

Chapter 2

Scala Fundamentals

In this chapter, we will cover building blocks of the Scala programming language. As name suggests, building blocks are like alphabets of a programming language. One of the observations we made both in industry as well as in academia is that learners tend to overlook basic building blocks, thinking that the subject matter is too basic. This creates a knowledge gap and when these building blocks are combined with complex algorithms, the combination looks harder to comprehend and requires learners to come back and learn the building blocks again. To avoid this, you should try to learn the building blocks well at the first attempt by paying attention. In isolation, building blocks look simple.

2.1 Literals

We use literals to create programs. They are directly or indirectly connected with natural languages and how we perform computations. For example, *String* is a collection of characters. But it turns out that we need individual characters as well. From a machine's point of view, this categorization helps to allocate memory efficiently.

2.1.1 Character Literals

Have you ever noticed the use of a single character in your daily life? Most probably the answer is yes. It might be for tagging items, it might be for enumerating items, etc. Well, it could equally be your grades. Scala allows us to express a single character using a character literal. A character literal can be defined by enclosing a single character in quotes. It can be either a printable unicode character or an escape sequence, defined in Section 2.1.6.

Valid characters:

```
`a'      `\\u0061'      `\\t'      `\\n'
```

Sample declaration:

```
val b:Char = `b'
```

The first one is character *a* and the second one is also character *a*, because unicode *0061* is for letter *a*. The last two characters print tab and new line, respectively. Please note the quotes: unlike how it is printed here, you have to use the same single quote, on both sides, when you type the code.

2.1.2 String Literals

Whether we are reading a newspaper on-line or reading a book, we use text extensively. Scala facilitates textual representation with string literals. A string is a sequence of characters. A string literal can be defined by enclosing a sequence of characters in double quotes. The characters can be either printable unicode characters or escape sequences, defined in Section 2.1.6.

Valid strings:

```
"Welcome"      "Welcome\\tto\\tprogramming"
"Scala is fun"
"\"Scala\" supports functional programming"

""" This is an example
of multi-line
string literals. """
```

Sample declaration:

```
val c:String = "Scala"
```

The second string literal above has a tab character in between words and hence the output is *Welcome to programming*, with spacing between words equal to the tab width. If we need double quotes, then these should be escaped, as shown in the fourth string literal. The output of the fourth string literal is *“Scala” supports functional programming*. The last string literal is an example of a multi-line string and it should be enclosed by three consecutive double quotes.

We can perform a rich set of operations on a string. For example, *reverse* reverses a given string: *“this”.reverse* prints *siht*. In order to split a sentence into words *split* can be used: *“Scala is powerful”.split(“ ”)* prints *Array[String] = Array(Scala, is, powerful)*. Similarly, a word in a string can be replaced by using *replace*: *“Scala is powerful”.replace(“powerful”, “fun”)* results *Scala is fun*.

2.1.3 Integer Literals

We use whole numbers a lot in our daily lives. We can express these numbers using Scala integer literals. Integer literals are widely used and are of two types based on length—Int or Long. Long integers are expressed with a suffix l or L. For example, 25 is an integer and 25L is a long integer. The permitted values for type Int are from -2^{31} to $2^{31} - 1$, inclusive. Auto conversion happens for small type: Byte, Short, and Char. When a type falls within the range of a smaller type, the number is converted to that smaller type along with the type information. Table 2.1 presents integer types and corresponding ranges.

Table 2.1: Integer Literals and Ranges

Type	Range
Byte	-2^7 to $2^7 - 1$
Short	-2^{15} to $2^{15} - 1$
Char	0 to $2^{16} - 1$
Int	-2^{31} to $2^{31} - 1$

Valid integers:

```
5 -5 0 25 0xAF
```

Sample declaration:

```
val a:Int = 5
```

Invalid declaration:

```
val d:Int = 3999999999
```

It is invalid, because the value assigned is out of integer range, as discussed above.

2.1.4 Floating Point Literals

In mathematics, we use real numbers frequently. Scala allows us to express those real numbers with the help of floating point numbers, called floating point literals. Floating point literals cover Float as well as Double. Float covers all IEEE 754 32-bit single-precision binary floating point values and Double covers all IEEE 754 64-bit double-precision binary floating point values. Float types have optional suffixes: f, F, d, or D.

