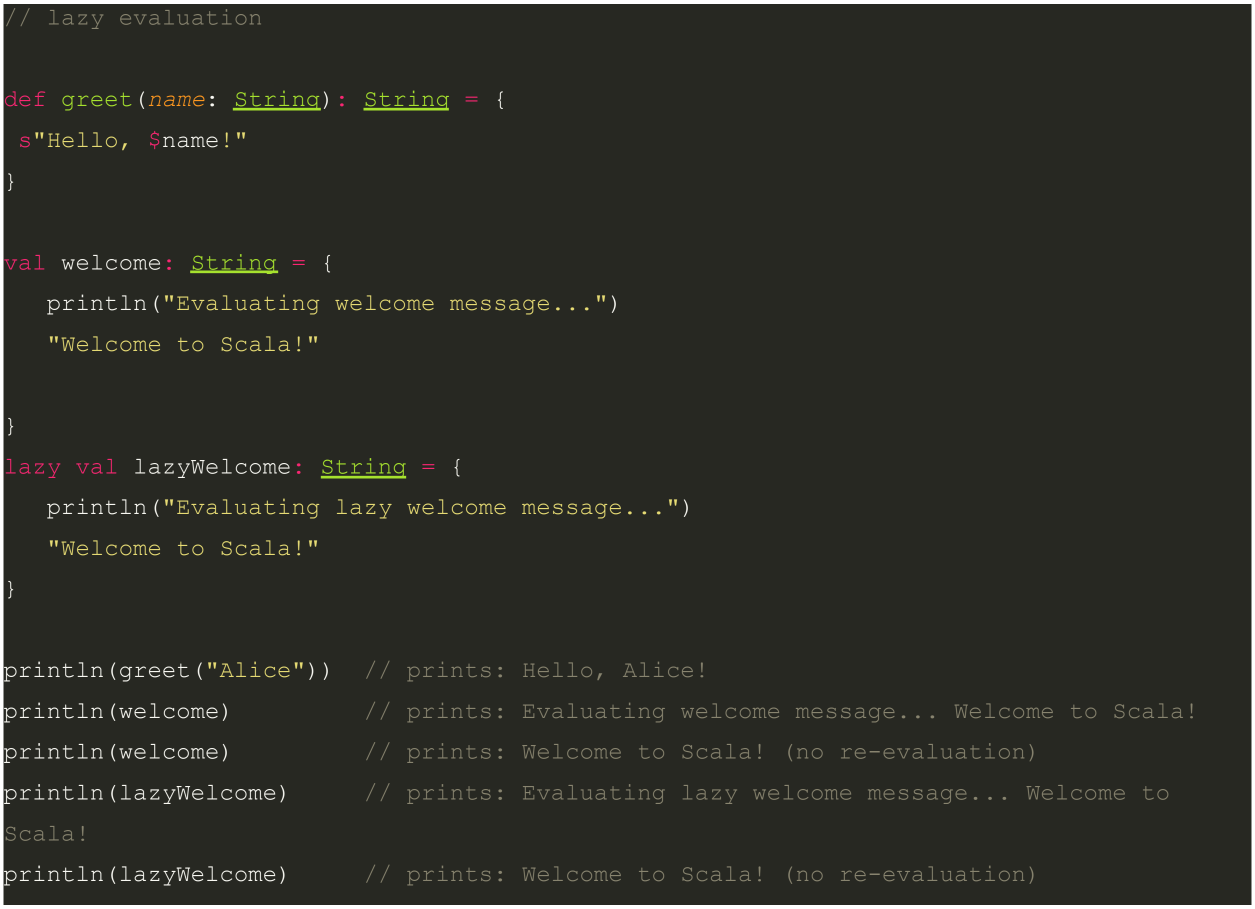
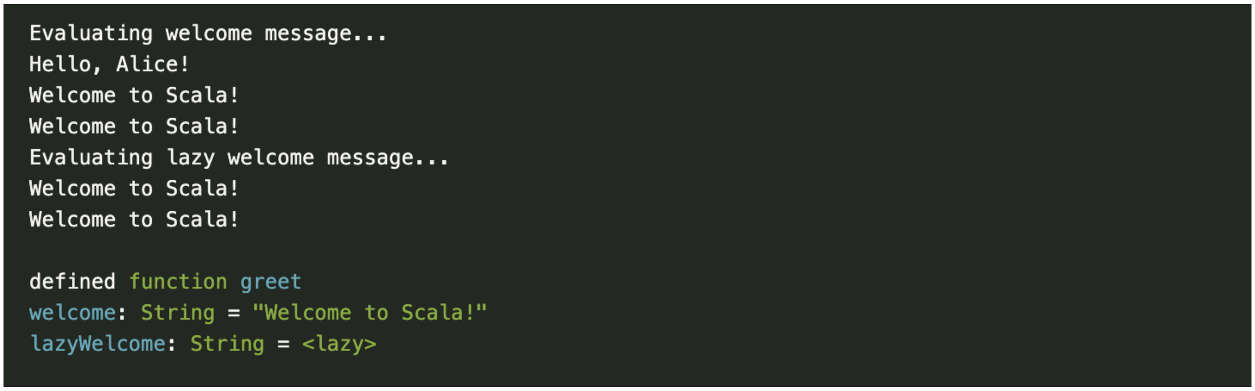
Day 3 - Concepts

1. Lazy Evaluation



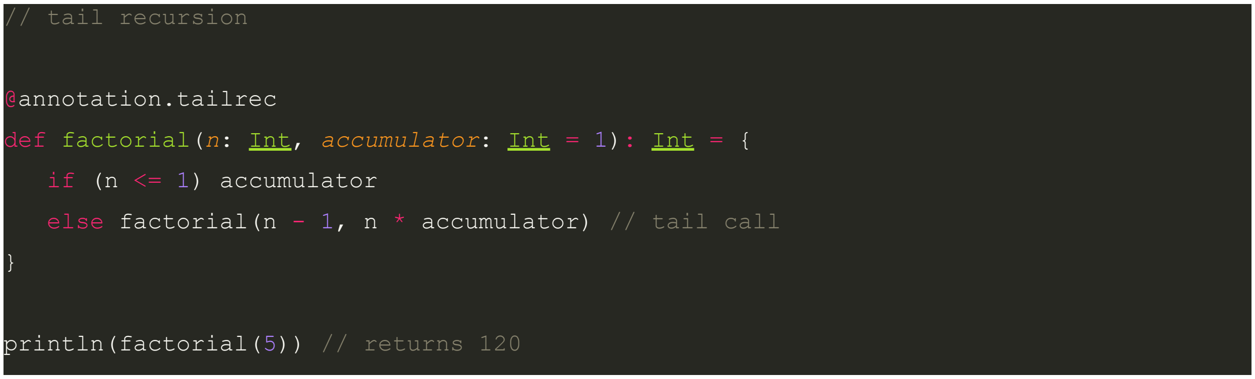
```
// lazy evaluation

def greet(name: String): String = {
 s"Hello, $name!"
}

val welcome: String = {
   println("Evaluating welcome message...")
   "Welcome to Scala!"

}
lazy val lazyWelcome: String = {
   println("Evaluating lazy welcome message...")
   "Welcome to Scala!"
}

println(greet("Alice"))   // prints: Hello, Alice!
println(welcome)          // prints: Evaluating welcome message... Welcome to Scala!
println(welcome)          // prints: Welcome to Scala! (no re-evaluation)
println(lazyWelcome)      // prints: Evaluating lazy welcome message... Welcome to
Scala!
println(lazyWelcome)      // prints: Welcome to Scala! (no re-evaluation)
```

