not be subclassed. To do this you simply declare the entire class final by adding a final modifier to the class declaration. For example, Listing 10.8 shows how you would declare ArrayElement final:

final class ArrayElement extends Element {

override def demo() {

println("ArrayElement's implementation invoked")

}

}

Listing 10.8 · Declaring a final class.

Using composition and inheritance

class LineElement(s: String) extends Element {

val contents = Array(s)

override def width = s.length

override def height = 1

}

In the previous version, LineElement had an inheritance relationship with ArrayElement, from which it inherited contents. It now has a composition relationship with Array: it holds a reference to an array of strings from its own contents field.

Implementing above, beside, and toString

Traits

Traits are a fundamental unit of code reuse in Scala. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits.