The Scala Interpreter

To start the Scala interpreter:

1. Install Scala.
2. Make sure that the scala/bin directory is on the PATH.
3. Open a command shell in your operating system.
4. Type scala followed by the Enter key.

Type commands followed by Enter.
Each time, the interpreter displays the answer
For example,

```
scala> 8 * 5 + 2
res0: Int = 42
```

The answer is given the name res0.

You can use that name in subsequent computations:

```
scala> 0.5 * res0
res1: Double = 21.0
scala> "Hello, " + res0
res2: java.lang.String = Hello, 42
```

As you can see, the interpreter also displays the type of the result—in our examples, Int, Double, and java.lang.String.

The Scala interpreter reads an expression, evaluates it, prints it, and reads the next expression.

This is called the *read-eval-print loop*, or REPL.

Declaring Values and Variables

Instead of using res0, res1, and so on, you can define your own names:

scala> **val answer = 8 * 5 + 2**

answer: Int = 42

You can use these names in subsequent expressions:

scala> 0.5 * answer

res3: Double = 21.0

A value declared with val is actually a constant—you can't change its contents:

scala> **answer = 0**

<console>:6: error: reassignment to val

To declare a variable whose contents can vary, use a var:

var counter = 0

counter = 1 // OK, can change a var

You **need not specify the type of a value or variable**.

It is **inferred from the type of the expression with which you initialize it.**

It is an **error to declare a value or variable without initializing it.**

However, you can **specify the type if necessary**. For example,

```
val greeting: String = null
val greeting: Any = "Hello"
```

You can **declare multiple values or variables together:**

```
val xmax, ymax = 100 // Sets xmax and ymax to 100
var greeting, message: String = null
```

Commonly Used Types

Like Java, Scala has **seven numeric types: Byte, Char, Short, Int, Long, float, and Double, and a Boolean type.**

However, unlike Java, **these types are *classes***.

There **is no distinction between primitive types and class types in Scala.**

You can **invoke methods on numbers**, for example:

```
1.toString() // Yields the string "1"
or, more excitingly,
1.to(10) // Yields Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Scala **relies on the underlying java.lang.String** class for strings.

However, it **augments that class with well over a hundred operations in the StringOps class.**

For example:
"Hello".intersect("World") // Yields "lo“

In this expression, **the java.lang.String object "Hello" is implicitly converted to a StringOps object, and then the intersect method of the StringOps class is applied**

Similarly, **there are classes RichInt, RichDouble, RichChar, and so on.**

Each of them has **a small set of convenience methods for acting on their poor cousins—Int, Double,or Char.**

The **to method is actually a method of the RichInt class.**
In the expression
1.to(10)
the **Int value 1 is first converted to a RichInt, and the to method is applied to that value.**

Finally, **there are classes BigInt and BigDecimal for computations with an arbitrary (but finite) number of digits.**

Arithmetic and Operator Overloading

Arithmetic operators in Scala work just as you would expect in Java or C++:

```
val answer = 8 * 5 + 2
```

The + - * / % operators do their usual job, as do the bit operators & | ^ >> <<.

There is just one surprising aspect: These operators are actually methods
a + b
is a shorthand for
a.+(b)

In general, you can write

a *method* b as a shorthand for a.*method*(b)
where *method* is a method with two parameters

For example, instead of
1.to(10)
you can write
1 to 10

More about Calling Methods

You have already seen how to call methods on objects, such as "Hello".intersect("World")

If the **method has no parameters, you don't have to use parentheses**.

For example, the **API of the StringOps class shows a method sorted, without (), which yields a new string with the letters in sorted order**.

Call it as "Bonjour".sorted // Yields the string "Bjnooru"

```
import scala.math._ // In Scala, the _ character is a “wildcard,” like * in Java

sqrt(2) // Yields 1.4142135623730951
pow(2, 4) // Yields 16.0
min(3, Pi) // Yields 3.0
```

If you don’t import the scala.math package, add the package name:

```
scala.math.sqrt(2)
```

The **apply** Method

In Scala, it is **common to use a syntax that looks like a function call**.

For example, **if s is a string, then s(i) is the ith character of the string**. (In C++, you would write s[i]; in Java, s.charAt(i).)

Try it out in the REPL:

val s = "Hello"

s(4) // Yields 'o'

You can think of **this as an overloaded form of the () operator.**

It is **implemented as a method with the name apply**. For example, in the **documentation of the StringOps class, you will find a method**

def apply(n: Int): Char

That is, s(4) is a shortcut for s.apply(4)

Control Structures and Functions

Conditional Expressions

Scala has an **if/else construct with the same syntax as in Java** or C++.

However, **in Scala, an if/else has a value, namely the value of the expression that follows the if or else.** For example,
if (x > 0) 1 else -1
has a value of 1 or -1, depending on the value of x.

You can put that value in a variable:
val s = if (x > 0) 1 else -1

This has the same effect as
if (x > 0) s = 1 else s = -1
However, **the first form is better because it can be used to initialize a val. In the second form, s needs to be a var**.

Java and C++ have a ?: operator for this purpose.

The expression
x > 0 ? 1 : -1 // Java or C++
is **equivalent to the Scala expression if (x > 0) 1 else -1.**

The **Scala if/else combines the if/else and ?:constructs that are separate in Java and C++.**

Any

In Scala, **every expression has a type.**

For example, the expression **if (x > 0) 1 else -1 has the type Int because both branches have the type Int.**

The type of a **mixed-type expression**, such as
if (x > 0) "positive" else -1
is the **common supertype of both branches**.

In this example, one branch is ajava.lang.String, and the other an Int. Their **common supertype is called Any**.

If the **else part is omitted, for example in**
if (x > 0) 1
then it is possible that the if statement yields no value.

However, in Scala, **every expression is supposed to have *some* value.**

This is finessed by **introducing a class Unit that has one value, written as ().**

The **if statement without an else is equivalent to**
if (x > 0) 1 else ()

Think of **() as a placeholder for "no useful value," and of Unit as an analog of void in Java**

CAUTION:

The **REPL is more nearsighted than the compiler**—it only sees one line of code at a time.
For example, when you type
if (x > 0) 1
else if (x == 0) 0 else -1
the REPL executes if (x > 0) 1 and shows the answer.

Then it gets confused about the else keyword.

If you want to break the line before the else, use braces:
if (x > 0) { 1
} else if (x == 0) 0 else -1

Statement Termination

1. In **Java** and C++, every statement **ends with a semicolon**.

In **Scala**—a **semicolon is never required if it falls just before the end of the line.**

A semicolon is also **optional before**
An },
an else, and
similar locations where it is clear from context that the end of a statement has been reached.

However, **if you want to have more than one statement on a single line, you need to separate them with semicolons. For example,**
if (n > 0) { r = r * n; n -= 1 }

If you **want to continue a long statement over two lines**, make sure that **the first line ends in a symbol that *cannot be* the end of a statement.**

An operator is often a good choice:

```
s = s0 + (v - v0) * t + // The + tells the parser that this is not the end
0.5 * (a - a0) * t * t
```

Another example: **) { r = r * n; n -= 1 } could be written as**

```
if (n > 0) {
r = r * n
n -= 1
}
```

Block Expressions and Assignments

In **Java** or C++, **a block statement is a sequence of statements enclosed in { }.**

In **Scala, a { } block contains a sequence of *expressions*, and the result is also an expression.**

The **value of the block is the value of the last expression**.

This feature can be useful if the initialization of a val takes more than one step.
For example,
val distance = { val dx = x - x0; val dy = y - y0; **sqrt(dx * dx + dy * dy)** }

In Scala, **assignments have no value—or, strictly speaking, they have a value of type Unit.**

A block that ends with an assignment, such as
{ r = r * n; n -= 1 }
has a Unit value.

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

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

Input and Output

To print a value, **use the print or println function**.
The latter **adds a newline character after the printout.**

For example,
print("Answer: ")
println(42)
yields the same output as
println("Answer: " + 42)

There is also a printf function with a C-style format string:
printf("Hello, %s! You are %d years old.%n", name, age)

Or better, use ***string interpolation***

print(**f**"Hello, **$name**! In six months, you'll be **${age + 0.5}%7.2f** years old.%n")

A **formatted string is prefixed with the letter f**.

The **expression $name is replaced with the value of the variable name. The expression ${age + 0.5}%7.2f**
is replaced with the value of age + 0.5, formatted as a floating-point number of width 7 and precision 2.

You can **read a line of input from the console with the readLine method** of the scala.io.StdIn class.

