Introduction to Scala in Spark

Kazi Aminul Islam

Department of Computer Science

Kennesaw State University

What is Scala?

- Scala stands for "scalable language.“
- Both object-oriented and functional
- Aim to address criticisms of Java
- To be concise!
- Designed by Martin Odersky (2004), a German computer scientist and professor of programming methods at École Polytechnique Fédérale de Lausanne in Switzerland.
- Martin’s Ph.D. advisor, Niklaus Wirth, developed Pascal.

Scala Overview

- It’s a high-level language.
- It’s statically typed.
- Its syntax is concise but still readable — we call it *expressive.*
- It supports the object-oriented programming (OOP) paradigm.
- It supports the functional programming (FP) paradigm.
- It has a sophisticated type inference system.
- Scala code results in *.class* files that run on the Java Virtual Machine (JVM).
- It’s easy to use Java libraries in Scala.

Two types of variables

- val is an immutable variable — like final in Java — and should be preferred
- var creates a mutable variable, and should only be used when there is a specific reason to use it
- Examples:
    - val x = 1   //immutable
    - var y = 0   //mutable

Common Data Types

Byte 8 bit signed value. Range from -128 to 127
Short 16 bit signed value. Range -32768 to 32767
Int 32 bit signed value. Range -2147483648 to 2147483647
Long 64 bit signed value. -9223372036854775808 to 9223372036854775807
Float 32 bit IEEE 754 single-precision float
Double 64 bit IEEE 754 double-precision float
Char 16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
String A sequence of Chars
Boolean Either the literal true or the literal false
Unit Corresponds to no value

Type Inference

- **Scala will infer a variable type.**
- **val** x = 1
- **val** s = "a string“
- **val** f = 3.14f
- **val** df = 3.14
- **val** p = **new Person**("Regina")

Declaring variable types

- **val** x: **Int** = 1
- **val** s: **String** = "a string"
- **val** f: **Float** = 3.14f
- **val** df: **Double** = 3.14
- **val** p: **Person** = **new Person**("Regina")

if/else Decision Structure

```
if (test1) {
    doA()
} else if (test2) {
    doB()
} else if (test3) {
    doC()
} else {
    doD()
}
```

If/else returns a value:

```
val x = if (a < b) a else b
```

Switch (Match)

```
val result = i match {
    case 1 => "one"
    case 2 => "two"
    case _ => "not 1 or 2"
}
```

For loop and expression

- for (arg <- args) println(arg)
- // "x to y" syntax
- for (i <- 0 to 5) println(i)
- // "x to y by" syntax
- for (i <- 0 to 10 by 2) println(i)
- // yield expression; create a vector
- val x = for (i <- 1 to 5) yield i * 2

- // yield; create a List (5, 6, 6)
- val fruits = List("apple", "banana", "lime", "orange")
- val fruitLengths = for {
-     f <- fruits
-     if f.length > 4
- } yield f.length

While and do-while

```
// while loop
while(condition) {
    statement(a)
    statement(b)
}
```

```
// do-while
do {
    statement(a)
    statement(b)
}
while(condition)
```

Classes

- No need to create “get” and “set” methods to access the fields in the class.

```
class Person(var firstName: String, var lastName: String) {
    def printFullName() = println(s"$firstName $lastName")
}
// This is how you use that class:
val p = new Person("Julia", "Kern")
println(p.firstName)
p.lastName = "Manes"
p.printFullName()
```

Scala Methods

- With return type:

```
def sum(a: Int, b: Int): Int = a + b
def concatenate(s1: String, s2: String): String = s1 + s2
```

- Without return type:

```
def sum(a: Int, b: Int) = a + b
def concatenate(s1: String, s2: String) = s1 + s2
```

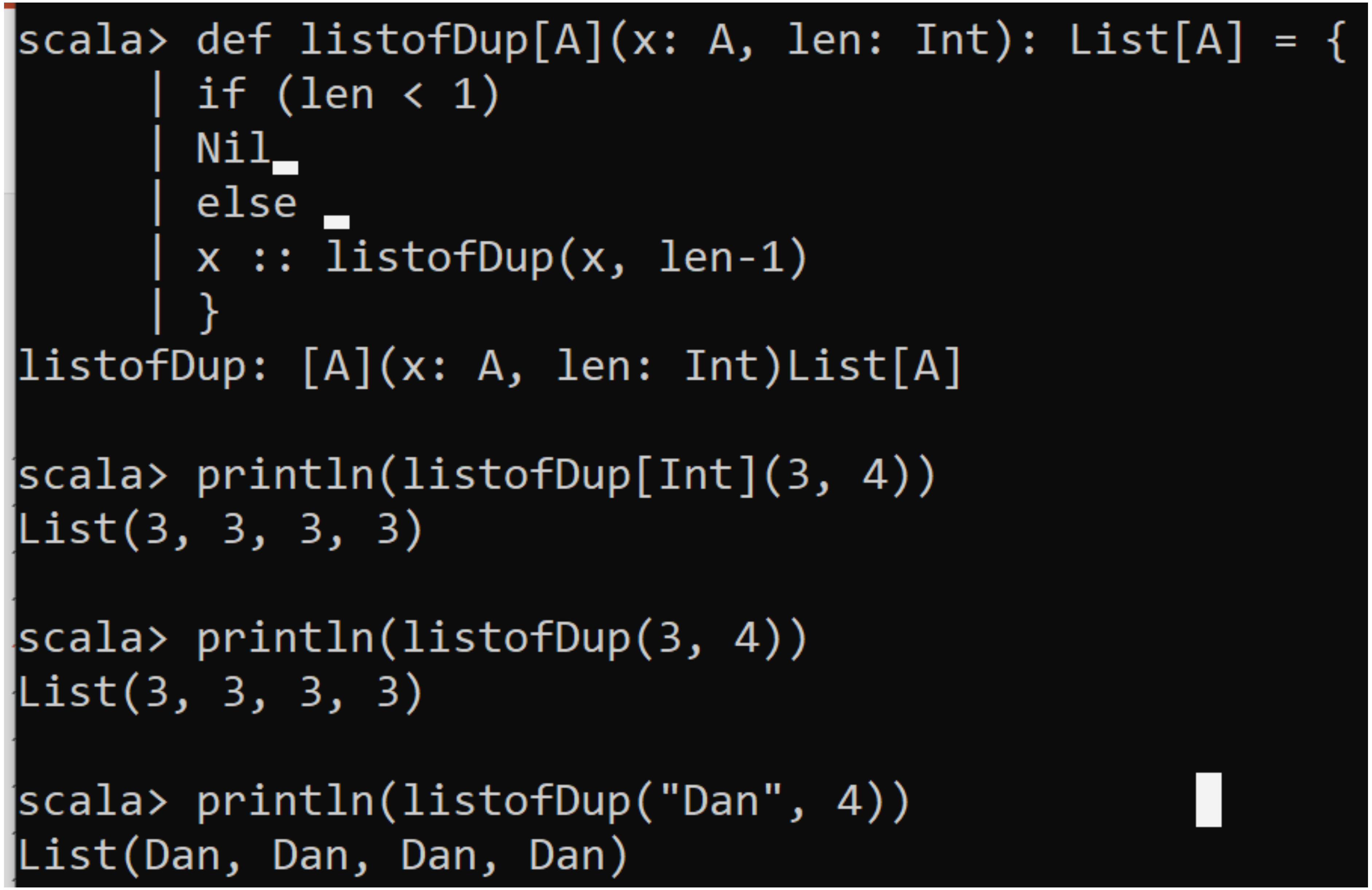
- How you call the methods

```
val x = sum(1, 2)
val y = concatenate("foo", "bar")
```

Polymorphic Methods

- Methods in Scala can be parameterized by type as well as value. The syntax is similar to that of generic classes.
- Type parameters are enclosed in square brackets, while value parameters are enclosed in parentheses.

Example

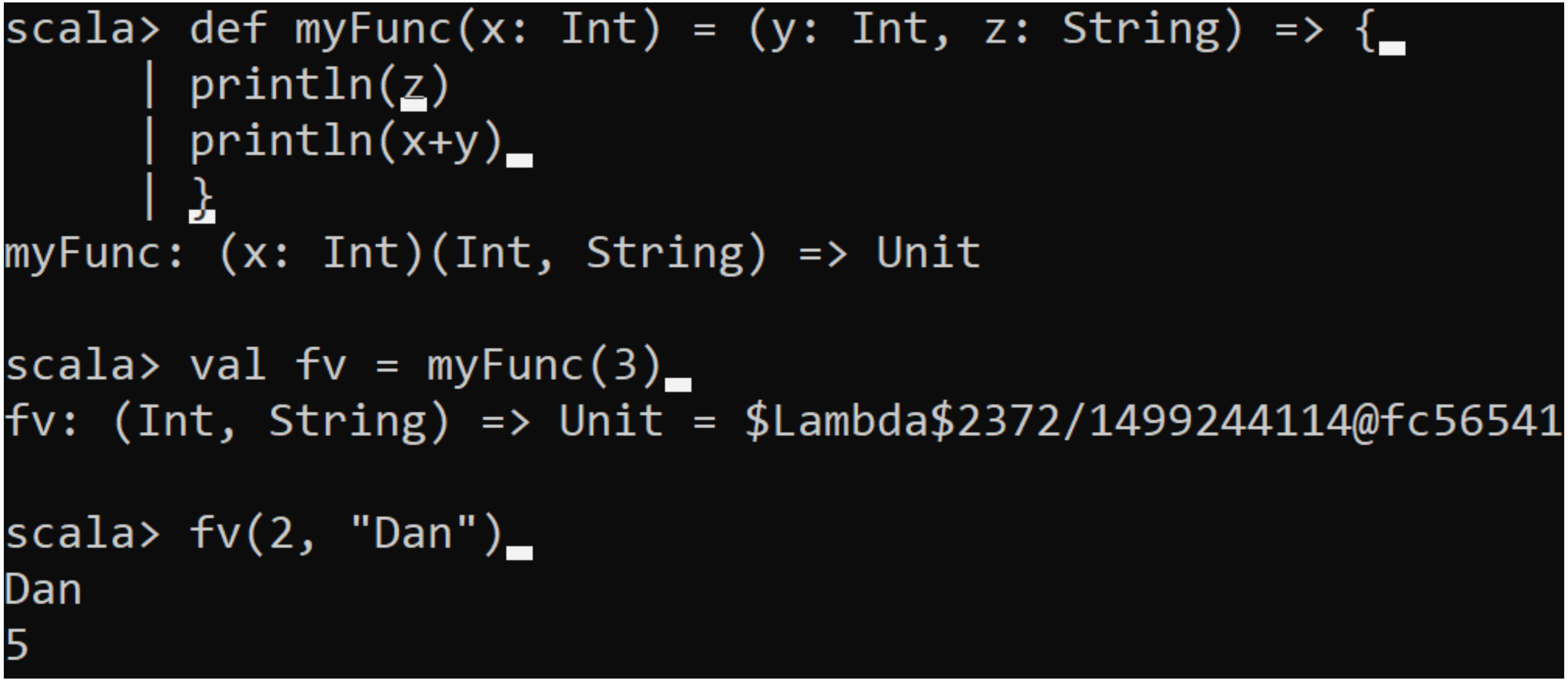


```
scala> def listofDup[A](x: A, len: Int): List[A] = {
     | if (len < 1)
     | Nil
     | else
     | x :: listofDup(x, len-1)
     | }
listofDup: [A](x: A, len: Int)List[A]

scala> println(listofDup[Int](3, 4))
List(3, 3, 3, 3)

scala> println(listofDup(3, 4))
List(3, 3, 3, 3)

scala> println(listofDup("Dan", 4))
List(Dan, Dan, Dan, Dan)
```

