BIG DATA ANALYTICS

Parallel Programming with Spark

Fundamentals of Scala and Functional Programming

Parallel Programming with Spark: *Outline*

- Overview of Spark
- Fundamentals of Scala and Functional Programming 
- Spark Concepts
  - Resilient Distributed Datasets (RDD)
  - Creating RDDs
  - Basic Transformations
  - Basic Actions
  - Word Count Example
- Spark operations
- Job execution
- Spark Applications : Cluster computing with working sets

Scala

The Scala Programming Language:

- Scala combines object-oriented and functional programming in one concise, high-level language.
- Statically typed, concise, and interoperable with Java.
- Scala's static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries.

Scala

What is Scala?

- Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way.
- It seamlessly integrates features of object-oriented and functional languages.

Scala

Scala is object-oriented:

- Scala is a pure object-oriented language in the sense that every value is an object.
- Types and behaviors of objects are described by classes and features.
- Classes can be extended by subclassing, and by using a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

Scala

Scala is functional:

- Scala is also a functional language in the sense that every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and it supports currying. Scala's case classes and its built-in support for pattern matching provide the functionality of algebraic types, which are used in many functional languages. Singleton objects provide a convenient way to group functions that aren't members of a class.
- Furthermore, Scala's notion of pattern matching naturally extends to the processing of XML data with the help of right-ignoring sequence patterns, by way of general extension via extractor objects. In this context, for comprehensions are useful for formulating queries. These features make Scala ideal for developing applications like web services.

Scala

Scala is statically typed:

Scala's expressive type system enforces, at compile-time, that abstractions are used in a safe and coherent manner. In particular, the type system supports:

- Generic classes
- Variance annotations
- Upper and lower type bounds
- Inner classes and abstract type members as object members
- Compound types
- Explicitly typed self references
- Implicit parameters and conversions
- Polymorphic methods

Scala

Scala is statically typed:

- Type inference means the user is not required to annotate code with redundant type information.
- In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

Scala

Scala is extensible:

- In practice, the development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it straightforward to add new language constructs in the form of libraries.
- In many cases, this can be done without using meta-programming facilities such as macros. For example:
  - Implicit classes allow adding extension methods to existing types.
  - String interpolation is user-extensible with custom interpolators.

Scala

Scala interoperates:

- Scala is designed to interoperate well with the popular Java Runtime Environment (JRE). In particular, the interaction with the mainstream object-oriented Java programming language is as seamless as possible. Newer Java features like SAMs (single abstract method), lambdas, annotations, and generics have direct analogues in Scala.
- Those Scala features without Java analogues, such as default and named parameters, compile as closely to Java as reasonably possible. Scala has the same compilation model (separate compilation, dynamic class loading) as Java and allows access to thousands of existing high-quality libraries.

Spark Scala: *Environment*

- To introduce Spark, let's run an interactive session using ***spark-shell***, which is a Scala REPL with a few Spark additions.
- The Scala REPL ("Read-Evaluate-Print-Loop") is a command-line interpreter that we use as a "playground" area to test your Scala code.
- To start a REPL session, just type scala or spark-shell at our operating system command line.
- Start up the shell with the following:

```
$ spark-shell or $ scala
```

Spark context available as **sc**.

scala>

Scala: *Quick Tour*

- To introduce Spark, let's run an interactive session using ***spark-shell***, which is a Scala REPL with a few Spark additions.
- The Scala REPL ("Read-Evaluate-Print-Loop") is a command-line interpreter that we use as a "playground" area to test your Scala code.
- To start a REPL session, just type scala or spark-shell at our operating system command line.
- Start up the shell with the following:

```
$ spark-shell or $ scala
```

Spark context available as **sc**.

scala>

Scala: *Quick Tour*

- To start Scala:

```
hadoop@ubuntu22:~$ cd /home/hadoop/spark-3.5.2-bin-hadoop3
hadoop@ubuntu22:~/spark-3.5.2-bin-hadoop3$ cd bin
hadoop@ubuntu22:~/spark-3.5.2-bin-hadoop3/bin$ ./spark-shell
```

```
Using Scala version 2.12.18 (OpenJDK 64-Bit Server VM, Java 1.8.0_432)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Scala: *Quick Tour*

Variables:

- Use `val` for immutable variables and `var` for mutable ones.