```
import scala.io
val name = StdIn.readLine("Your name: ")
print("Your age: ")
val age = StdIn.readInt()
println(s"Hello, ${name}! Next year, you will be ${age + 1}.")
```

Loops

Scala has the **same while and do loops as Java** and C++.

Scala has **no direct analog of the for (*initialize*; *test*; *update*) loop. If you need such a loop, you have two choices**.

1. You can use a while loop.
2. Or, you can use a for statement like this:

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

The **call 1 to n returns a Range of the numbers from 1 to n** (inclusive).

The construct
for (i <- *expr*) makes the variable i traverse all values of the expression to the right of the <-.

When traversing a string, you can loop over the index values:

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

Advanced **for** Loops

This section covers the advanced features.

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

For example,

```
for (i <- 1 to 3; j <- 1 to 3) print(f"${10 * i + j}%3d")
// Prints 11 12 13 21 22 23 31 32 33
```

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

```
for (i <- 1 to 3; j <- 1 to 3 if i != j) print(f"${10 * i + j}%3d")
// Prints 12 13 21 23 31 32
```

Note that there is **no semicolon** before the if.

You can have any **number of *definitions***, introducing **variables that can be used inside the loop:**

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

When the **body of the for loop starts with yield**, the loop **constructs a collection of values, one for each iteration:**

```
for (i <- 1 to 10) yield i % 3
// Yields Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

This type of **loop is called a for *comprehension***.

The generated collection is compatible with the first generator.

```
for (c <- "Hello"; i <- 0 to 1) yield (c + i).toChar
// Yields "HIeflmlmop“
```

```
for (i <- 0 to 1; c <- "Hello") yield (c + i).toChar
// Yields Vector('H', 'e', 'l', 'l', 'o', 'I', 'f', 'm', 'm', 'p')
```

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

```
for { i <- 1 to 3
from = 4 - i
j <- from to 3 }
```

Functions

Scala has functions in addition to methods. A method operates on an object, but a function doesn't.

To **define a function, specify the function's name, parameters, and body like this:**

```
def abs(x: Double) = if (x >= 0) x else -x
```

You **must specify the types of all parameters.**

However, **as long as the function is not recursive, you need not specify the return type.**

If the body of the function requires more than one expression, use a block.

The **last expression of the block becomes the value that the function returns.**

For example, the following function returns the value of r after the for loop.

```
def fac(n : Int) = {
var r = 1
for (i <- 1 to n) r = r * i
r
}
```

There **is no need for the return keyword in this example**.

With a **recursive function**, you **must specify the return type**. For example,

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

Default and Named Arguments

1.You can provide **default arguments for functions that are used when you don’t specify explicit values.**

For example,
def decorate(str: String, left: String = "[", right: String = "]") =
left + str + right
This function **has two parameters, left and right, with default arguments "[“ and "]".**

If you call **decorate("Hello"), you get "[Hello]".**

2. If you don’t like the defaults, supply your own:
decorate("Hello", "<<<", ">>>").

3. If you **supply fewer arguments than there are parameters, the defaults are applied from the end.**

For example,
decorate("Hello", ">>>[")
uses the **default value of the right parameter, yielding ">>>[Hello]".**

4. You can also **specify the parameter names when you supply the arguments.**

For example,
decorate(left = "<<<", str = "Hello", right = ">>>")
The result is "<<<Hello>>>". Note that the named arguments need not be in the same order as the parameters.

Variable Arguments

Sometimes, it is convenient to **implement a function that can take a variable number of arguments**.

The following example shows the syntax:

```
def sum(args: Int*) = {
var result = 0
for (arg <- args) result += arg
result
}
```

You can **call this function with as many arguments as you like.**

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

The **function receives a single parameter of type Seq**

If **you already have a sequence of values, you cannot pass it directly to such a function.**

For example, the following is not correct:
val s = sum(1 to 5) // Error

The **remedy is to tell the compiler that you want the parameter to be considered an argument sequence**. Append : _*, like this:

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

Procedures

Scala has a **special notation for a function that returns no value**.