Higher Order Function

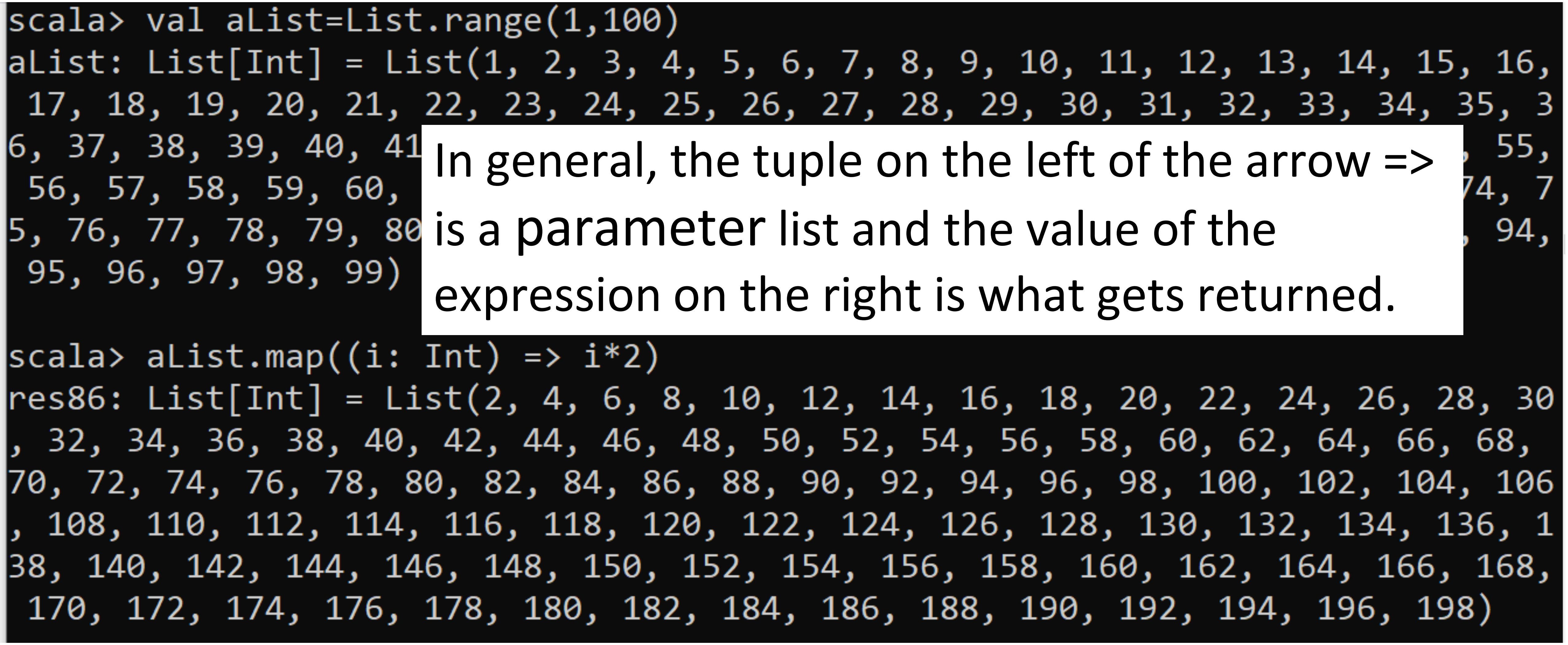
- Higher order functions take other functions as parameters or return a function as a result.
- **val** salaries = **Seq**(20000, 70000, 40000)
- **def** doubleSalary(x: **Int**): Int = x * 2
- **Val** newSalaries = salaries.map(doubleSalary) *// List(40000, 140000, 80000)*
- doubleSalary is a function which takes a single Int, x, and returns x * 2.

