Chapter 5

Basic Types and Operations

Now that you've seen classes and objects in action, it's a good time to look at Scala's basic types and operations in more depth. If you're familiar with Java, you'll be glad to find that Java's basic types and operators have the same meaning in Scala. However there are some interesting differences that will make this chapter worthwhile reading even if you're an experienced Java developer. Because some of the aspects of Scala covered in this chapter are essentially the same in Java, we've inserted notes indicating what Java developers can safely skip, to expedite your progress.

In this chapter, you'll get an overview of Scala's basic types, including `String`s and the value types `Int`, `Long`, `Short`, `Byte`, `Float`, `Double`, `Char`, and `Boolean`. You'll learn the operations you can perform on these types, including how operator precedence works in Scala expressions. You'll also learn how implicit conversions can "enrich" variants of these basic types, giving you additional operations beyond those supported by Java.

5.1 Some basic types

Several fundamental types of Scala, along with the ranges of values instances of these types may have, are shown in Table 5.1. Collectively, types `Byte`, `Short`, `Int`, `Long`, and `Char` are called *integral types*. The integral types plus `Float` and `Double` are called *numeric types*.

Other than `String`, which resides in package `java.lang`, all of the types shown in Table 5.1 are members of package `scala`.[1] For example, the full

[1]Packages, which were briefly described in Step 2 in Chapter 2, will be covered in depth in Chapter 13.

Table 5.1 · Some basic types

Value type	Range
`Byte`	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
`Short`	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
`Int`	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
`Long`	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
`Char`	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
`String`	a sequence of `Char`s
`Float`	32-bit IEEE 754 single-precision float
`Double`	64-bit IEEE 754 double-precision float
`Boolean`	`true` or `false`

name of `Int` is `scala.Int`. However, given that all the members of package `scala` and `java.lang` are automatically imported into every Scala source file, you can just use the simple names (*i.e.*, names like `Boolean`, `Char`, or `String`) everywhere.

Savvy Java developers will note that Scala's basic types have the exact same ranges as the corresponding types in Java. This enables the Scala compiler to transform instances of Scala *value types*, such as `Int` or `Double`, down to Java primitive types in the bytecodes it produces.

5.2 Literals

All of the basic types listed in Table 5.1 can be written with *literals*. A literal is a way to write a constant value directly in code.

Fast track for Java programmers
The syntax of most literals shown in this section are exactly the same as in Java, so if you're a Java master, you can safely skip much of this section. The two differences you should read about are Scala's literals for raw strings and symbols, which are described starting on page 122.

Integer literals

Integer literals for the types `Int`, `Long`, `Short`, and `Byte` come in three forms: decimal, hexadecimal, and octal. The way an integer literal begins indicates the base of the number. If the number begins with a 0x or 0X, it is hexadecimal (base 16), and may contain 0 through 9 as well as upper or lowercase digits A through F. Some examples are:

```
scala> val hex = 0x5
hex: Int = 5

scala> val hex2 = 0x00FF
hex2: Int = 255

scala> val magic = 0xcafebabe
magic: Int = -889275714
```

Note that the Scala shell always prints integer values in base 10, no matter what literal form you may have used to initialize it. Thus the interpreter displays the value of the hex2 variable you initialized with literal 0x00FF as decimal 255. (Of course, you don't need to take our word for it. A good way to start getting a feel for the language is to try these statements out in the interpreter as you read this chapter.) If the number begins with a zero, it is octal (base 8), and may, therefore, only contain digits 0 through 7. Some examples are:

```
scala> val oct = 035  // (35 octal is 29 decimal)
oct: Int = 29

scala> val nov = 0777
nov: Int = 511

scala> val dec = 0321
dec: Int = 209
```

If the number begins with a non-zero digit, and is otherwise undecorated, it is decimal (base 10). For example:

```
scala> val dec1 = 31
dec1: Int = 31

scala> val dec2 = 255
dec2: Int = 255

scala> val dec3 = 20
dec3: Int = 20
```

If an integer literal ends in an L or l, it is a Long, otherwise it is an Int. Some examples of Long integer literals are:

```
scala> val prog = 0XCAFEBABEL
prog: Long = 3405691582

scala> val tower = 35L
tower: Long = 35

scala> val of = 31l
of: Long = 31
```