If the **function body is enclosed in braces *without a preceding* = *symbol*, then the return type is Unit.**

Such a function is called a *procedure*.

For example, the following procedure prints a string inside a box, like

```
-------
|Hello|
-------
```

Since the **procedure doesn't return any value, we omit the** = symbol

```
def box(s : String) {
val border = "-" * (s.length + 2)
print(f"$border%n|$s|%n$border%n")  }
```

Lazy Values

When a **val is declared as lazy, its initialization is deferred until it is accessed for the first time.**

For example,
lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString

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

To verify this, **try it out in the REPL, but misspell the file name**.

There **will be no error when the initialization statement is executed**.

However**, if you access words, you will get an error message that the file is not found.**

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

Compare

val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
// **Evaluated as soon as words is defined**

lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
// **Evaluated the first time words is used**

def words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
// Evaluated every time words is used

Exceptions

Scala exceptions **work the same way as in Java or C++.**

When you throw an exception, for example

throw new IllegalArgumentException("x should not be negative")

the **current computation is aborted, and the runtime system looks for an exception handler that can accept an IllegalArgumentException**.

Control resumes with the innermost such handler.

If **no such handler exists, the program terminates**

A **throw expression has the special type Nothing**.

That is useful in if/else expressions.

If **one branch has type Nothing, the type of the if/else expression is the type of the other branch.**

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

The **first branch has type Double, the second has type Nothing.**

Therefore, the if/else expression also has type Double.

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

For example:

```
val in = new URL("http://horstmann.com/fred.gif").openStream()
try {
process(in)
} finally {
in.close()
}
```

The **finally clause is executed whether or not the process function throws an exception. The reader is always closed**

Note that **try/catch and try/finally have complementary goals**.

The **try/catch statement handles exceptions, and the try/finally statement takes some action when an exception is not handled.**

You can **combine them into a single try/catch/finally statement:**

try { ... } catch { ... } finally { ... }
This is the same as
try { try { ... } catch { ... } } finally { ... }

Exercise 1:

The signum of a number is 1 if the number is positive, –1 if it is negative, and 0 if it is zero. Write a function that computes this value.

```
def signum1(x: Int) = if (x == 0) 0 else if (x < 0) -1 else 1
```

Exercise 2:

What is the value of an empty block expression {}? What is its type?

An empty block as type Unit

Unit is a subtype of scala.AnyVal.

There is **only one value of type Unit, (), and it is not represented by any object in the underlying runtime system**.

A method with return type Unit is analogous to a Java method which is declared void.

So, **Unit is a sort of placeholder meaning no useful value**.

Exercise 3:

Write a Scala equivalent for the Java loop

```
for (int i = 10; i >= 0; i--) System.out.println(i);
```

```
for (i <- 0 to 10 reverse) println(i)
```

(OR)

Using by to control the increment:

```
for (i <- 10 to 0 by -1) println(i)
```

Exercise 4:

Write a procedure countdown(n: Int) that prints the numbers from n to 0.

```
def countdown1(x: Int){
  for (i <- x to 0 by -1) println(i)}
```

Reference-Scala For The Impatient -- Chapter 2 (derlin.github.io)

Fixed-Length Arrays

If you need an array whose length doesn’t change, use the Array type in Scala.

For example,

```
val nums = new Array[Int](10)
// An array of ten integers, all initialized with zero

val a = new Array[String](10)
// A string array with ten elements, all initialized with null

val s = Array("Hello", "World")
// An Array[String] of length 2—the type is inferred
// Note: No new when you supply initial values
```

Variable-Length Arrays: Array Buffers

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
// Or new ArrayBuffer[Int]
// An empty array buffer, ready to hold integers

b += 1
// ArrayBuffer(1)

// Add an element at the end with +=
b += (1, 2, 3, 5)
// ArrayBuffer(1, 1, 2, 3, 5)

// Add multiple elements at the end by enclosing them in parentheses
b ++= Array(8, 13, 21)
// ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
// You can append any collection with the ++= operator
```