Function that Returns a Function



```
scala> def myFunc(x: Int) = (y: Int, z: String) => {
     | println(z)
     | println(x+y)
     | }
myFunc: (x: Int)(Int, String) => Unit

scala> val fv = myFunc(3)
fv: (Int, String) => Unit = $Lambda$2372/1499244114@fc56541

scala> fv(2, "Dan")
Dan
5
```

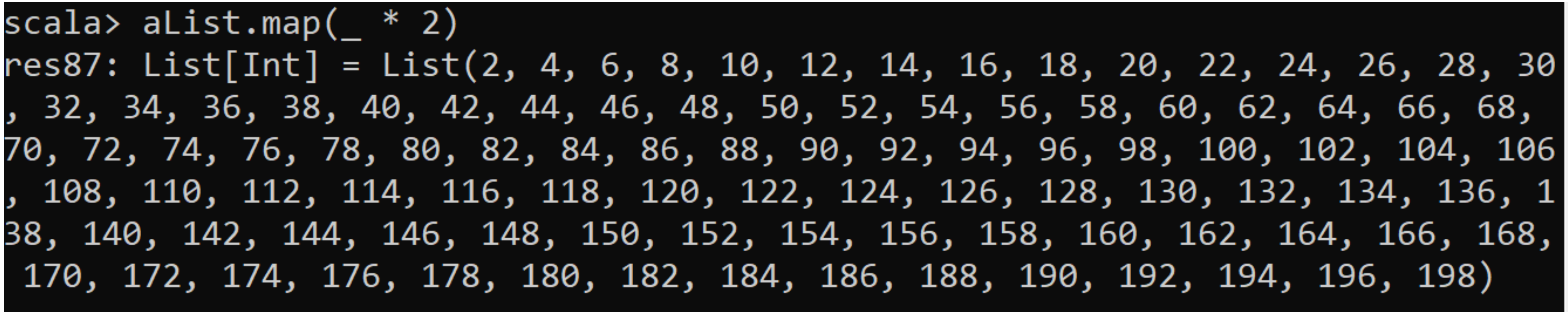
Anonymous Function



```
scala> val aList=List.range(1,100)
aList: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 3
6, 37, 38, 39, 40, 41                                                    , 55,
 56, 57, 58, 59, 60,                                                     74, 7
5, 76, 77, 78, 79, 80                                                    , 94,
 95, 96, 97, 98, 99)

scala> aList.map((i: Int) => i*2)
res86: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30
, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106
, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 1
38, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168,
 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198)
```

In general, the tuple on the left of the arrow => is a parameter list and the value of the expression on the right is what gets returned.

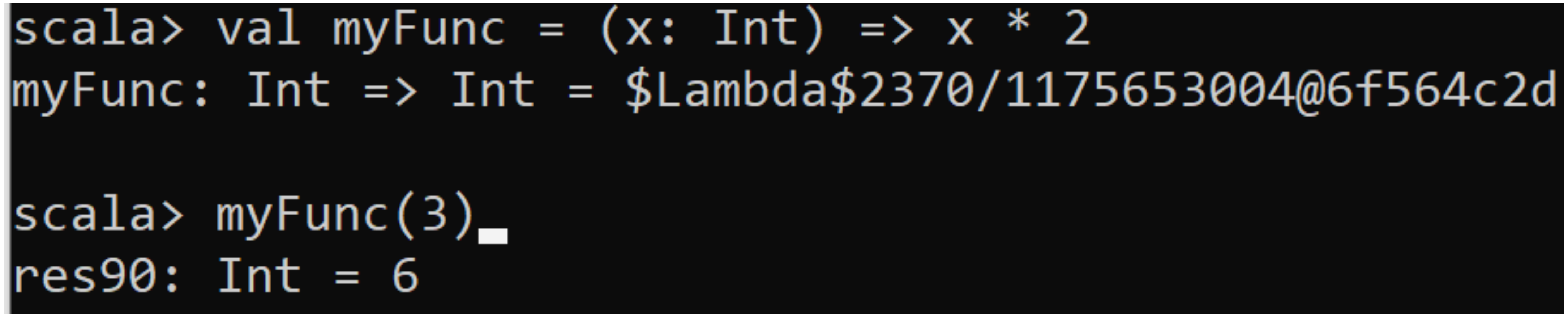
Anonymous Function (cont.)


```
scala> aList.map(_ * 2)
res87: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30
, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106
, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 1
38, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168,
 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198)
```

Underscore in this context can only be used once in the function body.
For example, to compute square, _ * _ won’t work. Instead, use Math.pow(_, 2).