If an Int literal is assigned to a variable of type Short or Byte, the literal is treated as if it were a Short or Byte type so long as the literal value is within the valid range for that type. For example:

```
scala> val little: Short = 367
little: Short = 367

scala> val littler: Byte = 38
littler: Byte = 38
```

Floating point literals

Floating point literals are made up of decimal digits, optionally containing a decimal point, and optionally followed by an E or e and an exponent. Some examples of floating-point literals are:

```
scala> val big = 1.2345
big: Double = 1.2345

scala> val bigger = 1.2345e1
bigger: Double = 12.345

scala> val biggerStill = 123E45
biggerStill: Double = 1.23E47
```

Note that the exponent portion means the power of 10 by which the other portion is multiplied. Thus, 1.2345e1 is 1.2345 *times* 10^1, which is 12.345. If a floating-point literal ends in an F or f, it is a Float, otherwise it is a Double. Optionally, a Double floating-point literal can end in D or d. Some examples of Float literals are:

```
scala> val little = 1.2345F
little: Float = 1.2345

scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

That last value expressed as a `Double` could take these (and other) forms:

```
scala> val anotherDouble = 3e5
anotherDouble: Double = 300000.0

scala> val yetAnother = 3e5D
yetAnother: Double = 300000.0
```

Character literals

Character literals are composed of any Unicode character between single quotes, such as:

```
scala> val a = 'A'
a: Char = A
```

In addition to providing an explicit character between the single quotes, you can provide an octal or hex number for the character code point preceded by a backslash. The octal number must be between `'\0'` and `'\377'`. For example, the Unicode character code point for the letter A is 101 octal. Thus:

```
scala> val c = '\101'
c: Char = A
```

A character literal can also be given as a general Unicode character consisting of four hex digits and preceded by a \u, as in:

```
scala> val d = '\u0041'
d: Char = A

scala> val f = '\u0044'
f: Char = D
```

In fact, such Unicode characters can appear anywhere in a Scala program. For instance you could also write an identifier like this:

Table 5.2 · Special character literal escape sequences

Literal	**Meaning**
\n	line feed (\u000A)
\b	backspace (\u0008)
\t	tab (\u0009)
\f	form feed (\u000C)
\r	carriage return (\u000D)
\"	double quote (\u0022)
\'	single quote (\u0027)
\\	backslash (\u005C)

```
scala> val B\u0041\u0044 = 1
BAD: Int = 1
```

This identifier is treated as identical to BAD, the result of expanding the two Unicode characters in the code above. In general, it is a bad idea to name identifiers like this, because it is hard to read. Rather, this syntax is intended to allow Scala source files that include non-ASCII Unicode characters to be represented in ASCII.

Finally, there are also a few character literals represented by special escape sequences, shown in Table 5.2. For example:

```
scala> val backslash = '\\'
backslash: Char = \
```

String literals

A string literal is composed of characters surrounded by double quotes:

```
scala> val hello = "hello"
hello: java.lang.String = hello
```

The syntax of the characters within the quotes is the same as with character literals. For example:

```
scala> val escapes = "\\\"\'"
escapes: java.lang.String = \"'
```

Because this syntax is awkward for strings that contain a lot of escape sequences or strings that span multiple lines, Scala includes a special syntax for *raw strings*. You start and end a raw string with three double quotation marks in a row ("""). The interior of a raw string may contain any characters whatsoever, including newlines, quotation marks, and special characters, except of course three quotes in a row. For example, the following program prints out a message using a raw string:

```
println("""Welcome to Ultamix 3000.
           Type "HELP" for help.""")
```

Running this code does not produce quite what is desired, however:

```
Welcome to Ultamix 3000.
           Type "HELP" for help.
```

The issue is that the leading spaces before the second line are included in the string! To help with this common situation, you can call `stripMargin` on strings. To use this method, put a pipe character (`|`) at the front of each line, and then call `stripMargin` on the whole string:

```
println("""|Welcome to Ultamix 3000.
           |Type "HELP" for help.""".stripMargin)
```

Now the code behaves as desired:

```
Welcome to Ultamix 3000.
Type "HELP" for help.
```

Symbol literals

A symbol literal is written `'`*ident*, where *ident* can be any alphanumeric identifier. Such literals are mapped to instances of the predefined class `scala.Symbol`. Specifically, the literal `'cymbal` will be expanded by the compiler to a factory method invocation: `Symbol("cymbal")`. Symbol literals are typically used in situations where you would use just an identifier in a dynamically typed language. For instance, you might want to define a method that updates a record in a database:

```
scala> def updateRecordByName(r: Symbol, value: Any) {
         // code goes here
       }
updateRecordByName: (Symbol,Any)Unit
```

The method takes as parameters a symbol indicating the name of a record field and a value with which the field should be updated in the record. In a dynamically typed language, you could invoke this operation passing an undeclared field identifier to the method, but in Scala this would not compile:

```
scala> updateRecordByName(favoriteAlbum, "OK Computer")
<console>:6: error: not found: value favoriteAlbum
       updateRecordByName(favoriteAlbum, "OK Computer")
                          ^
```

Instead, and almost as concisely, you can pass a symbol literal:

```
scala> updateRecordByName('favoriteAlbum, "OK Computer")
```

There is not much you can do with a symbol, except find out its name:

```
scala> val s = 'aSymbol
s: Symbol = 'aSymbol

scala> s.name
res20: String = aSymbol
```

Another thing that's noteworthy is that symbols are *interned.* If you write the same symbol literal twice, both expressions will refer to the exact same `Symbol` object.

Boolean literals

The `Boolean` type has two literals, `true` and `false`:

```
scala> val bool = true
bool: Boolean = true

scala> val fool = false
fool: Boolean = false
```

That's all there is to it. You are now literally[2] an expert in Scala.

[2]figuratively speaking

5.3 Operators are methods

Scala provides a rich set of operators for its basic types. As mentioned in previous chapters, these operators are actually just a nice syntax for ordinary method calls. For example, `1 + 2` really means the same thing as `(1).+(2)`. In other words, class `Int` contains a method named `+` that takes an `Int` and returns an `Int` result. This `+` method is invoked when you add two `Int`s:

```
scala> val sum = 1 + 2    // Scala invokes (1).+(2)
sum: Int = 3
```

To prove this to yourself, you can write the expression explicitly as a method invocation:

```
scala> val sumMore = (1).+(2)
sumMore: Int = 3
```

In fact, `Int` contains several *overloaded* `+` methods that take different parameter types.[3] For example, `Int` has another method, also named `+`, that takes and returns a `Long`. If you add a `Long` to an `Int`, this alternate `+` method will be invoked, as in:

```
scala> val longSum = 1 + 2L    // Scala invokes (1).+(2L)
longSum: Long = 3
```

The `+` symbol is an operator—an infix operator to be specific. Operator notation is not limited to methods like `+` that look like operators in other languages. You can use *any* method in operator notation. For example, class `String` has a method, `indexOf`, that takes one `Char` parameter. The `indexOf` method searches the string for the first occurrence of the specified character, and returns its index or `-1` if it doesn't find the character. You can use `indexOf` as an operator, like this:

```
scala> val s = "Hello, world!"
s: java.lang.String = Hello, world!

scala> s indexOf 'o'     // Scala invokes s.indexOf('o')
res0: Int = 4
```

[3] *Overloaded* methods have the same name but different argument types. More on method overloading in Section 6.11.

In addition, String offers an overloaded indexOf method that takes two parameters, the character for which to search and an index at which to start. (The other indexOf method, shown previously, starts at index zero, the beginning of the String.) Even though this indexOf method takes two arguments, you can use it in operator notation. But whenever you call a method that takes multiple arguments using operator notation, you have to place those arguments in parentheses. For example, here's how you use this other indexOf form as an operator (continuing from the previous example):

```
scala> s indexOf ('o', 5) // Scala invokes s.indexOf('o', 5)
res1: Int = 8
```

Any method can be an operator

In Scala operators are not special language syntax: any method can be an operator. What makes a method an operator is how you *use* it. When you write "s.indexOf('o')", indexOf is not an operator. But when you write "s indexOf 'o'", indexOf *is* an operator, because you're using it in operator notation.

So far, you've seen examples of *infix* operator notation, which means the method to invoke sits between the object and the parameter or parameters you wish to pass to the method, as in "7 + 2". Scala also has two other operator notations: prefix and postfix. In prefix notation, you put the method name before the object on which you are invoking the method, for example, the '-' in -7. In postfix notation, you put the method after the object, for example, the "toLong" in "7 toLong".

In contrast to the infix operator notation—in which operators take two operands, one to the left and the other to the right—prefix and postfix operators are *unary*: they take just one operand. In prefix notation, the operand is to the right of the operator. Some examples of prefix operators are -2.0, !found, and ~0xFF. As with the infix operators, these prefix operators are a shorthand way of invoking methods. In this case, however, the name of the method has "unary_" prepended to the operator character. For instance, Scala will transform the expression -2.0 into the method invocation "(2.0).unary_-". You can demonstrate this to yourself by typing the method call both via operator notation and explicitly:

```
scala> -2.0                        // Scala invokes (2.0).unary_-
res2: Double = -2.0

scala> (2.0).unary_-
res3: Double = -2.0
```

The only identifiers that can be used as prefix operators are `+`, `-`, `!`, and `~`. Thus, if you define a method named `unary_!`, you could invoke that method on a value or variable of the appropriate type using prefix operator notation, such as `!p`. But if you define a method named `unary_*`, you wouldn't be able to use prefix operator notation, because `*` isn't one of the four identifiers that can be used as prefix operators. You could invoke the method normally, as in `p.unary_*`, but if you attempted to invoke it via `*p`, Scala will parse it as if you'd written `*.p`, which is probably not what you had in mind![4]

Postfix operators are methods that take no arguments, when they are invoked without a dot or parentheses. In Scala, you can leave off empty parentheses on method calls. The convention is that you include parentheses if the method has side effects, such as `println()`, but you can leave them off if the method has no side effects, such as `toLowerCase` invoked on a `String`:

```
scala> val s = "Hello, world!"
s: java.lang.String = Hello, world!

scala> s.toLowerCase
res4: java.lang.String = hello, world!
```

In this latter case of a method that requires no arguments, you can alternatively leave off the dot and use postfix operator notation:

```
scala> s toLowerCase
res5: java.lang.String = hello, world!
```

In this case, `toLowerCase` is used as a postfix operator on the operand `s`.

To see what operators you can use with Scala's basic types, therefore, all you really need to do is look at the methods declared in the type's classes in the Scala API documentation. Given that this is a Scala tutorial, however, we'll give you a quick tour of most of these methods in the next few sections.

[4] All is not necessarily lost, however. There is an extremely slight chance your program with the *p might compile as C++.

Fast track for Java programmers
Many aspects of Scala described in the remainder of this chapter are the same as in Java. If you're a Java guru in a rush, you can safely skip to Section 5.7 on page 132, which describes how Scala differs from Java in the area of object equality.

5.4 Arithmetic operations

You can invoke arithmetic methods via infix operator notation for addition (+), subtraction (-), multiplication (*), division (/), and remainder (%), on any numeric type. Here are some examples:

```
scala> 1.2 + 2.3
res6: Double = 3.5

scala> 3 - 1
res7: Int = 2

scala> 'b' - 'a'
res8: Int = 1

scala> 2L * 3L
res9: Long = 6

scala> 11 / 4
res10: Int = 2

scala> 11 % 4
res11: Int = 3

scala> 11.0f / 4.0f
res12: Float = 2.75

scala> 11.0 % 4.0
res13: Double = 3.0
```

When both the left and right operands are integral types (`Int`, `Long`, `Byte`, `Short`, or `Char`), the `/` operator will tell you the whole number portion of the quotient, excluding any remainder. The `%` operator indicates the remainder of an implied integer division.

The floating-point remainder you get with `%` is not the one defined by the IEEE 754 standard. The IEEE 754 remainder uses rounding division, not truncating division, in calculating the remainder, so it is quite different from the integer remainder operation. If you really want an IEEE 754 remainder, you can call `IEEEremainder` on `scala.math`, as in:

```
scala> math.IEEEremainder(11.0, 4.0)
res14: Double = -1.0
```

The numeric types also offer unary prefix operators + (method `unary_+`) and - (method `unary_-`), which allow you to indicate a literal number is positive or negative, as in `-3` or `+4.0`. If you don't specify a unary + or -, a literal number is interpreted as positive. Unary + exists solely for symmetry with unary -, but has no effect. The unary - can also be used to negate a variable. Here are some examples:

```
scala> val neg = 1 + -3
neg: Int = -2

scala> val y = +3
y: Int = 3

scala> -neg
res15: Int = 2
```

5.5 Relational and logical operations

You can compare numeric types with relational methods greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=), which yield a `Boolean` result. In addition, you can use the unary '`!`' operator (the `unary_!` method) to invert a `Boolean` value. Here are a few examples:

```
scala> 1 > 2
res16: Boolean = false

scala> 1 < 2
res17: Boolean = true

scala> 1.0 <= 1.0
res18: Boolean = true

scala> 3.5f >= 3.6f
res19: Boolean = false
```

```
scala> 'a' >= 'A'
res20: Boolean = true

scala> val thisIsBoring = !true
thisIsBoring: Boolean = false

scala> !thisIsBoring
res21: Boolean = true
```

The logical methods, logical-and (&&) and logical-or (||), take `Boolean` operands in infix notation and yield a `Boolean` result. For example:

```
scala> val toBe = true
toBe: Boolean = true

scala> val question = toBe || !toBe
question: Boolean = true

scala> val paradox = toBe && !toBe
paradox: Boolean = false
```

The logical-and and logical-or operations are *short-circuited* as in Java: expressions built from these operators are only evaluated as far as needed to determine the result. In other words, the right-hand side of logical-and and logical-or expressions won't be evaluated if the left-hand side determines the result. For example, if the left-hand side of a logical-and expression evaluates to `false`, the result of the expression will definitely be `false`, so the right-hand side is not evaluated. Likewise, if the left-hand side of a logical-or expression evaluates to `true`, the result of the expression will definitely be `true`, so the right-hand side is not evaluated. For example:

```
scala> def salt() = { println("salt"); false }
salt: ()Boolean

scala> def pepper() = { println("pepper"); true }
pepper: ()Boolean

scala> pepper() && salt()
pepper
salt
res22: Boolean = false
```

```
scala> salt() && pepper()
salt
res23: Boolean = false
```

In the first expression, `pepper` and `salt` are invoked, but in the second, only `salt` is invoked. Given `salt` returns `false`, there's no need to call `pepper`.

> **Note**
> You may be wondering how short-circuiting can work given operators are just methods. Normally, all arguments are evaluated before entering a method, so how can a method avoid evaluating its second argument? The answer is that all Scala methods have a facility for delaying the evaluation of their arguments, or even declining to evaluate them at all. The facility is called *by-name parameters* and is discussed in Section 9.5.

5.6 Bitwise operations

Scala enables you to perform operations on individual bits of integer types with several bitwise methods. The bitwise methods are: bitwise-and (&), bitwise-or (|), and bitwise-xor (ˆ).[5] The unary bitwise complement operator (~, the method `unary_~`), inverts each bit in its operand. For example:

```
scala> 1 & 2
res24: Int = 0

scala> 1 | 2
res25: Int = 3

scala> 1 ˆ 3
res26: Int = 2

scala> ~1
res27: Int = -2
```

The first expression, `1 & 2`, bitwise-ands each bit in 1 (0001) and 2 (0010), which yields 0 (0000). The second expression, `1 | 2`, bitwise-ors each bit in

[5]The bitwise-xor method performs an *exclusive or* on its operands. Identical bits yield a 0. Different bits yield a 1. Thus 0011 ˆ 0101 yields 0110.

the same operands, yielding 3 (0011). The third expression, `1 ^ 3`, bitwise-xors each bit in 1 (0001) and 3 (0011), yielding 2 (0010). The final expression, `~1`, inverts each bit in 1 (0001), yielding -2, which in binary looks like 11111111111111111111111111111110.

Scala integer types also offer three shift methods: shift left (<<), shift right (>>), and unsigned shift right (>>>). The shift methods, when used in infix operator notation, shift the integer value on the left of the operator by the amount specified by the integer value on the right. Shift left and unsigned shift right fill with zeroes as they shift. Shift right fills with the highest bit (the sign bit) of the left-hand value as it shifts. Here are some examples:

```
scala> -1 >> 31
res28: Int = -1

scala> -1 >>> 31
res29: Int = 1

scala> 1 << 2
res30: Int = 4
```

-1 in binary is 11111111111111111111111111111111. In the first example, `-1 >> 31`, -1 is shifted to the right 31 bit positions. Since an `Int` consists of 32 bits, this operation effectively moves the leftmost bit over until it becomes the rightmost bit.[6] Since the >> method fills with ones as it shifts right, because the leftmost bit of -1 is 1, the result is identical to the original left operand, 32 one bits, or -1. In the second example, `-1 >>> 31`, the leftmost bit is again shifted right until it is in the rightmost position, but this time filling with zeroes along the way. Thus the result this time is binary 00000000000000000000000000000001, or 1. In the final example, `1 << 2`, the left operand, 1, is shifted left two positions (filling in with zeroes), resulting in binary 00000000000000000000000000000100, or 4.

5.7 Object equality

If you want to compare two objects for equality, you can use either ==, or its inverse `!=`. Here are a few simple examples:

[6]The leftmost bit in an integer type is the sign bit. If the leftmost bit is 1, the number is negative. If 0, the number is positive.

```
scala> 1 == 2
res31: Boolean = false

scala> 1 != 2
res32: Boolean = true

scala> 2 == 2
res33: Boolean = true
```

These operations actually apply to all objects, not just basic types. For example, you can use == to compare lists:

```
scala> List(1, 2, 3) == List(1, 2, 3)
res34: Boolean = true

scala> List(1, 2, 3) == List(4, 5, 6)
res35: Boolean = false
```

Going further, you can compare two objects that have different types:

```
scala> 1 == 1.0
res36: Boolean = true

scala> List(1, 2, 3) == "hello"
res37: Boolean = false
```

You can even compare against `null`, or against things that might be `null`. No exception will be thrown:

```
scala> List(1, 2, 3) == null
res38: Boolean = false

scala> null == List(1, 2, 3)
res39: Boolean = false
```

As you see, == has been carefully crafted so that you get just the equality comparison you want in most cases. This is accomplished with a very simple rule: first check the left side for `null`, and if it is not `null`, call the `equals` method. Since `equals` is a method, the precise comparison you get depends on the type of the left-hand argument. Since there is an automatic null check, you do not have to do the check yourself.[7]

[7] The automatic check does not look at the right-hand side, but any reasonable `equals` method should return `false` if its argument is `null`.

This kind of comparison will yield `true` on different objects, so long as their contents are the same and their `equals` method is written to be based on contents. For example, here is a comparison between two strings that happen to have the same five letters in them:

```
scala> ("he"+"llo") == "hello"
res40: Boolean = true
```

How Scala's == differs from Java's

In Java, you can use == to compare both primitive and reference types. On primitive types, Java's == compares value equality, as in Scala. On reference types, however, Java's == compares *reference equality*, which means the two variables point to the same object on the JVM's heap. Scala provides a facility for comparing reference equality, as well, under the name `eq`. However, `eq` and its opposite, `ne`, only apply to objects that directly map to Java objects. The full details about `eq` and `ne` are given in Sections 11.1 and 11.2. Also, see Chapter 30 on how to write a good `equals` method.

5.8 Operator precedence and associativity

Operator precedence determines which parts of an expression are evaluated before the other parts. For example, the expression `2 + 2 * 7` evaluates to 16, not 28, because the * operator has a higher precedence than the + operator. Thus the multiplication part of the expression is evaluated before the addition part. You can of course use parentheses in expressions to clarify evaluation order or to override precedence. For example, if you really wanted the result of the expression above to be 28, you could write the expression like this:

```
(2 + 2) * 7
```

Given that Scala doesn't have operators, per se, just a way to use methods in operator notation, you may be wondering how operator precedence works. Scala decides precedence based on the first character of the methods used in operator notation (there's one exception to this rule, which will be discussed below). If the method name starts with a `*`, for example, it will have a higher precedence than a method that starts with a `+`. Thus `2 + 2 * 7` will be evaluated as `2 + (2 * 7)`, and `a +++ b *** c` (in which `a`, `b`, and `c` are variables, and `+++` and `***` are methods) will be evaluated `a +++ (b *** c)`, because the `***` method has a higher precedence than the `+++` method.

Table 5.3 · Operator precedence

(all other special characters)
`* / %`
`+ -`
`:`
`= !`
`< >`
`&`
`^`
`|`
(all letters)
(all assignment operators)

Table 5.3 shows the precedence given to the first character of a method in decreasing order of precedence, with characters on the same line having the same precedence. The higher a character is in this table, the higher the precedence of methods that start with that character. Here's an example that illustrates the influence of precedence:

```
scala> 2 << 2 + 2
res41: Int = 32
```

The `<<` method starts with the character `<`, which appears lower in Table 5.3 than the character `+`, which is the first and only character of the `+` method. Thus `<<` will have lower precedence than `+`, and the expression will be evaluated by first invoking the `+` method, then the `<<` method, as in `2 << (2 + 2)`. `2 + 2` is 4, by our math, and `2 << 4` yields 32. Here's another example:

```
scala> 2 + 2 << 2
res42: Int = 16
```

Since the first characters are the same as in the previous example, the methods will be invoked in the same order. First the + method will be invoked, then the << method. So 2 + 2 will again yield 4, and 4 << 2 is 16.

The one exception to the precedence rule, alluded to above, concerns *assignment operators*, which end in an equals character. If an operator ends in an equals character (=), and the operator is not one of the comparison operators <=, >=, ==, or !=, then the precedence of the operator is the same as that of simple assignment (=). That is, it is lower than the precedence of any other operator. For instance:

```
x *= y + 1
```

means the same as:

```
x *= (y + 1)
```

because *= is classified as an assignment operator whose precedence is lower than +, even though the operator's first character is *, which would suggest a precedence higher than +.

When multiple operators of the same precedence appear side by side in an expression, the *associativity* of the operators determines the way operators are grouped. The associativity of an operator in Scala is determined by its *last* character. As mentioned on page 87 of Chapter 3, any method that ends in a ':' character is invoked on its right operand, passing in the left operand. Methods that end in any other character are the other way around. They are invoked on their left operand, passing in the right operand. So a * b yields a.*(b), but a ::: b yields b.:::(a).

No matter what associativity an operator has, however, its operands are always evaluated left to right. So if a is an expression that is not just a simple reference to an immutable value, then a ::: b is more precisely treated as the following block:

```
{ val x = a; b.:::(x) }
```

In this block a is still evaluated before b, and then the result of this evaluation is passed as an operand to b's ::: method.

This associativity rule also plays a role when multiple operators of the same precedence appear side by side. If the methods end in ':', they are grouped right to left; otherwise, they are grouped left to right. For example, `a ::: b ::: c` is treated as `a ::: (b ::: c)`. But `a * b * c`, by contrast, is treated as `(a * b) * c`.

Operator precedence is part of the Scala language. You needn't be afraid to use it. Nevertheless, it is good style to use parentheses to clarify what operators are operating upon what expressions. Perhaps the only precedence you can truly count on other programmers knowing without looking up is that multiplicative operators, `*`, `/`, and `%`, have a higher precedence than the additive ones `+` and `-`. Thus even if `a + b << c` yields the result you want without parentheses, the extra clarity you get by writing `(a + b) << c` may reduce the frequency with which your peers utter your name in operator notation, for example, by shouting in disgust, "`bills !*&^%~ code!`".[8]

5.9 Rich wrappers

You can invoke many more methods on Scala's basic types than were described in the previous sections. A few examples are shown in Table 5.4. These methods are available via *implicit conversions*, a technique that will be described in detail in Chapter 21. All you need to know for now is that for each basic type described in this chapter, there is also a "rich wrapper" that provides several additional methods. To see all the available methods on the basic types, therefore, you should look at the API documentation on the rich wrapper for each basic type. Those classes are listed in Table 5.5.

5.10 Conclusion

The main take-aways from this chapter are that operators in Scala are method calls, and that implicit conversions to rich variants exist for Scala's basic types that add even more useful methods. In the next chapter, we'll show you what it means to design objects in a functional style that gives new implementations of some of the operators that you have seen in this chapter.

[8] By now you should be able to figure out that given this code, the Scala compiler would invoke `(bills.!*&^%~(code)).!()`.

Table 5.4 · Some rich operations

Code	**Result**
`0 max 5`	`5`
`0 min 5`	`0`
`-2.7 abs`	`2.7`
`-2.7 round`	`-3L`
`1.5 isInfinity`	`false`
`(1.0 / 0) isInfinity`	`true`
`4 to 6`	`Range(4, 5, 6)`
`"bob" capitalize`	`"Bob"`
`"robert" drop 2`	`"bert"`

Table 5.5 · Rich wrapper classes

Basic type	**Rich wrapper**
`Byte`	`scala.runtime.RichByte`
`Short`	`scala.runtime.RichShort`
`Int`	`scala.runtime.RichInt`
`Char`	`scala.runtime.RichChar`
`Float`	`scala.runtime.RichFloat`
`Double`	`scala.runtime.RichDouble`
`Boolean`	`scala.runtime.RichBoolean`
`String`	`scala.collection.immutable.StringOps`