Output:



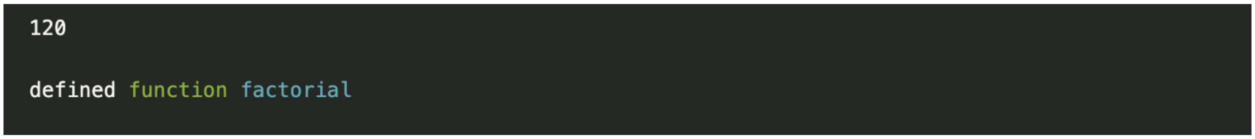
```
Evaluating welcome message...
Hello, Alice!
Welcome to Scala!
Welcome to Scala!
Evaluating lazy welcome message...
Welcome to Scala!
Welcome to Scala!

defined function greet
welcome: String = "Welcome to Scala!"
lazyWelcome: String = <lazy>
```

2. Tail Recursion



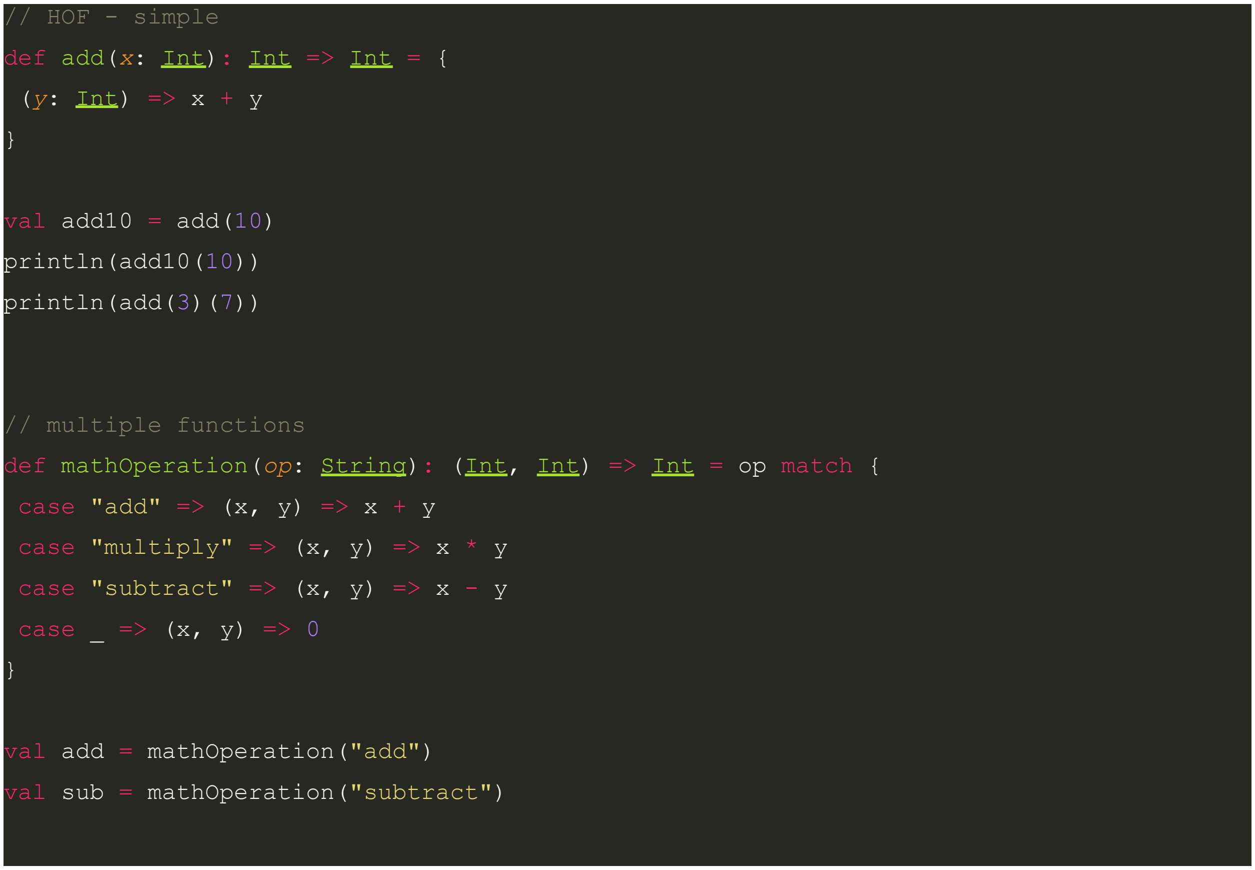
```
// tail recursion

@annotation.tailrec
def factorial(n: Int, accumulator: Int = 1): Int = {
   if (n <= 1) accumulator
   else factorial(n - 1, n * accumulator) // tail call
}

println(factorial(5)) // returns 120
```

Output:



```
120

defined function factorial
```

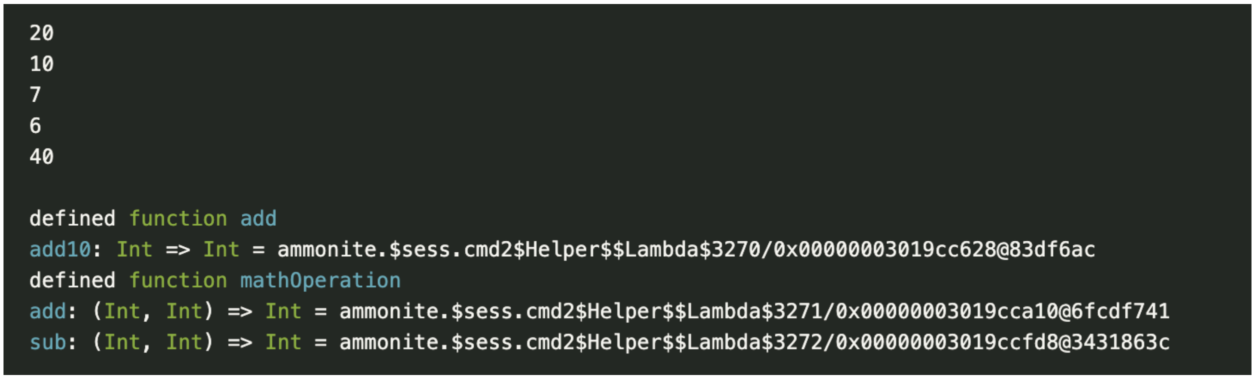
3. Function returning functions



```
// HOF - simple
def add(x: Int): Int => Int = {
 (y: Int) => x + y
}

val add10 = add(10)
println(add10(10))
println(add(3)(7))


// multiple functions
def mathOperation(op: String): (Int, Int) => Int = op match {
 case "add" => (x, y) => x + y
 case "multiply" => (x, y) => x * y
 case "subtract" => (x, y) => x - y
 case _ => (x, y) => 0
}

val add = mathOperation("add")
val sub = mathOperation("subtract")
```

```
println(add(3, 4))
println(sub(10, 4))
println(mathOperation("multiply")(10, 4))
```