Use Functions as Variables

- Function in Scala is a first class value.
- So you may treat them as if they were variables.



```
scala> val myFunc = (x: Int) => x * 2
myFunc: Int => Int = $Lambda$2370/1175653004@6f564c2d

scala> myFunc(3)
res90: Int = 6
```

Common Mistakes in Functions

- Functions (procedures) that return no values

```
def myFunc(x: Int): Int {
        return (x*x)
}
```

Missing =

Correct way: def myFunc(x: Int) = x*x

- Functions (procedures) that return no values

```
def myProc(x: Int) {
        println(x*x)
}
```

Collection classes

- Basic Scala collection classes: List, ListBuffer, Vector, ArrayBuffer, Map, and Set
- Populating lists
  - val nums = List.range(0, 10)
  - val nums = (1 to 10 by 2).toList
  - val letters = ('a' to 'f').toList
  - val letters = ('a' to 'f' by 2).toList
- Sequence methods
  - val nums = (1 to 10).toList // create a list
  - nums.foreach(println) // foreach()
  - nums.filter(_ < 4).foreach(println) // filter()
  - val doubles = nums.map(_ * 2) // map()
  - nums.foldLeft(0)(_ + _) // foldLeft(0) sum them together
  - nums.foldLeft(1)(_ * _) // foldLeft(1) multiply them together

Tuples

- Tuples let you put a heterogeneous collection of elements in a little container. Tuples can contain between two and 22 values, and they can all be different types.
- class Person(var name: String)
- val t = (11, "Eleven", new Person("Eleven"))
- Reference tuple field by t._1, t._2, and t._3, respectively.
- Assign tuple fields to variables
  val (num, string, person) = (11, "Eleven", new Person("Eleven"))

Case Class

- A Case Class is just like a regular class, which has a feature for modeling unchangeable data.
- A case object has some more features than a regular object, such as serializable, ProductArity, and hashCode().

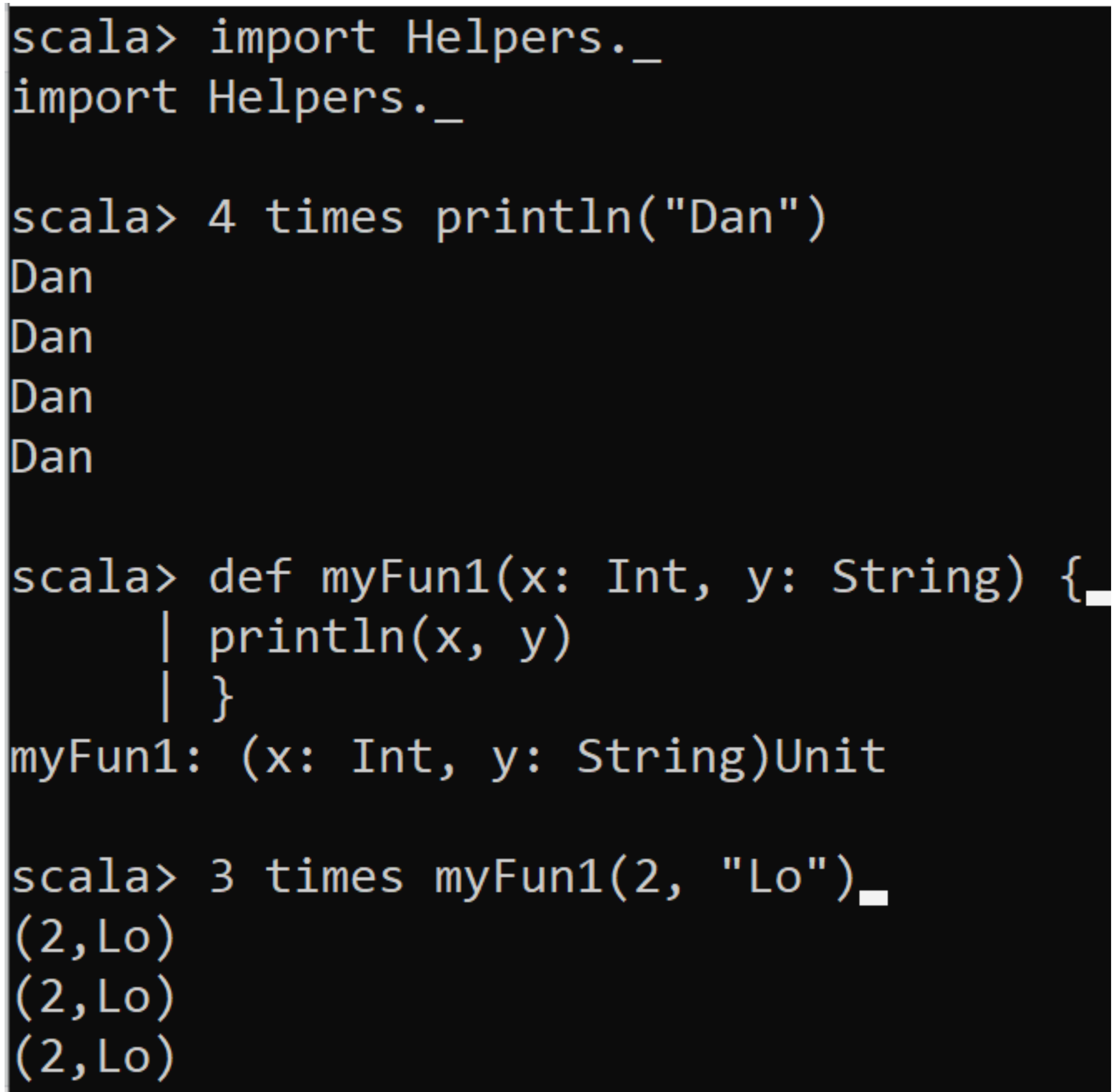
```
case class employee (name:String, age:Int)
object Main
{
    // Main method
    def main(args: Array[String])
    {
        var c = employee("Nidhi", 23)

        // Display both Parameter
        println("Name of the employee is " + c.name);
        println("Age of the employee is " + c.age);
    }
}
```