```
b.trimEnd(5)
// ArrayBuffer(1, 1, 2)
// Removes the last five elements
```

You can also insert and remove elements at an arbitrary location,
For example:

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

b.insert(2, **7, 8, 9**)
// ArrayBuffer(1, 1, **7, 8, 9**, 6, 2)
// You can insert as many elements as you like

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

b.remove(2, 3)
// ArrayBuffer(1, 1, 2)
// The second parameter tells how many elements to remove

Traversing Arrays and Array Buffers

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

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

The until method is similar to the to method, except that it excludes the last value.

Therefore, the variable i goes from 0 to a.length – 1

To visit every second element, let i traverse

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

Transforming Arrays

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

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

Oftentimes, when you **traverse a collection, you only want to process the elements that match a particular condition.**

This is achieved with a guard: an if inside the for.

Here we double every even element, dropping the odd ones:

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

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

Suppose we want to **remove all negative elements from an array buffer of integers**

```
val result = for (elem <- a if elem >= 0) yield elem
```

Suppose that **we want to modify the original array buffer instead, removing the unwanted elements**.

Then **we can collect their positions:**

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

Now remove the elements at those positions, starting from the back:

```
for (i <- positionsToRemove.reverse) a.remove(i)
```

Common Algorithms

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

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

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

4 the mkString
method lets you specify the separator between elements

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

Multidimensional Arrays

To construct such an array, use the ofDim method:

val **matrix = Array.ofDim[Double](3, 4)** // Three rows, four columns

To access an element, use two pairs of parentheses:

matrix(row)(column) = 42

Constructing a Map

You can construct a map as
val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)

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

If you want a mutable map, use
val scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)

If you want to start out with a blank map, you have to supply type parameters:
val scores = scala.collection.mutable.Map[String, Int]()

You could have equally well defined the map as
val scores = Map(("Alice", 10), ("Bob", 3), ("Cindy", 8))
The -> operator is just a little easier on the eyes than the parentheses

Accessing Map Values

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

If the map doesn’t contain a value for the requested key, an exception is thrown.

To check whether there is a key with the given value, call the contains method:

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

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

```
val bobsScore = scores.getOrElse("Bob", 0)
```

Updating Map Values

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

- scores("Bob") = 10

// Updates the existing value for the key "Bob" (assuming scores is mutable)

- scores("Fred") = 7

// Adds a new key/value pair to scores (assuming it is mutable)

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

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

.

To **remove a key and its associated value**, use the -= operator:
scores -= "Alice“

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

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

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

Iterating over Maps

The following amazingly **simple loop iterates over all key/value** pairs of a map:
for ((k, v) <- map) process k and v

If for some reason **you want to visit only the keys or values, use the keySet and values methods,**
scores.keySet // A set such as Set("Bob", "Cindy", "Fred", "Alice")

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

To **reverse a map—that** is, switch keys and values—use
for ((k, v) <- map) yield (v, k)

Sorted Maps

There are **two common implementation strategies** for maps:
hash tables and balanced trees.

Hash tables use the hash codes of the keys to scramble entries,
so iterating over the elements yields them in unpredictable order

If you **need to visit the keys in sorted order, use a SortedMap instead.**

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

Tuples

Maps are collections of key/value pairs.

Pairs are the simplest case of tuples

A tuple value is formed by enclosing individual values in parentheses.
For example,

(1, 3.14, "Fred") is a tuple of type Tuple3[**Int, Double, java.lang.String**]
which is also written as
(Int, Double, java.lang.String)

If you have a tuple, say,

val t = (1, 3.14, "Fred")

then you can **access its components with the methods _1, _2, _3**, for example:

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

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

Tuples are **useful for functions that return more than one value**.

For example,
the **partition method of the StringOps class returns a pair of strings, containing the characters that fulfill a condition and those that don’t**:

"New York".partition(_.isUpper) // Yields the pair ("NY", "ew ork")

Zipping

One reason for **using tuples is to bundle together values so that they can be processed together.**

This is commonly done with the zip method. For example,
the code

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

yields an array of pairs
Array(("<", 2), ("-", 10), (">", 2))

The pairs can then be processed together:

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