Output:

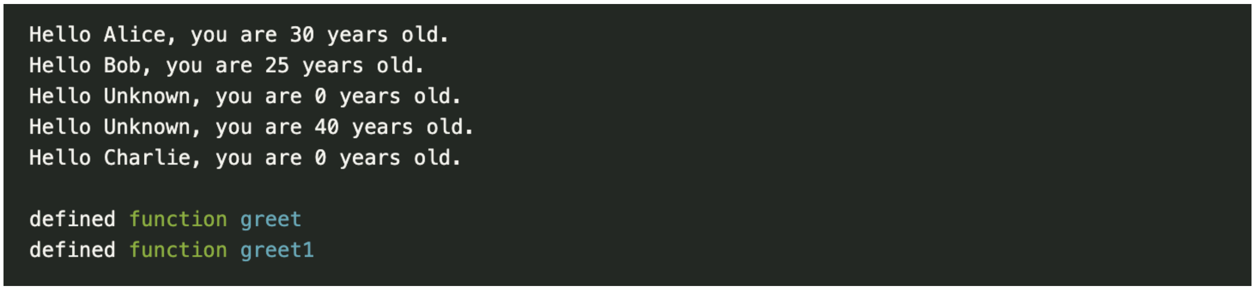


```
20
10
7
6
40

defined function add
add10: Int => Int = ammonite.$sess.cmd2$Helper$$Lambda$3270/0x00000003019cc628@83df6ac
defined function mathOperation
add: (Int, Int) => Int = ammonite.$sess.cmd2$Helper$$Lambda$3271/0x00000003019cca10@6fcdf741
sub: (Int, Int) => Int = ammonite.$sess.cmd2$Helper$$Lambda$3272/0x00000003019ccfd8@3431863c
```

4. Function with keyword parameters

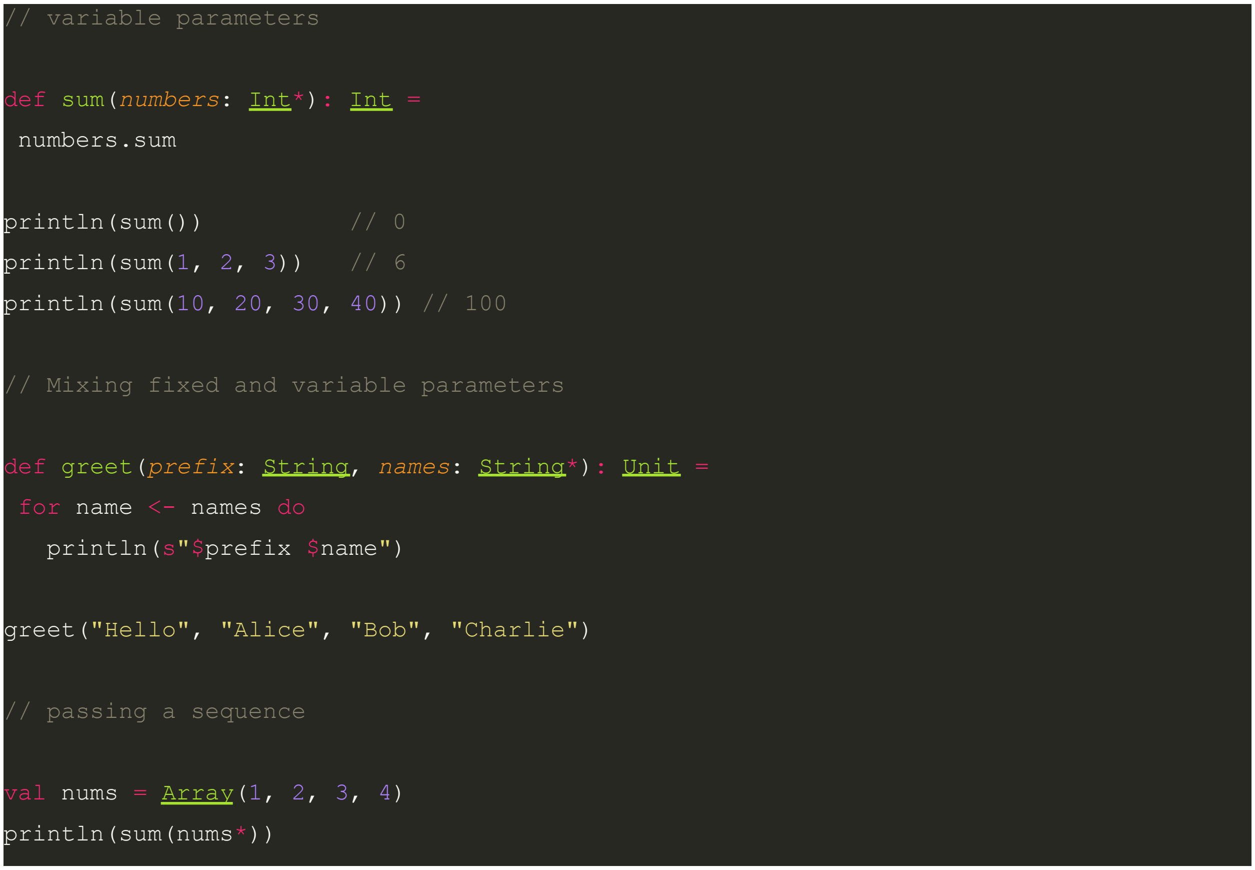
```
// function with keyword parameters

def greet(name: String, age: Int): String =
 s"Hello $name, you are $age years old."

println(greet(name = "Alice", age = 30))  // Hello Alice, you are 30 years old.
println(greet(age = 25, name = "Bob"))    // Order can be swapped: Hello Bob, you are
25 years old.

// default values with keyword parameters

def greet(name: String = "Unknown", age: Int = 0): String =
 s"Hello $name, you are $age years old."

println(greet())                         // Hello Unknown, you are 0 years old.
println(greet(age = 40))                  // Hello Unknown, you are 40 years old.
println(greet(name = "Charlie"))          // Hello Charlie, you are 0 years old.
```

Output:



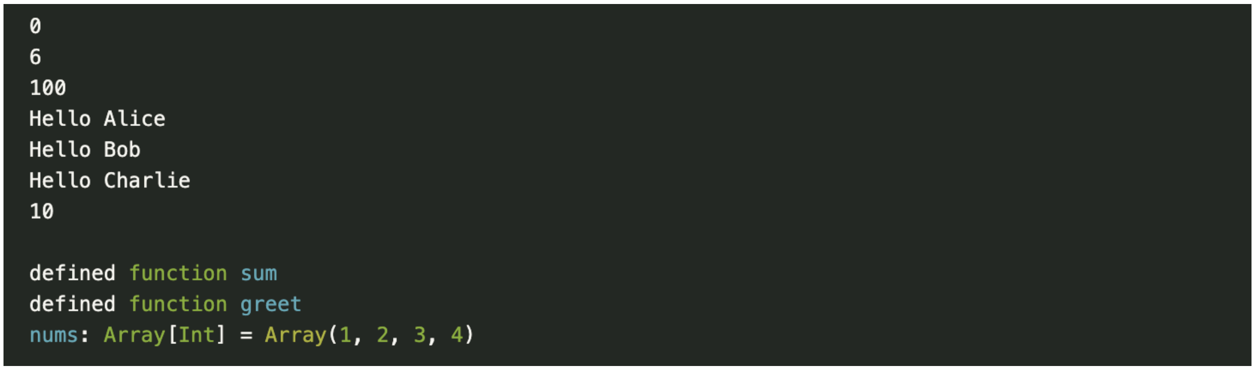
```
Hello Alice, you are 30 years old.
Hello Bob, you are 25 years old.
Hello Unknown, you are 0 years old.
Hello Unknown, you are 40 years old.
Hello Charlie, you are 0 years old.

defined function greet
defined function greet1
```

5. Function with variable no. of Parameters



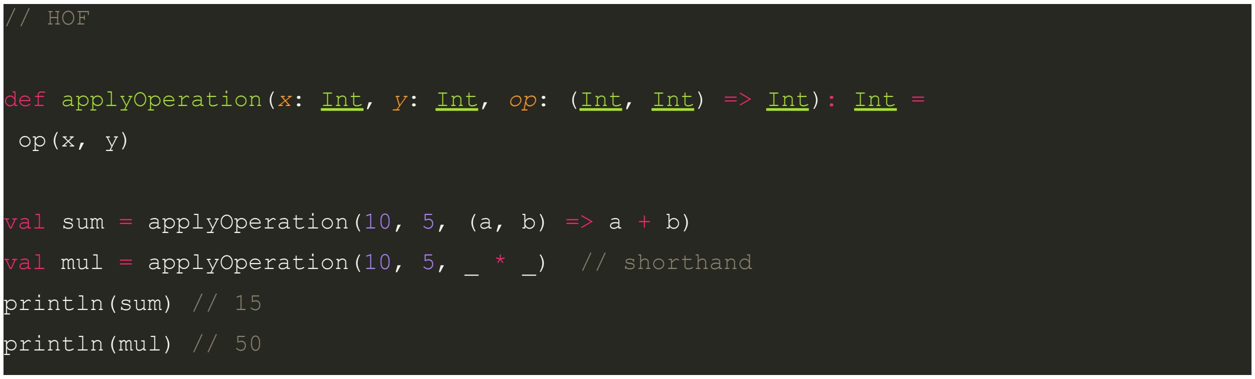
```scala
// variable parameters

def sum(numbers: Int*): Int =
 numbers.sum

println(sum())           // 0
println(sum(1, 2, 3))    // 6
println(sum(10, 20, 30, 40)) // 100

// Mixing fixed and variable parameters

def greet(prefix: String, names: String*): Unit =
 for name <- names do
   println(s"$prefix $name")

greet("Hello", "Alice", "Bob", "Charlie")

// passing a sequence

val nums = Array(1, 2, 3, 4)
println(sum(nums*))
```

Output:



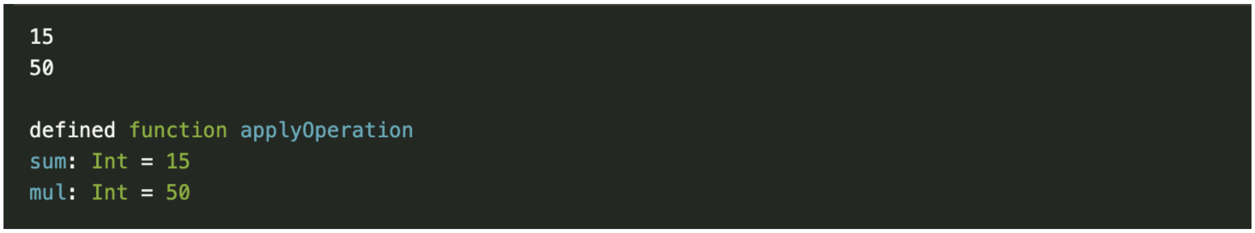
```
0
6
100
Hello Alice
Hello Bob
Hello Charlie
10

defined function sum
defined function greet
nums: Array[Int] = Array(1, 2, 3, 4)
```

6. Higher Order Function



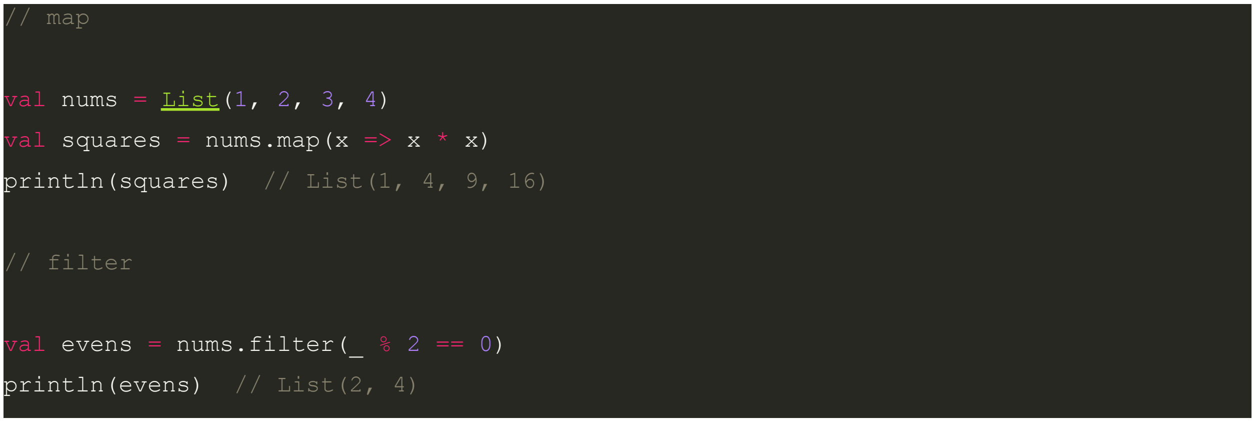
```scala
// HOF

def applyOperation(x: Int, y: Int, op: (Int, Int) => Int): Int =
 op(x, y)

val sum = applyOperation(10, 5, (a, b) => a + b)
val mul = applyOperation(10, 5, _ * _)  // shorthand
println(sum) // 15
println(mul) // 50
```

Output:



```
15
50

defined function applyOperation
sum: Int = 15
mul: Int = 50
```

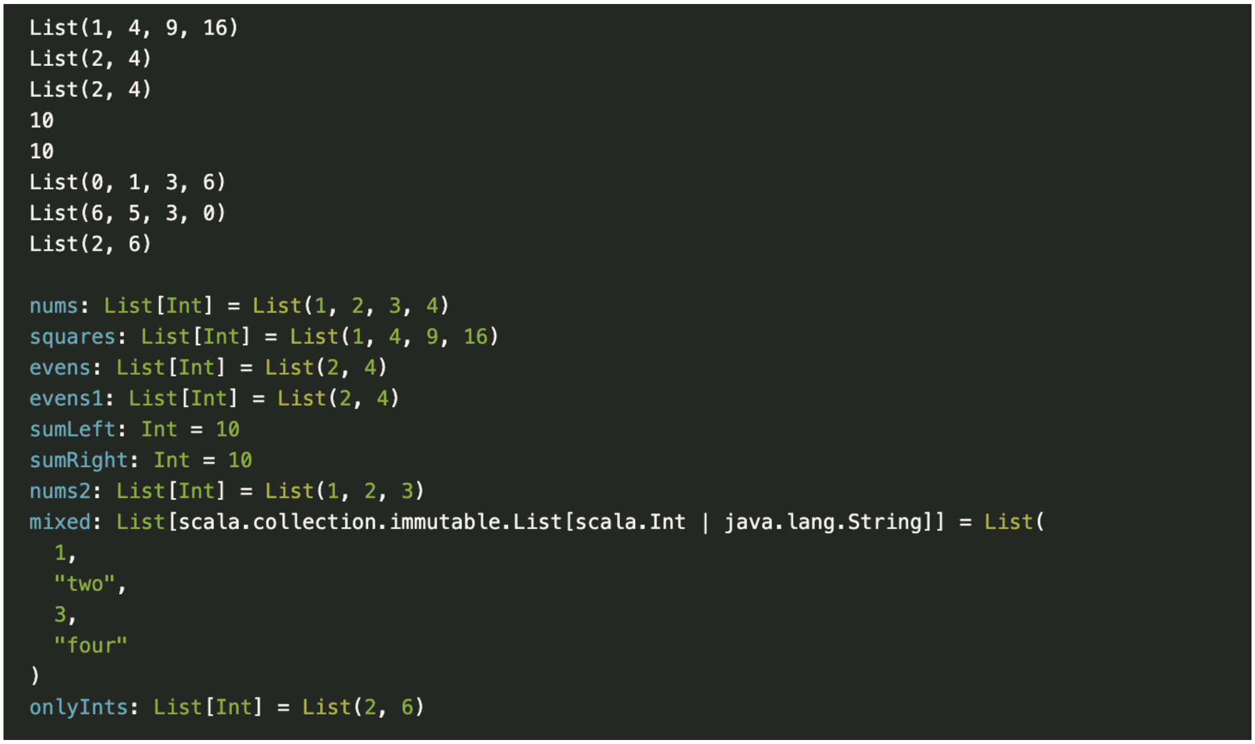
7. Map, reduce, filter , foldleft, foldRight ,scanLeft,scanRight and collect methods



```scala
// map

val nums = List(1, 2, 3, 4)
val squares = nums.map(x => x * x)
println(squares)  // List(1, 4, 9, 16)

// filter

val evens = nums.filter(_ % 2 == 0)
println(evens)  // List(2, 4)
```

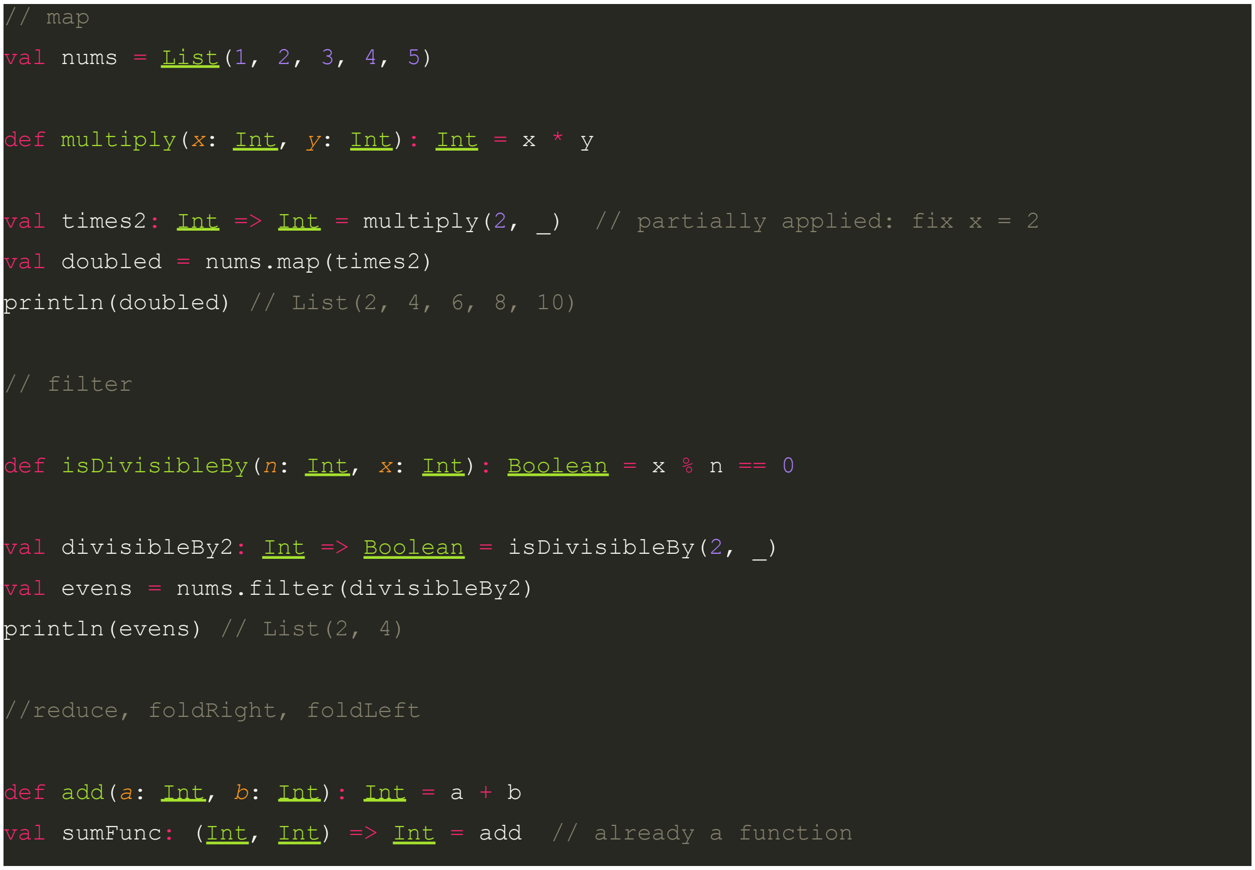
```
// reduce

val evens1 = nums.filter(_ % 2 == 0)
println(evens1)  // List(2, 4)

// foldLeft and foldRight
val sumLeft = nums.foldLeft(0)(_ + _)   // left-to-right
val sumRight = nums.foldRight(0)(_ + _) // right-to-left
println(sumLeft)   // 10
println(sumRight)  // 10

//scanLeft and scanRight
val nums2 = List(1, 2, 3)
println(nums2.scanLeft(0)(_ + _))  // List(0, 1, 3, 6)
println(nums2.scanRight(0)(_ + _)) // List(6, 5, 3, 0)

//collect
val mixed = List(1, "two", 3, "four")
val onlyInts = mixed.collect { case i: Int => i * 2 }
println(onlyInts) // List(2, 6)
```

Output:

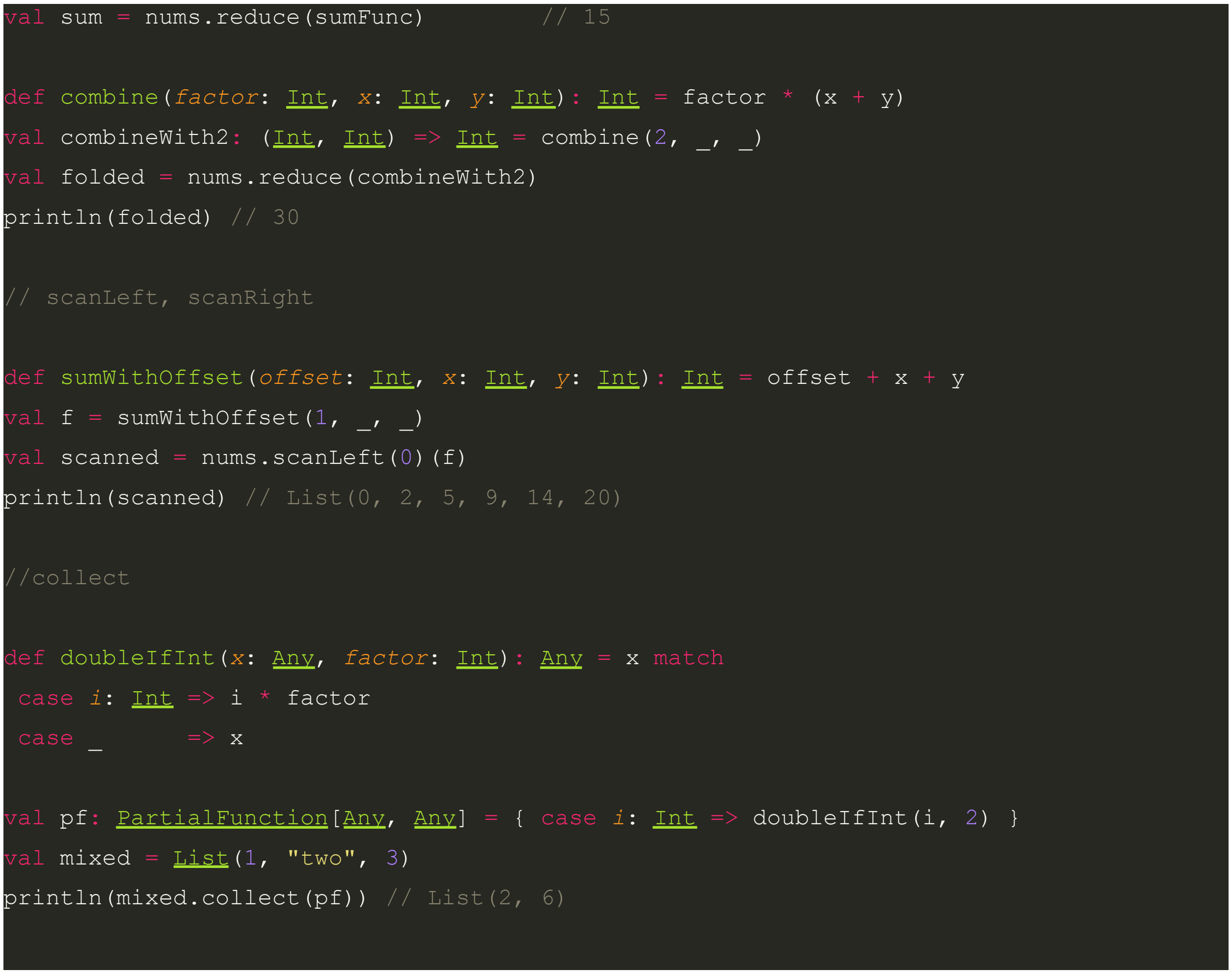


```
List(1, 4, 9, 16)
List(2, 4)
List(2, 4)
10
10
List(0, 1, 3, 6)
List(6, 5, 3, 0)
List(2, 6)

nums: List[Int] = List(1, 2, 3, 4)
squares: List[Int] = List(1, 4, 9, 16)
evens: List[Int] = List(2, 4)
evens1: List[Int] = List(2, 4)
sumLeft: Int = 10
sumRight: Int = 10
nums2: List[Int] = List(1, 2, 3)
mixed: List[scala.collection.immutable.List[scala.Int | java.lang.String]] = List(
  1,
  "two",
  3,
  "four"
)
onlyInts: List[Int] = List(2, 6)
```

8. Partial applications on the above methods

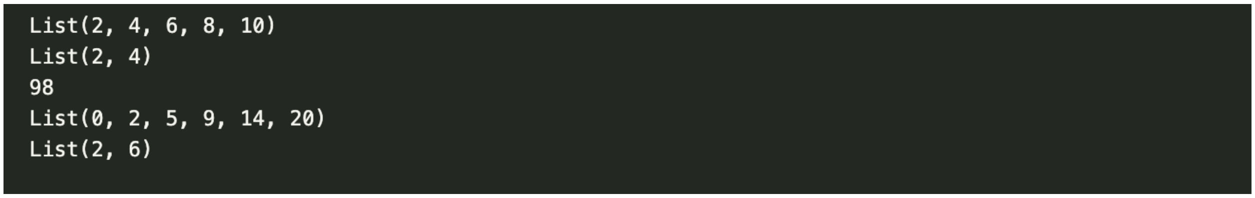


```
// map
val nums = List(1, 2, 3, 4, 5)

def multiply(x: Int, y: Int): Int = x * y

val times2: Int => Int = multiply(2, _)  // partially applied: fix x = 2
val doubled = nums.map(times2)
println(doubled) // List(2, 4, 6, 8, 10)

// filter

def isDivisibleBy(n: Int, x: Int): Boolean = x % n == 0

val divisibleBy2: Int => Boolean = isDivisibleBy(2, _)
val evens = nums.filter(divisibleBy2)
println(evens) // List(2, 4)

//reduce, foldRight, foldLeft

def add(a: Int, b: Int): Int = a + b
val sumFunc: (Int, Int) => Int = add  // already a function
```