Implicit Class

- An implicit class is a class marked with the implicit keyword. This keyword makes the class's primary constructor available for implicit conversions when the class is in scope.
- They must be defined inside of another trait/class/object.
- They may only take one non-implicit argument in their constructor.
- There may not be any method, member or object in scope with the same name as the implicit class.

Implicit Class Example

```
object Helpers {
  implicit class IntWithTimes(x: Int) {
    def times[A](f: => A): Unit = {
      def loop(current: Int): Unit =
        if(current > 0) {
          f
          loop(current - 1)
        }
      loop(x)
    }
  }
}
```



```
scala> import Helpers._
import Helpers._

scala> 4 times println("Dan")
Dan
Dan
Dan
Dan

scala> def myFun1(x: Int, y: String) {
     | println(x, y)
     | }
myFun1: (x: Int, y: String)Unit

scala> 3 times myFun1(2, "Lo")
(2,Lo)
(2,Lo)
(2,Lo)
```

Generic Class

- Generic classes in Scala take a type as a parameter within square brackets []
- It gives a flexible way to create a class that deals with multiple data types.

```
class Stack[A] {
  private var elements: List[A] = Nil
  def push(x: A) { elements = x :: elements }
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

Use a Generic Class

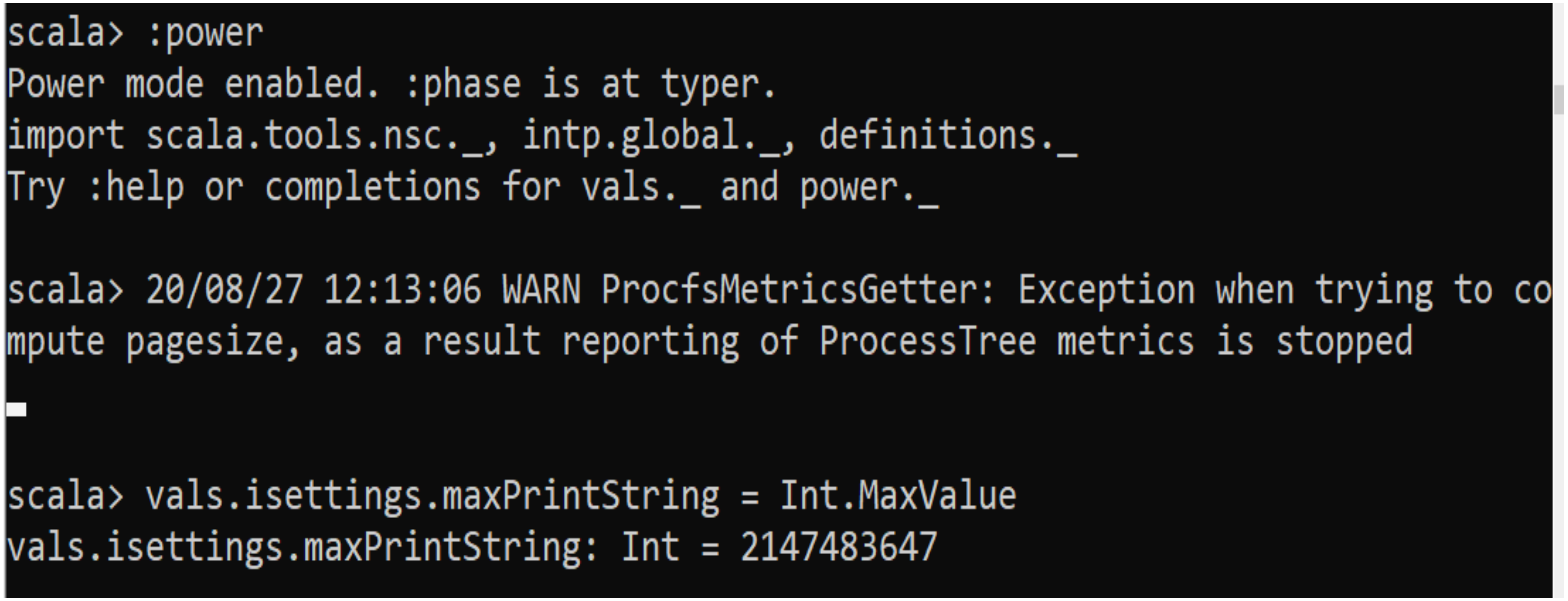
```
val stack = new Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop)  // prints 2
println(stack.pop)  // prints 1
```

Spark Shell

- spark-shell --master local[4] will start a Spark shell running on local machine with 4 CPUs.
- The --master option specifies the master URL for a distributed cluster, or local to run locally with one thread, or local[N] to run locally with N threads.
- You should start by using local for testing. For a full list of options, run Spark shell with the --help option.

Power User Mode

- Spark shell truncates output from your program. To fix that you can set the maximum length of the strings printed by the REPL in power user mode as follows:



```
scala> :power
Power mode enabled. :phase is at typer.
import scala.tools.nsc._, intp.global._, definitions._
Try :help or completions for vals._ and power._

scala> 20/08/27 12:13:06 WARN ProcfsMetricsGetter: Exception when trying to co
mpute pagesize, as a result reporting of ProcessTree metrics is stopped


scala> vals.isettings.maxPrintString = Int.MaxValue
vals.isettings.maxPrintString: Int = 2147483647
```

RDD Operations

- Scala programs are running on a cluster, multiple nodes, for parallel computing.
- RDD are lazy design; execute when an actual action is called.
- Two types of operations
    - *transformations*, which create a new dataset from an existing one, and
    - *actions*, which return a value to the driver program after running a computation on the dataset.
- Two types of shared variable in Spark
    - Broadcast variables, used to cache a value in memory on all nodes
    - Accumulator, used for counters and sums, globally across all nodes

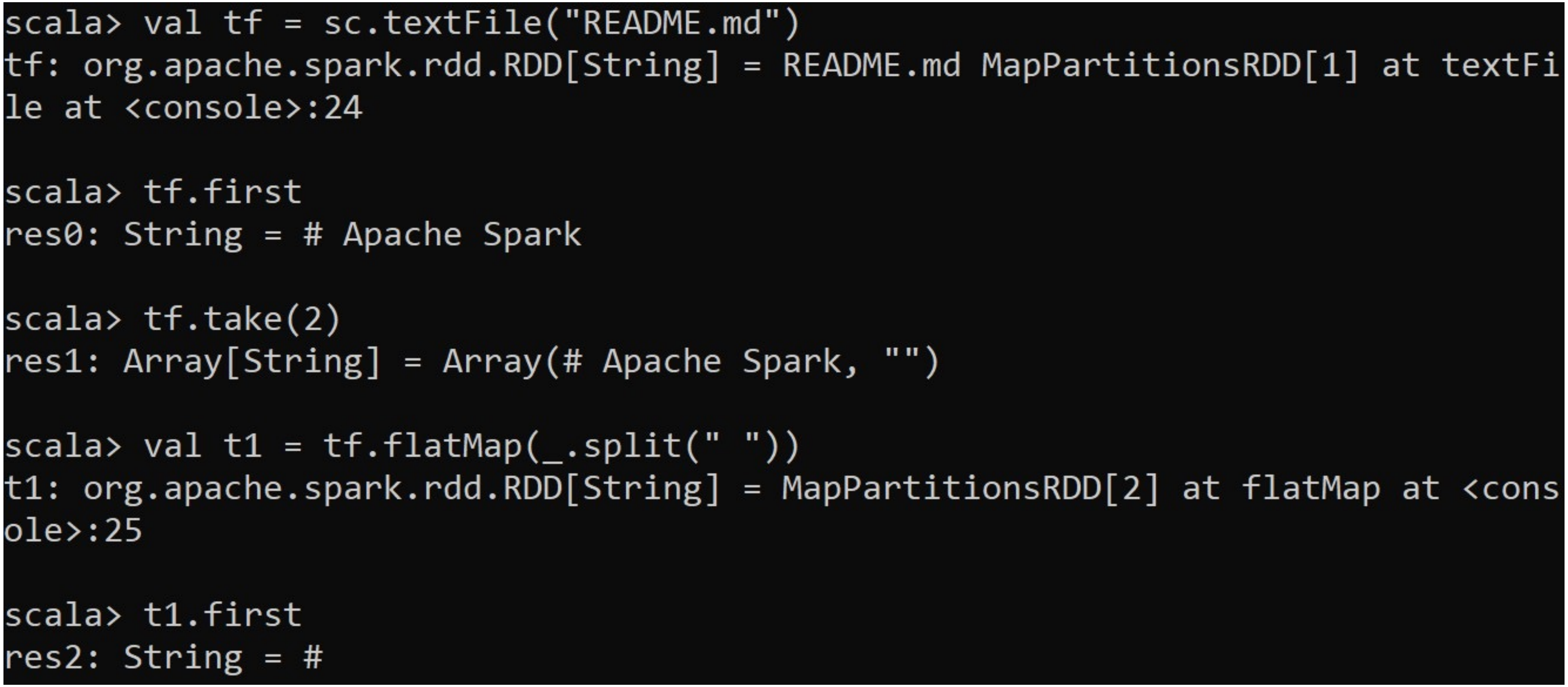
RDD Transformations (subset)

- map(func) Return a new distributed dataset formed by passing each element of the source through a function func.
- filter(func) Return a new dataset formed by selecting those elements of the source on which func returns true.
- flatMap(func) Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
- groupByKey([numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: Aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.
- reduceByKey(func, [numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
- aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations.
- sortByKey([ascending], [numPartitions]) When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
- join(otherDataset, [numPartitions]) When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

RDD Actions (subset)

- reduce(func) Aggregate the elements of the dataset using a commutative and associative function func (which takes two arguments and returns one).
- collect() Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter for a small subset of the data.
- count() Return the number of elements in the dataset.
- first() Return the first element of the dataset (similar to take(1)).
- take(n) Return an array with the first n elements of the dataset.
- takeOrdered(n, [ordering]) Return the first n elements of the RDD using either their natural order or a custom comparator.
- saveAsTextFile(path) Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
- saveAsSequenceFile(path) (Java and Scala) Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
- saveAsObjectFile(path) (Java and Scala) Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
- countByKey() Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
- foreach(func) Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Example