```
val immutableValue = 10     // Cannot be reassigned
var mutableValue = 20       // Can be reassigned
mutableValue = 25           // Reassigning
println(immutableValue, mutableValue)
```

Output:

```
(10,25)
```

Scala: *Quick Tour*

Variables:

```
scala> val immutableValue = 10
immutableValue: Int = 10

scala> var mutableValue = 20
mutableValue: Int = 20

scala> mutableValue = 25
mutableValue: Int = 25

scala> println(immutableValue, mutableValue)
(10,25)

scala>
```

Scala: *Quick Tour*

Basic Data Types:

```
val num: Int = 42
val pi: Double = 3.14
val greeting: String = "Hello, Scala!"
val isScalaFun: Boolean = true
println(num)
println(pi)
println(greeting)
println(isScalaFun)
```

Output:

```
42
3.14
Hello, Scala!
true
```

Scala: *Quick Tour*

Basic Data Types:

```
scala> val num: Int = 42
num: Int = 42

scala> val pi: Double = 3.14
pi: Double = 3.14

scala> val greeting: String = "Hello, Scala!"
greeting: String = Hello, Scala!

scala> val isScalaFun: Boolean = true
isScalaFun: Boolean = true

scala> println(num)
42

scala> println(pi)
3.14

scala> println(greeting)
Hello, Scala!

scala> println(isScalaFun)
true
```

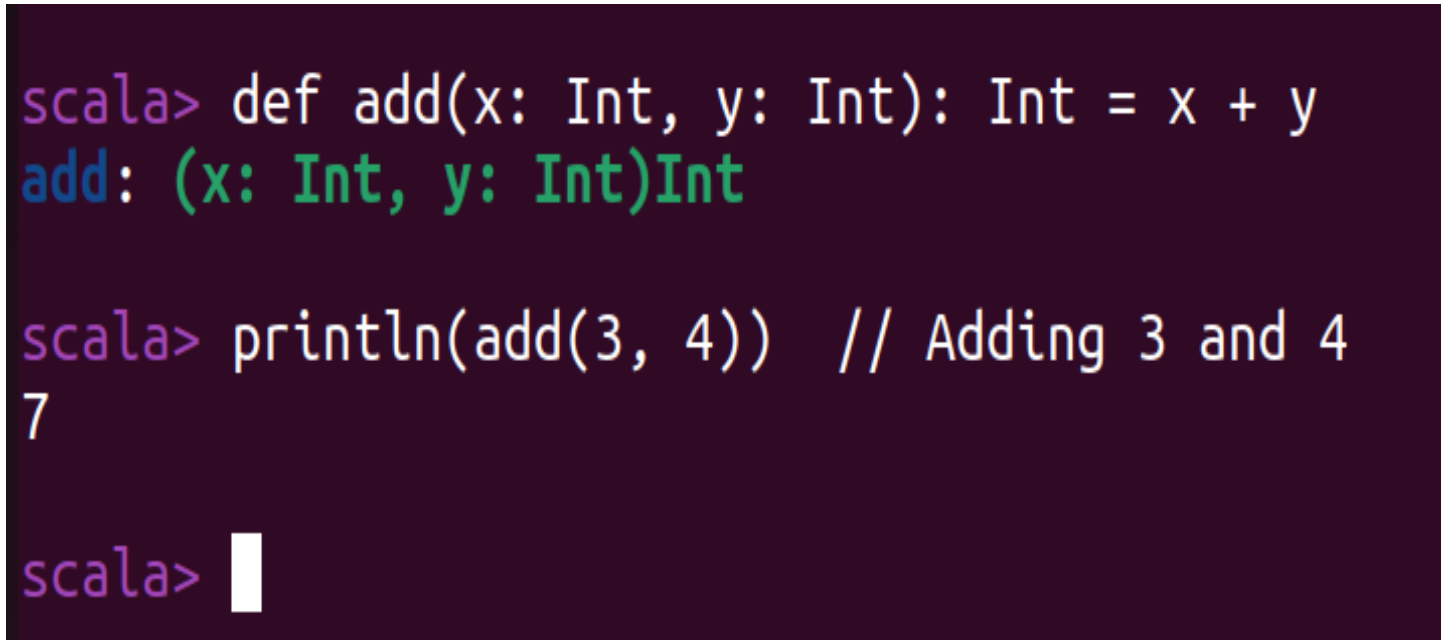
Scala: *Quick Tour*

Functions:

```
def add(x: Int, y: Int): Int = x + y
println(add(3, 4))   // Adding 3 and 4
```

Output:

```
7
```



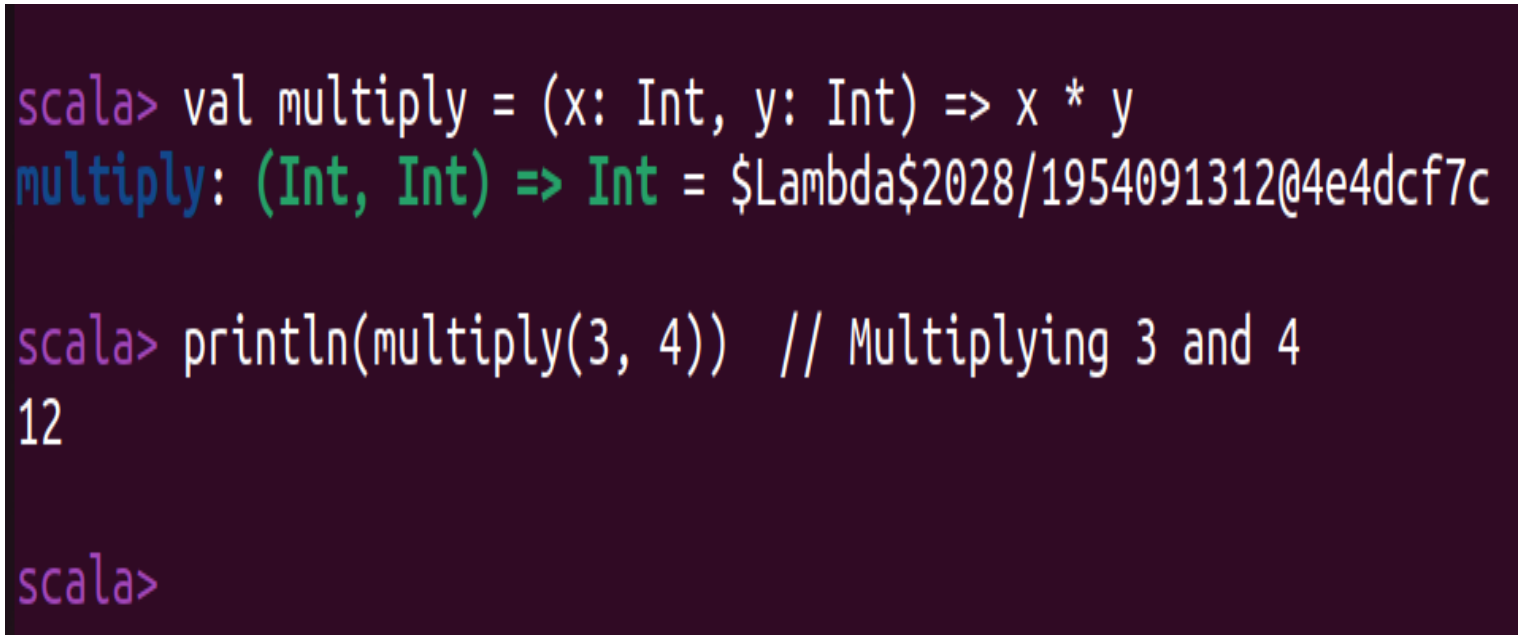
Scala: *Quick Tour*

Anonymous Function (Lambda):

```
val multiply = (x: Int, y: Int) => x * y
println(multiply(3, 4))   // Multiplying 3 and 4
```

Output:

```
12
```



Scala: *Quick Tour*

Control Structures: if

```
val age = 18
if (age >= 18) {
  println("Eligible to vote")
} else {
  println("Not eligible to vote")
}
```

Output:

```
Eligible to vote
```

Scala: *Quick Tour*

Control Structures: if

```
scala> val age = 18
age: Int = 18

scala> if (age >= 18) {
     |   println("Eligible to vote")
     | } else {
     |   println("Not eligible to vote")
     | }
Eligible to vote

scala>
```

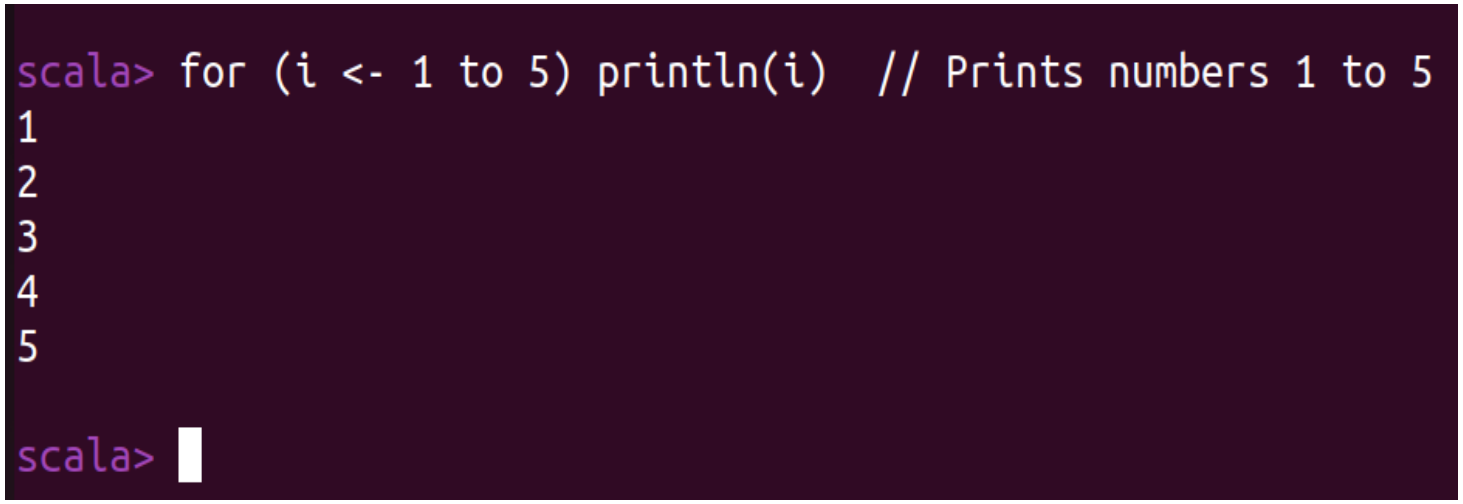
Scala: *Quick Tour*

Control Structures: for loop

```
for (i <- 1 to 5) println(i)   // Prints numbers 1 to 5
```

Output:

```
1
2
3
4
5
```



Scala: *Quick Tour*

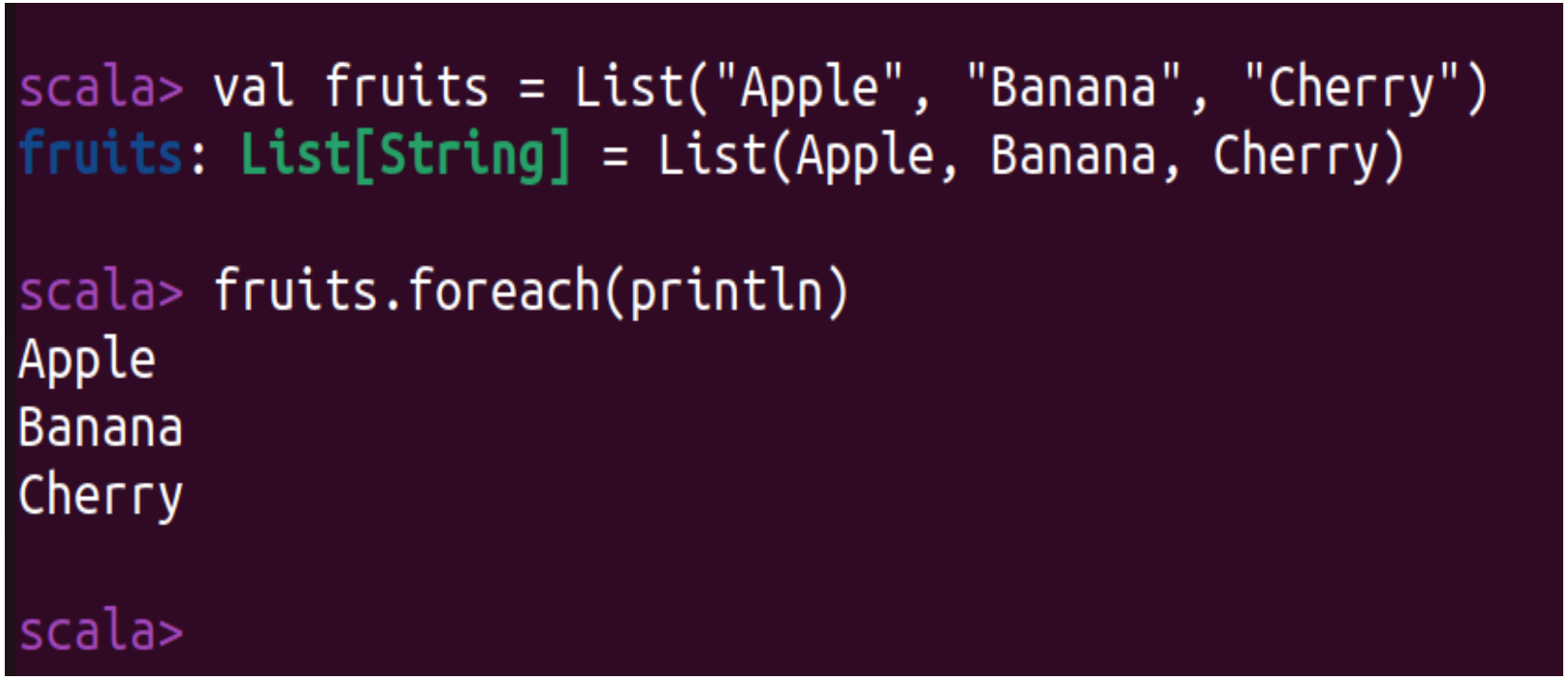
Collections:

List