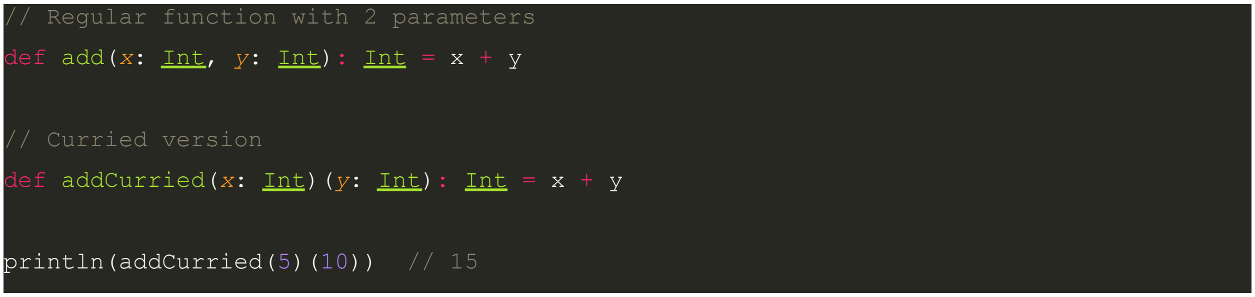
```
val sum = nums.reduce(sumFunc)        // 15

def combine(factor: Int, x: Int, y: Int): Int = factor * (x + y)
val combineWith2: (Int, Int) => Int = combine(2, _, _)
val folded = nums.reduce(combineWith2)
println(folded) // 30

// scanLeft, scanRight

def sumWithOffset(offset: Int, x: Int, y: Int): Int = offset + x + y
val f = sumWithOffset(1, _, _)
val scanned = nums.scanLeft(0)(f)
println(scanned) // List(0, 2, 5, 9, 14, 20)

//collect

def doubleIfInt(x: Any, factor: Int): Any = x match
 case i: Int => i * factor
 case _      => x

val pf: PartialFunction[Any, Any] = { case i: Int => doubleIfInt(i, 2) }
val mixed = List(1, "two", 3)
println(mixed.collect(pf)) // List(2, 6)
```

Output:

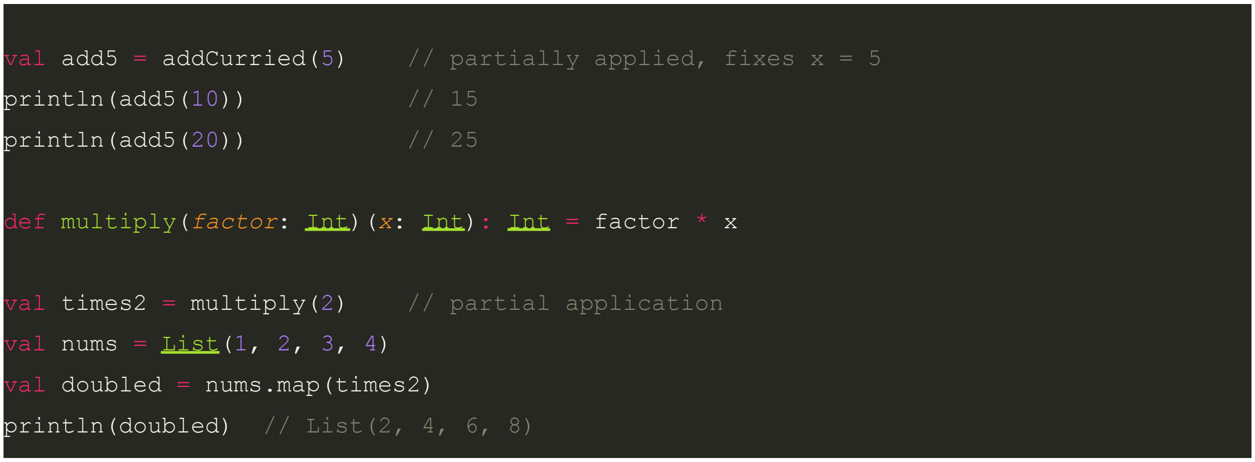


```
List(2, 4, 6, 8, 10)
List(2, 4)
98
List(0, 2, 5, 9, 14, 20)
List(2, 6)
```

9. Currying

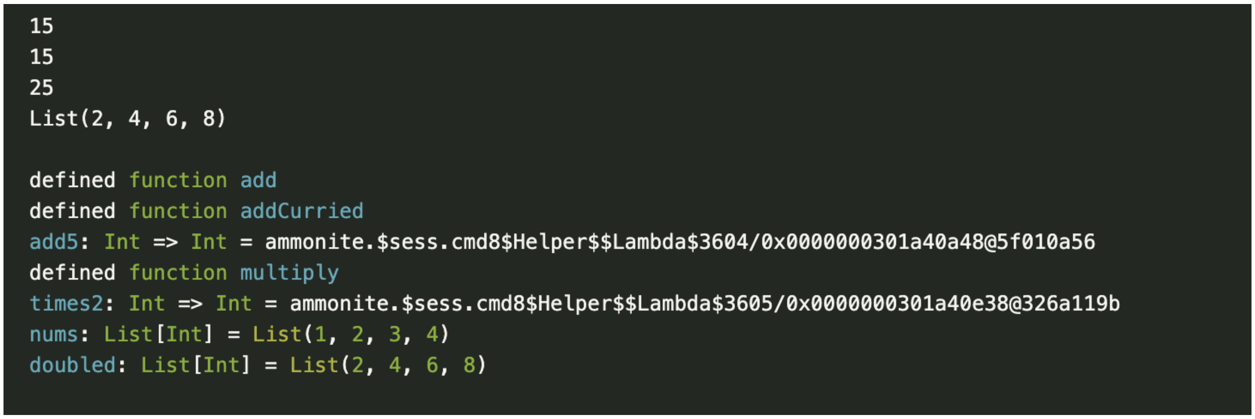


```
// Regular function with 2 parameters
def add(x: Int, y: Int): Int = x + y

// Curried version
def addCurried(x: Int)(y: Int): Int = x + y

println(addCurried(5)(10))  // 15
```