```
scala> val tf = sc.textFile("README.md")
tf: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at textFi
le at <console>:24

scala> tf.first
res0: String = # Apache Spark

scala> tf.take(2)
res1: Array[String] = Array(# Apache Spark, "")

scala> val t1 = tf.flatMap(_.split(" "))
t1: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <cons
ole>:25

scala> t1.first
res2: String = #
```

Running RDD Operations on Spark

- Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor.
- Prior to execution, Spark computes the task’s closure.
- The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD (in the following example, the foreach()).
- This closure is serialized and sent to each executor as a copy.

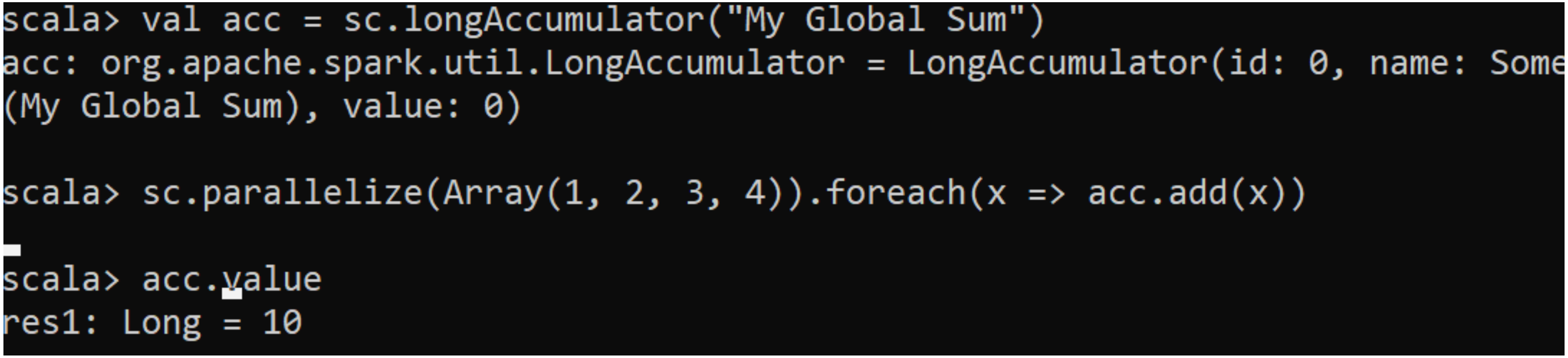
Example: Architecture Pitfalls

```
var counter = 0 // this counter used in foreach below will be sent to each
executor (multiple copies)
var rdd = sc.parallelize(data) //distribute user created data over the cluster

// Wrong: Don't do this!!
rdd.foreach(x => counter += x)

println("Counter value: " + counter)
```

Program Behaves Differently

- The variables within the closure sent to each executor are now copies and thus, when counter is referenced within the foreach function, it’s no longer the counter on the driver node.
- There is still a counter in the memory of the driver node but this is no longer visible to the executors!
- The final value of **counter** will still be zero since all operations on **counter** were referencing the value within the serialized closure.
- Use Accumulator instead for counters or sums.

Accumulators Example



```
scala> val acc = sc.longAccumulator("My Global Sum")
acc: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some
(My Global Sum), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => acc.add(x))


scala> acc.value
res1: Long = 10
```

For more information

- Scala Documentation https://docs.scala-lang.org/
- Scala Book, Meriam Lachkar and Alvin Alexander, https://docs.scala-lang.org/overviews/scala-book/introduction.html
- Programming in Scala, First Edition by Martin Odersky, Lex Spoon, and Bill Venners, https://www.artima.com/pins1ed/
- RDD Programming Guide, https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations