Data Types

1. Int
2. Byte
3. Short
4. Long
5. Float
6. Double
7. Boolean(true, false)
8. Char
9. String

```
val b: Byte = 1
val x: Int = 1
val l: Long = 1
val s: Short = 1
val d: Double = 2.0
val f: Float = 3.0
```

Declaration and Variables
Method 1: val x = 9
Method 2: var y = 1
Use var only if you are sure you will need to re-assign something later.

Arrays

Arrays are instantiated using the Array[T](a, b, c) syntax, and entries within each array are retrieved using a(n)
var z:Array[String] = new Array[String](3)
or
var z = new Array[String](3)

```
@ val a = Array[Int](1, 2, 3, 4)

@ a(0) // first entry, array indices start from 0
res36: Int = 1

@ a(3) // last entry
res37: Int = 4

@ val a2 = Array[String]("one", "two", "three", "four")
a2: Array[String] = Array("one", "two", "three", "four")

@ a2(1) // second entry
res39: String = "two"
```
</> 3.20.scala

Create Array with Range

Use of range() method to generate an array containing a sequence of increasing integers in a given range. You can use the final argument as a step to create the sequence; if you do not use the final argument, then step would be assumed as 1.

```
import Array._

object Demo {
   def main(args: Array[String]) {
      var myList1 = range(10, 20, 2)
      var myList2 = range(10,20)

      // Print all the array elements
      for ( x <- myList1 ) {
         print( " " + x )
      }

      println()
      for ( x <- myList2 ) {
         print( " " + x )
      }
   }
}
```

Loops, Conditionals, Comprehensions

```
@ var total = 0

@ val items = Array(1, 10, 100, 1000)

@ for (item <- items) total += item

@ total
res65: Int = 1111
```

</> 3.28.scala

```
@ var total = 0

@ for (i <- Range(0, 5)) {
    println("Looping " + i)
    total = total + i
  }
Looping 0
Looping 1
Looping 2
Looping 3
Looping 4

@ total
res68: Int = 10
```

</> 3.29.scala

You can loop over nested `Array`s by placing multiple `<-`s in the header of the loop:

```
@ val multi = Array(Array(1, 2, 3), Array(4, 5, 6))

@ for (arr <- multi; i <- arr) println(i)
1
2
3
4
5
6
```

</> 3.30.scala

Condition(if-else):

```
@ var total = 0

@ for (i <- Range(0, 10)) {
    if (i % 2 == 0) total += i
    else total += 2
  }

@ total
res74: Int = 30
```

</> 3.32.scala

```
@ var total = 0

@ for (i <- Range(0, 10)) {
    total += (if (i % 2 == 0) i else 2)
  }

@ total
res77: Int = 30
```

</> 3.33.scala

Methods and Functions

You can define methods using the `def` keyword.
Passing in the wrong type of argument, or missing required arguments, is a compiler error. However, if the argument has a default value, then passing it is optional.

```
@ printHello("1") // wrong type of argument
cmd128.sc:1: type mismatch;
 found   : String("1")
 required: Int
val res128 = printHello("1")
                        ^
Compilation Failed
```
</> 3.42.scala

```
@ def printHello2(times: Int = 0) = {
    println("hello " + times)
  }

@ printHello2(1)
hello 1

@ printHello2()
hello 0
```
</> 3.43.scala

Apart from performing actions like printing, methods can also return values. The last expression within the curly brace {} block is treated as the return value of a Scala method.

```
@ def hello(i: Int = 0) = {
    "hello " + i
  }
```

You can define function values using the => syntax.

```
@ var g: Int => Int = i => i + 1
```

Classes

You can define classes using the class keyword, and instantiate them using new.

```
@ class Foo(x: Int) {
    def printMsg(msg: String) = {
      println(msg + x)
    }
  }
```
</> 3.53.scala

```
@ val f = new Foo(1)

@ f.printMsg("hello")
hello1

@ f.x
cmd120.sc:1: value x is not a member of Foo
Compilation Failed
```
</> 3.54.scala

Reference:
https://docs.scala-lang.org/tour/basics.html
https://www.handsonscala.com/chapter-3-basic-scala.html
https://www.tutorialspoint.com/scala/index.htm