Valid floating point numbers:

```
1e10f 1e-10f 5.5 .5
```

Sample declaration:

```
val a = 1e-10f
```

Invalid declaration:

```
val a: Int = 1e-10f
```

It is invalid, because the value on the right hand side is not an integer value. The right data type, in this case, is Float.

2.1.5 Boolean Literals

We need a way to express yes or no, equivalently true or false. In our daily lives, we do that quite a lot. Similarly, Scala has a feature called boolean literal, which allows us to express yes and no.

Valid boolean values:

```
true false
```

Sample declarations:

```
val a = true
val b: Boolean = false
```

2.1.6 Escape Sequences

Escape sequences can be used to print special characters. The following escape sequences can be used with character and string literals.

<code>\b</code>	<code>\u0008</code>	Back space (BS)
<code>\t</code>	<code>\u0009</code>	Horizontal tab (HT)
<code>\n</code>	<code>\u000a</code>	Line feed (LF)
<code>\f</code>	<code>\u000c</code>	Form feed (FF)
<code>\r</code>	<code>\u000d</code>	Carriage return (CR)
<code>"</code>	<code>\u0022</code>	Double quote (")
<code>'</code>	<code>\u0027</code>	Single quote (')
<code>\\</code>	<code>\u005c</code>	Back slash (\)

So when the statement `print("a\nb")` is executed, characters a and b are printed in separate lines. First, character a is printed, then new line character is printed, which causes b to be printed in a separate line.

2.1.7 Symbol Literals

Symbol is a case class. We will discuss classes in Chapter 3. A symbol 'y' is a shortcut for the expression `scala.Symbol("y")`. *Symbol* is a case class and can be found in package *scala*, with the following definition.

```
package scala
final case class Symbol private (name: String) {
  override def toString: String = "'" + name
}
```

Figure 2.1 shows a typical identifier comparison with *eq* operator (or method). The output is *true*. A string comparison may involve character to character comparison, in some cases, and hence lookups tend to be more efficient with symbol literals, as *eq* can be applied. Comparisons, with symbol literals, are constant time (i.e. $O(1)$).

Fig. 2.1: Symbol literals

```
object Literals {
  def main(args: Array[String]): Unit = {
    val a = 'sampleIdentifier
    println("sampleIdentifier" eq a.name)
  }
}
```

2.1.8 Other Lexical Elements

1. Whitespace and Comments

In English, words are separated by a space, approximately equivalent to the width of one character. Similarly, in Scala, tokens are separated by a whitespace; tokens can also be separated by comments. Scala has two types of comments—single line and multi-line. Single line comments start with *//* and extend to the end of line. Multi-line comments are embedded within */** and **/*. It is a good practice to comment our code. See the code fragment below.

```
...
// Length
val a = 5
// Breadth
val b = 4
val area = a * b
...
```

The code is certainly more comprehensible, with the help of comments; maintenance becomes a lot easier, specially when the program is to be maintained by a different person. Similarly, let's look at the multi-line comment below.

```
...
/*
  This program calculates an area
  of a rectangle.
*/
val length = 5
val breadth = 4
val area = length * breadth
...
```

The code fragment above demonstrates a multi-line comment. Comparing this comment with the previous comment should give you an idea of when to use each of them. Please also note the identifier names in these two code fragments.

2. Newline Characters

Statements in Scala can be terminated by semicolons or newlines. In other words, semicolons are optional, if we use newlines as separators, and newlines are optional, if we use semicolons as separators. Let's look at the four code fragments below.

```
...
val side = 5
val area = side * side
...

...
val side = 5;
val area = side * side;
...

...
val side = 5; val area = side * side;
...

...
val side = 5; val area = side * side
println(area)
...
```

All of the above four code fragments are syntactically correct. You can follow the approach that best suits you, if you have the liberty to decide. If you are working in a team environment, it is good to have common conventions.

2.2 Identifiers and Reserved Words

Think of a quadratic equation of the form $ax^2 + bx + c = 0$. x is a variable and a , b , and c are constants. All of these are identifiers; they identify or represent values. It is important to remember that there are rules that govern what kind of values they hold, including whether they can hold multiple values. Also please recall the introductory concepts from Chapter 1. We are discussing elements of a typical computing environment. Our thought processes should be geared toward how we can organize the computing elements so that we can achieve our computational goals.

Similarly, an identifier in Scala denotes a computational element. A computational element can be an integer, a string, a character, a class, etc. An identifier can start with a letter and the starting letter can be followed by any arbitrary sequence of letters, digits, or underscore. The '\$' character should not be used to define identifiers as it is reserved for compiler-generated identifiers. Reserved words are pre-defined words that cannot be used as identifiers. Table 2.2 presents reserved words in Scala.

Table 2.2: Reserved Words

abstract	case	catch	class	def
do	else	extends	false	final
finally	for	forSome	if	implicit
import	lazy	match	new	null
object	override	package	private	protected
return	sealed	super	this	throw
trait	try	true	type	val
var	while	with	yield	
-	:	=	>	<-
				<:
				<%
				>:
				#
				@

Valid identifiers:

```
length    _y    +    _MIN    green_?
```

2.3 Types

From a user’s perspective, types are categories that have common properties. For example, *Int* and *Long* are both integers. From our arithmetic knowledge, we know that all integers have some common properties that can be leveraged for a computational goal. For example, when we add two integers, digit by digit, there is something called carry, which is added to the next higher position digit.

Scala has many fundamental types, which can be used as building blocks for custom defined types. *Byte*, *Short*, *Int*, *Long*, *Char* are some of the basic types. *Float*

and *Double* are types to represent decimal numbers. All of these types together are called *numeric types*. The type *String* is used to represent text; it is part of *java.lang* package. All other types described here are part of *scala* package.

```
object TypesDemo {
  def main(args: Array[String]): Unit = {
    val length: Int = 5
    val breadth = 2.5
    val area = length * breadth
    println(area)
    val problemName: String = "Area of a rectangle"
    val purpose = "Practice"
    println(purpose+": "+problemName)
  }
}
```

Fig. 2.2: Types Demonstration

Figure 2.2 shows three different type declarations—*Int*, *Float*, and *String*. Please note that *length* is explicitly declared as *Int* but the type for *breadth* comes from type inference. This is one of the advantages of programming in Scala; it has the capability to infer type based on the corresponding value. So *breadth* gets type *Float* and subsequently the type of the area is inferred as *Float*, because when an integer is multiplied with a floating point number, the result is a floating point number.

In the case of Scala, when an *Int* is multiplied with a *Float*, the resultant type is *Float*. Similarly, the first string, *problemName*, is declared explicitly, whereas the type for second string, *purpose*, is derived from its value. The output for the second *println* is the concatenation of three strings, the second one being the value within double quotes. The output for the first *println* is 12.5.

2.4 Declarations and Definitions

A declaration is a way to tell the Scala compiler about names, types, parameters, etc. Using definitions, we can provide detail information including values and steps for computation. A value declaration takes the form *val x: T*; a value definition takes the form *val x: T = e*, where *val* is a reserved word to tell the compiler that the corresponding identifier is an immutable value, which means it cannot be changed later; *x* is an identifier; *T* is a type, and *e* is an expression. So *x* gets a value, which is a result of evaluation of the expression *e*. When we explicitly specify the type *T*, the result of evaluation of expression *e* should be *T*, otherwise it is a compile time error.

Figure 2.3 shows value declarations as well as definitions. This program has four values: *itemName*, *quantity*, *priceInDollar*, and *totalPrice*. Scala's values are constants, which means re-assignment is a compile time error. The first three identifiers


```
object DeclarationsDefinitionsDemo {  
  def main(args: Array[String]): Unit = {  
    val itemName = "Orange"  
    val quantity = 5  
    val priceInDollar = 2  
    val totalPrice: Int = quantity * priceInDollar  
    println("Total price of "+itemName+" = "+totalPrice)  
  }  
}
```

Fig. 2.3: Value Declaration and Definition

get their type through Scala's type inference, whereas the fourth one is explicitly declared. Declaration and definition are done within the same line of code. The last line of code performs auto conversion of integer values to corresponding string values. If we have `+` in between string and numeric values, the numeric values are automatically converted to string values and then concatenated.

A variable declaration has the form *var x: T*; a variable definition takes the form *var x: T = e*. The main difference between *val* and *var* is that *var* can be re-assigned a value. *A very important thing to note is that var may not be safe for parallel programming. Sequential programs written using val are easier to transform to parallel programs.* So you are highly encouraged to program using *val* and avoid *var* whenever possible.

```
object VarDeclarationsDefinitionDemo {  
  def main(args: Array[String]): Unit = {  
    var itemName = "Orange"  
    var quantity = 5  
    var priceInDollar = 2  
    var totalPrice = quantity * priceInDollar  
    println(itemName+": "+totalPrice)  
    itemName = "Apple"  
    quantity = 6  
    priceInDollar = 3  
    totalPrice = quantity * priceInDollar  
    println(itemName+": "+totalPrice)  
  }  
}
```

Fig. 2.4: Variable Declaration and Definition

Figure 2.4 shows variable declarations as well as variable definitions. Please note that all the variables have been re-used. First, we assign values for orange and calculate total price and print the value. Next, we use the same variables and assign values corresponding to apple and perform a similar calculation, which is followed by a *println* statement.

This is perfectly fine in a sequential execution. We will see later how to represent real world objects with Scala objects. Once we do that, there will be some getter and setter methods associated with each variable. If the setter methods are called from concurrent processes then the consistency becomes important. That's when we start realizing the importance of immutability.

Next, let's take a short example of type declaration. Figure 2.5 shows a type alias definition. `intList` can be used as `List[Int]`, which is exactly what is being done in the next LOC, `val a: intList = List(1,2,3)`. Here `a` gets its type from type alias, which in turn gets its type from its definition. So the list creation on the right hand side should comply with the type definition of type alias. If it doesn't, then it is a compile time error.

```
object TypeDeclarationDefinitionDemo {  
  def main(args: Array[String]): Unit = {  
    type intList = List[Int]  
    val a: intList = List(1,2,3)  
    a.foreach(println)  
  }  
}
```

Fig. 2.5: Type Definition

We will cover the remaining declarations and definitions in later chapters.

2.5 Expressions

Expressions can be evaluated and the result of evaluation can be assigned or returned to the caller. Also an expression has a type, which may come from Scala inference. If type is declared as in `val x: T = e`, then the type of `e` must match with `T`. The simplest expressions are literals. For example, in `val a = 5`, `val` is a reserved word, which directs the compiler that identifier `a` is value, `=` is an assignment operator, and `5` is a simple expression, an integer literal. In Scala, an expression can be any one of the following types:

1. Literal
2. The `null` value
3. Designator
4. `this` and `super`
5. Named and default argument
6. Method values
7. Tuples
8. Instance creation expression
9. Block

10. Typed expression
11. Annotated expression
12. Conditional expression
13. While loop expression
14. Do loop expression
15. For loop
16. Return expression
17. Throw expression
18. Try expression
19. Constant expression

We will discuss each of these expression types in the relevant chapters. For now, let's look at an example with several expressions. Figure 2.6 shows three different types of expression. The first one has digit *6* as an expression, which is a literal expression. The second one is $a + 1$. For this expression, type inference uses the type of a , `Float`, to determine the type of b . And of course, $a + 1$ is evaluated and assigned to b .

```
object ExpressionsDemo {  
  def main(args: Array[String]): Unit = {  
    val a: Float = 6  
    val b = a + 1  
    println(b)  
    val c = {  
      val d = 2  
      val e = 3  
      d + e  
    }  
    println(c)  
  }  
}
```

Fig. 2.6: Expressions

The next expression is a block expression. Scala has an interesting feature that allows us to assign a block to an identifier and use it as a value. The block is evaluated and types are inferred before assigning value to the variable c . The type of d and e can be inferred based on their value on the right hand side, and $d + e$ will have the same type, which is `Int`, in this case. Now, the last statement of the block is a return statement; Scala does not require us to write `return` explicitly. So the value of $d + e$ is assigned to c , which is 5. The first `println` prints 7.0 and the second `println` prints 5.

2.6 Conclusion

In this chapter, we covered character literals, which are the most fundamental building blocks. Then we discussed string literals; these are widely used as text represents a large percentage of information processing. Similarly, we covered numerals and their range. Escape sequences are important literals to remember as these become a source of bugs if used improperly. Symbol literals are not widely used but these are faster to lookup. Comments are an important part of program documentation. New-line sounds obvious, but in Scala it has a special meaning, i.e., a newline eliminates the need for a semicolon.

2.7 Review Questions

1. What is the difference between ‘a’ and “a”?
2. Where does the type come from in `val languageName = “Scala”`?
3. What is the range of type `Int` in Scala?
4. Is `0xAD` a valid integer?
5. Is `0xBG` a valid integer?
6. Is `0xCF` a valid integer?
7. Is `val c: Int = 1e-20f` a valid declaration?
8. Is `val x = true` a valid declaration and a valid definition?
9. Is `val y: Boolean = 1` a valid definition?
10. How is a multi-line comment written?
11. Can a new line be used as a statement terminator?
12. Can a semicolon be used as a statement terminator?
13. When can’t a semicolon be replaced by a new line for statement termination?
14. Is `forSome` a reserved word?
15. Why is a type important?
16. What is the simplest possible expression?

2.8 Problems

1. Write a program to reverse the letters of a word. Hint: you can use the Scala library.
2. Write a program to read two words from a keyboard, concatenate them, and then reverse the letters.
3. Write a program that uses a block to read length and breadth as double precision floating point numbers. This block also calculates the area and returns or assigns the result to a variable called *area*. Print the area on the console.

2.9 Answers to Review Questions

1. 'a' is a character literal and "a" is a string literal.
2. In *val languageName = "Scala"*, the type comes from inference.
3. The range of type *Int* is 2^{-31} to $2^{31} - 1$.
4. *0xAD* is a valid integer, it is in hexadecimal representation.
5. *0xBG* is not a valid integer, *G* is not part of hexadecimal representation.
6. *oxCF* is not a valid integer, it should start with *0x*, not *ox*.
7. *val c: Int = 1e-20f* is not a valid declaration, because the value on the right hand side is not an integer value.
8. *val x = true* is a valid declaration and definition, the type is inferred automatically.
9. *val y: Boolean = 1* is not a valid definition because *1* is not part of boolean literals.
10. A multi-line comment is embedded in */* */*.
11. Yes, a new line can be used as a statement terminator.
12. Yes, a semicolon can be used as a statement terminator.
13. If there are two statements in a single line, the first one must be terminated by using a semicolon.
14. Yes, *forSome* is a Scala reserved word.
15. A type helps to categorize building blocks and makes it convenient to analyze a program. From the machine's perspective, types are used for memory allocation.
16. The simplest possible expression is a literal.

2.10 Solutions to Problems

```
1. object ReverseLetters {  
    def main(args: Array[String]) {  
        print("Please enter a word: ")  
        val word = scala.io.StdIn.readLine();  
        val wordReversed = word.toString.reverse  
        println(wordReversed)  
    }  
}
```

```
2. import scala.io.StdIn._
   object StringConcat {
       def main(args: Array[String]): Unit = {
           print("Please enter the first word: ")
           val firstWord = readLine()
           print("Please enter the second word: ")
           val secondWord = readLine()
           val combination = firstWord.toString.concat(
               secondWord.toString)
           println(combination)
           val wordCombReversed = combination.reverse
           println(wordCombReversed)
       }
   }

3. import scala.io.StdIn._
   object AreaCalculation {
       def main(args: Array[String]): Unit = {
           val area = {
               print("Please enter the length: ")
               val length = readDouble()
               print("Please enter the breadth: ")
               val breadth = readDouble()
               length * breadth
           }
           println(area)
       }
   }
```



<http://www.springer.com/978-3-319-69367-5>

Programming with Scala

Language Exploration

Upadhyaya, B.

2017, XIX, 194 p. 67 illus., Softcover

ISBN: 978-3-319-69367-5