```
val fruits = List("Apple", "Banana", "Cherry")
fruits.foreach(println)
```

Output:

```
Apple
Banana
Cherry
```



Scala: *Quick Tour*

Collections:

Map

```
val capitals = Map("India" -> "New Delhi", "USA" ->
"Washington D.C.")

println(capitals("India"))
```

Output:

```
New Delhi
```

```
scala> val capitals = Map("India" -> "New Delhi", "USA" -> "Washington D.C.")
capitals: scala.collection.immutable.Map[String,String] = Map(India -> New Delh
i, USA -> Washington D.C.)

scala> println(capitals("India"))
New Delhi

scala>
```

Scala: *Quick Tour*

Collections:

Set

```
val uniqueNums = Set(1, 2, 2, 3)
println(uniqueNums)
```

Output:

```
Set(1, 2, 3)
```

```
scala> val uniqueNums = Set(1, 2, 2, 3)
uniqueNums: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> println(uniqueNums)
Set(1, 2, 3)

scala>
```

Scala: *Quick Tour*

Higher-Order Functions:

```
val nums = List(1, 2, 3, 4)
val doubled = nums.map(x => x * 2)   // Double each element
println(doubled)
```

Output:

```
List(2, 4, 6, 8)
```

```
scala> val nums = List(1, 2, 3, 4)
nums: List[Int] = List(1, 2, 3, 4)

scala> val doubled = nums.map(x => x * 2)  // Double each element
doubled: List[Int] = List(2, 4, 6, 8)

scala> println(doubled)
List(2, 4, 6, 8)

scala>
```

Scala: *Quick Tour*

Classes and Objects:

```
class Person(val name: String, var age: Int) {

  def greet(): Unit = println(s"Hi, my name is $name, and I
am $age years old.")

}

val person = new Person("John", 30)

person.greet()
```

Output:

```
Hi, my name is John, and I am 30 years old.
```

Scala: *Quick Tour*

Classes and Objects:

```
scala> class Person(val name: String, var age: Int) {
     |   def greet(): Unit = println(s"Hi, my name is $name, and I am $age year
s old.")
     | }
defined class Person

scala>

scala> val person = new Person("John", 30)
person: Person = Person@4b821389

scala> person.greet()
Hi, my name is John, and I am 30 years old.

scala>
```

Scala: *Quick Tour*

Traits:

```
trait Animal {
  def sound(): Unit
}
class Dog extends Animal {
  def sound(): Unit = println("Woof!")
}
val dog = new Dog()
dog.sound()
```

Output:

```
Woof!
```

Scala: *Quick Tour*

Traits:

```
scala> trait Animal {
     |   def sound(): Unit
     | }
defined trait Animal

scala>

scala> class Dog extends Animal {
     |   def sound(): Unit = println("Woof!")
     | }
defined class Dog

scala>

scala> val dog = new Dog()
dog: Dog = Dog@4d813b5f

scala> dog.sound()
Woof!

scala>
```

Scala: *Quick Tour*

Pattern Matching:

```
val day = "Monday"
day match {
  case "Monday" => println("Start of the week!")
  case "Friday" => println("Weekend is near!")
  case _ => println("Just another day!")
}
```

Output:

```
Start of the week!
```

Scala: *Quick Tour*

Pattern Matching:

```
scala> val day = "Monday"
day: String = Monday

scala> day match {
     |   case "Monday" => println("Start of the week!")
     |   case "Friday" => println("Weekend is near!")
     |   case _ => println("Just another day!")
     | }
Start of the week!

scala>
```

Scala: *Quick Tour*

Functional Programming:

Immutability Example

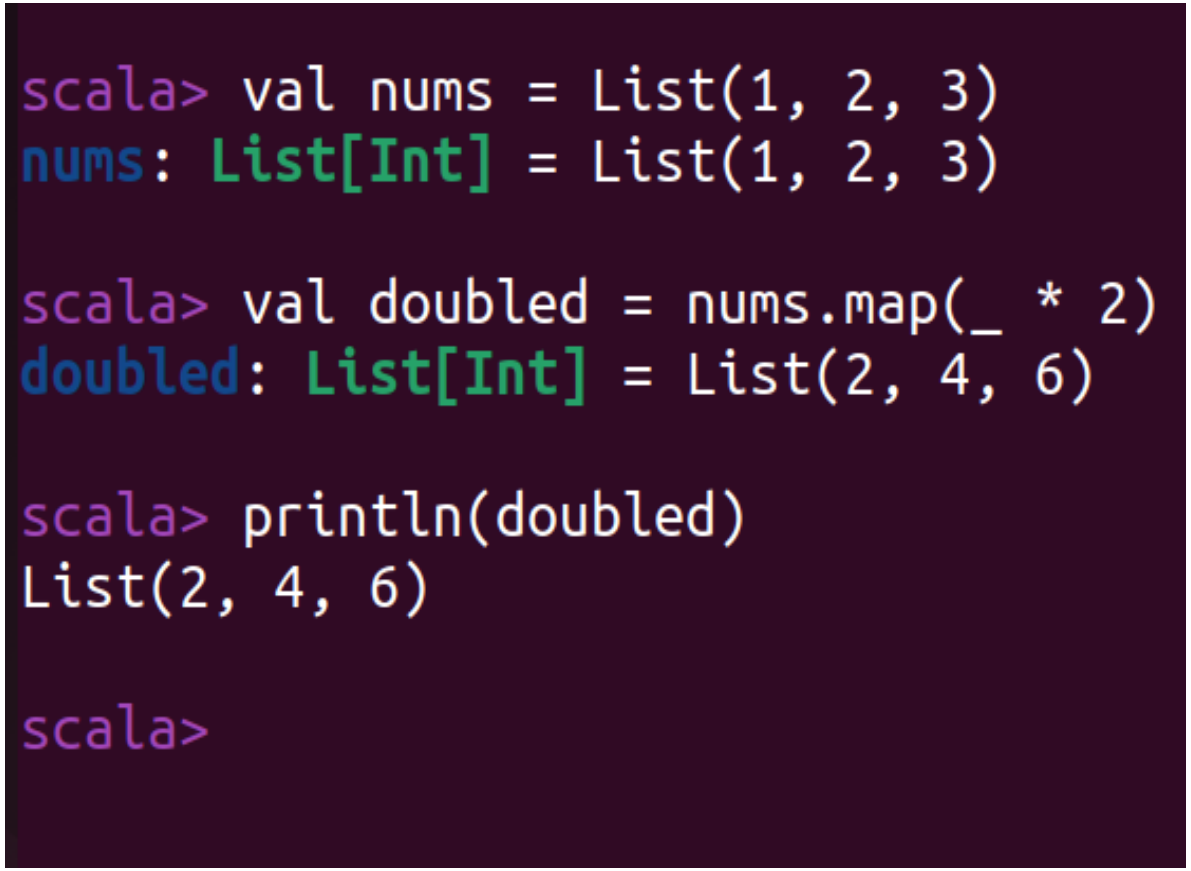
```
val nums = List(1, 2, 3)

val doubled = nums.map(_ * 2)

println(doubled)
```

Output:

```
List(2, 4, 6)
```



Scala: *Quick Tour*

Functional Programming:

Closures Example

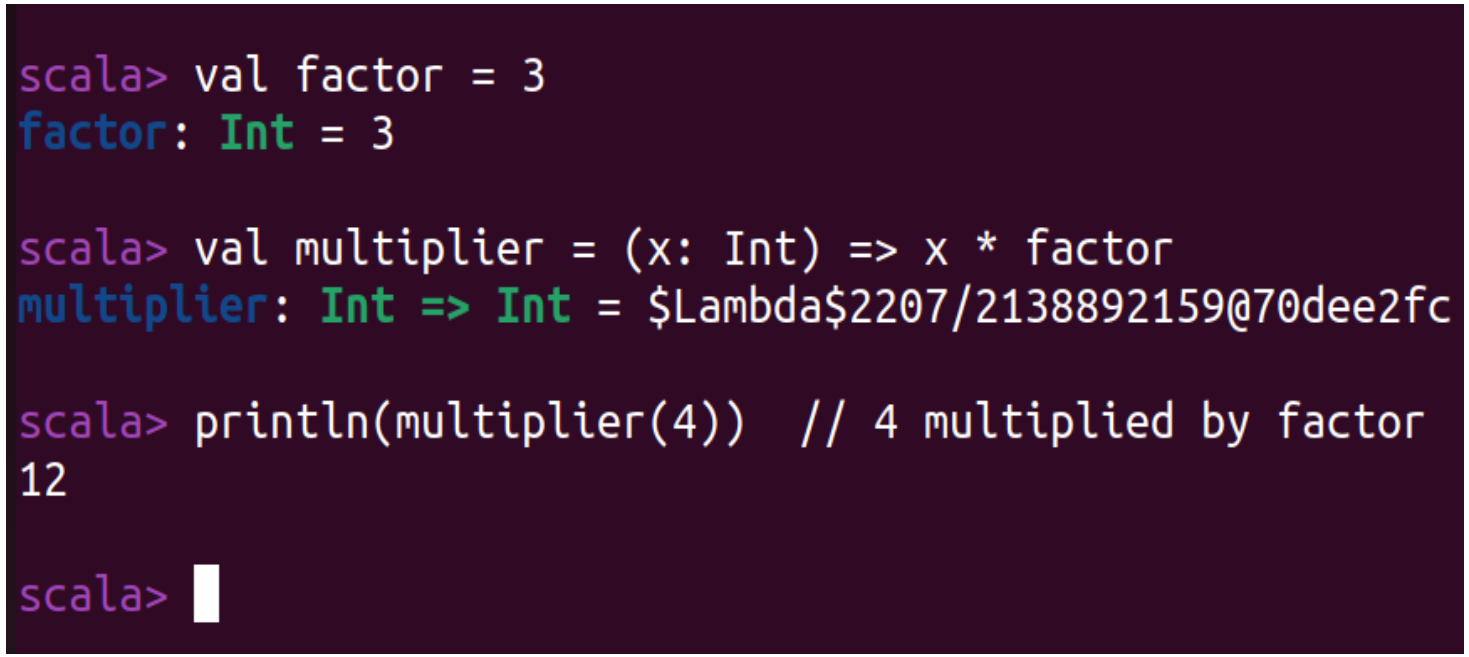
```
val factor = 3

val multiplier = (x: Int) => x * factor

println(multiplier(4))   // 4 multiplied by factor
```

Output:

```
12
```



Scala: *Quick Tour*

Concurrency with Futures:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
val future = Future {
  Thread.sleep(1000)   // Simulating async operation
  42
}
future.map(result => println(s"The answer is $result"))
Thread.sleep(2000)   // Wait for the future to complete
```

Output:

```
The answer is 42
```

Scala: *Quick Tour*

Concurrency with Futures:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val future = Future {
     |   Thread.sleep(1000)  // Simulating async operation
     |   42
     | }
future: scala.concurrent.Future[Int] = Future(<not completed>)

scala> future.map(result => println(s"The answer is $result"))
res20: scala.concurrent.Future[Unit] = Future(<not completed>)
The answer is 42

scala> Thread.sleep(2000)  // Wait for the future to complete

scala>
```