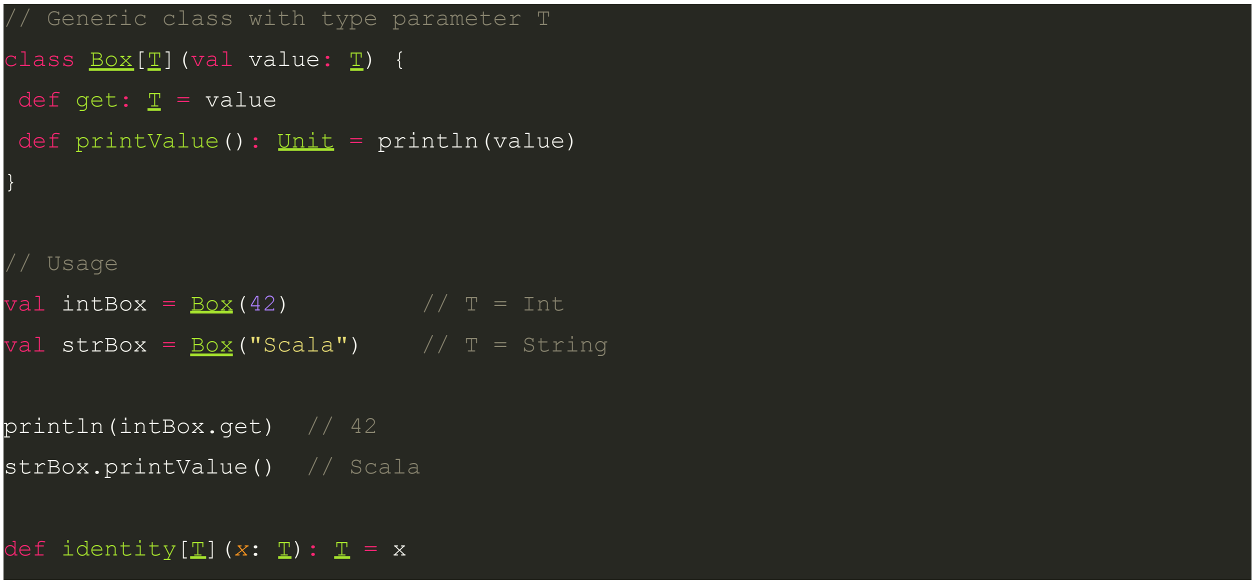
```scala
val add5 = addCurried(5)     // partially applied, fixes x = 5
println(add5(10))            // 15
println(add5(20))            // 25

def multiply(factor: Int)(x: Int): Int = factor * x

val times2 = multiply(2)     // partial application
val nums = List(1, 2, 3, 4)
val doubled = nums.map(times2)
println(doubled)   // List(2, 4, 6, 8)
```

Output:

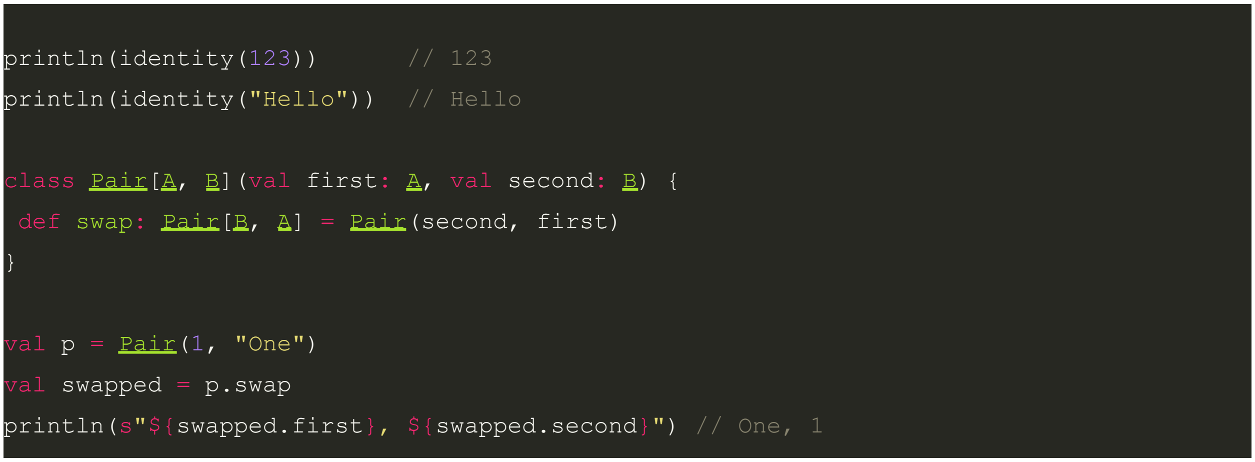


```
15
15
25
List(2, 4, 6, 8)

defined function add
defined function addCurried
add5: Int => Int = ammonite.$sess.cmd8$Helper$$Lambda$3604/0x0000000301a40a48@5f010a56
defined function multiply
times2: Int => Int = ammonite.$sess.cmd8$Helper$$Lambda$3605/0x0000000301a40e38@326a119b
nums: List[Int] = List(1, 2, 3, 4)
doubled: List[Int] = List(2, 4, 6, 8)
```

10. Generics

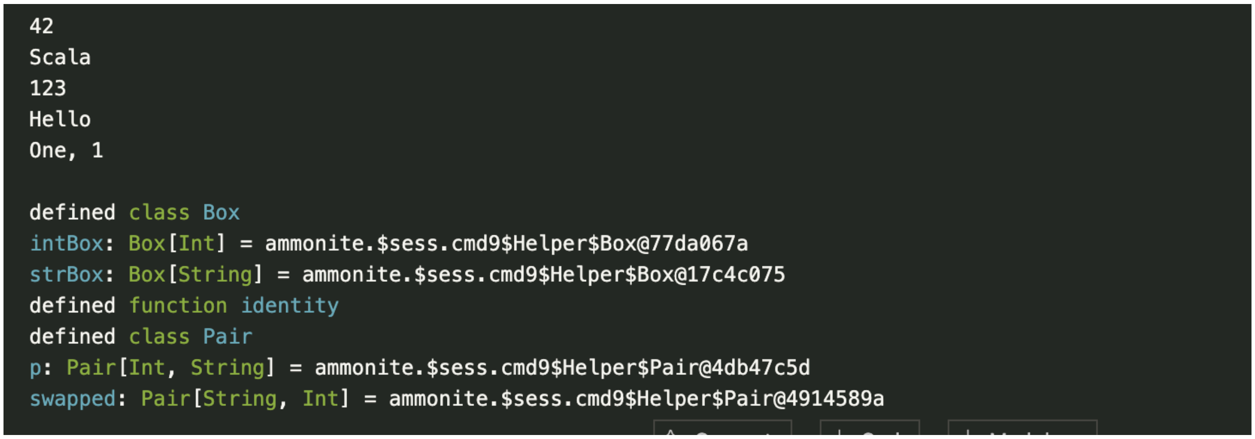


```scala
// Generic class with type parameter T
class Box[T](val value: T) {
 def get: T = value
 def printValue(): Unit = println(value)
}

// Usage
val intBox = Box(42)          // T = Int
val strBox = Box("Scala")     // T = String

println(intBox.get)   // 42
strBox.printValue()   // Scala

def identity[T](x: T): T = x
```



```scala
println(identity(123))       // 123
println(identity("Hello"))  // Hello

class Pair[A, B](val first: A, val second: B) {
 def swap: Pair[B, A] = Pair(second, first)
}

val p = Pair(1, "One")
val swapped = p.swap
println(s"${swapped.first}, ${swapped.second}") // One, 1
```

Output:



```
42
Scala
123
Hello
One, 1

defined class Box
intBox: Box[Int] = ammonite.$sess.cmd9$Helper$Box@77da067a
strBox: Box[String] = ammonite.$sess.cmd9$Helper$Box@17c4c075
defined function identity
defined class Pair
p: Pair[Int, String] = ammonite.$sess.cmd9$Helper$Pair@4db47c5d
swapped: Pair[String, Int] = ammonite.$sess.cmd9$Helper$Pair@4914589a
```