Introduction

Why Scala?

- ✓ First, a developer can achieve a significant productivity jump by using Scala.
- ✓ Second, it helps developers write robust code with reduced bugs.
- ✓ Third, Spark is written in Scala, so Scala is a natural fit for developing Spark applications.

Functional Programming

What is Functional Programming

- ✓ First, functional programming provides a tremendous boost in developer productivity.
- ✓ Second, functional programming makes it easier to write concurrent or multithreaded applications.
- ✓ Third, functional programming helps you to write robust code
  Finally, functional programming languages make it easier to write elegant code, which is easy to read, understand, and reason about.

Functions – Characteristics

First Class

FP treats functions as first-class citizens. A function has the same status as a variable or value. It allows a function to be used just like a variable. While in case of any imperative language such as C it treats function and variable differently.

Composable

Functions in functional programming are composable. Function composition is a mathematical and computer science concept of combining simple functions to create a complex one.

No Side Effects

A function in functional programming does not have side effects. The result returned by a function depends only on the input arguments to the function. The behavior of a function does not change with time. It returns the same output every time for a given input, no matter how many times it is called. In other words, a function does not have a state. It does not depend on or update any global variable.

Simple

Functions in functional programming are simple. A function consists of a few lines of code and it does only one thing. A simple function is easy to reason about. Thus, it enables robustness and high quality.

Immutable Data Structures

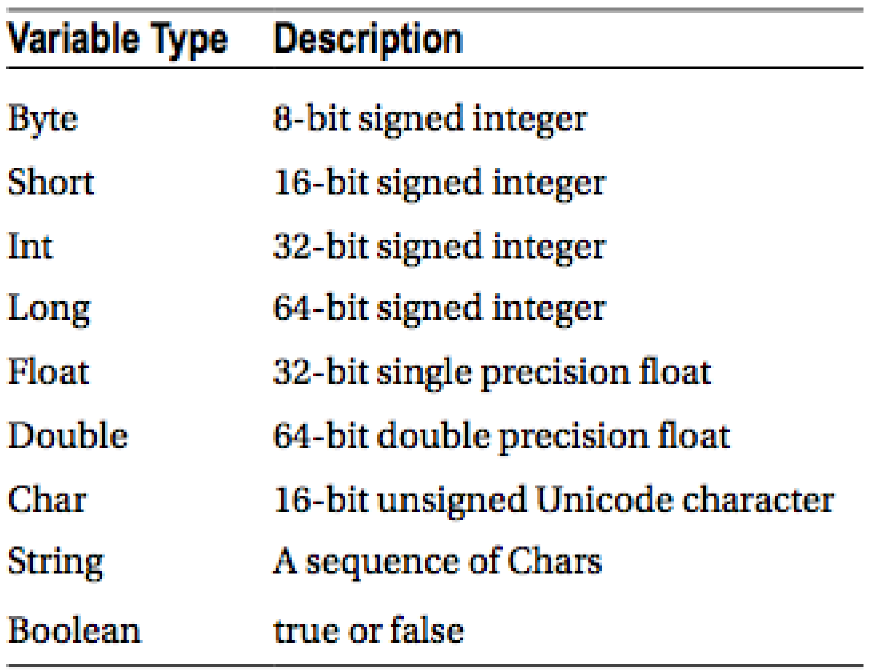
Functional programming emphasizes the usage of immutable data structures. A purely functional program does not use any mutable data structure or variable. In other words, data is never modified in place,unlike in imperative programming languages such as C/C++, Java, and Python . Immutable data structures provide a number of benefits. First, they reduce bugs. It is easy to reason about code written with immutable data structures. Second, immutable data structures make it easier to write multi-threaded applications. Writing an application that utilizes all the cores is not an easy task.

Everything Is an Expression

In functional programming, every statement is an expression that returns a value. For example, the if-else control structure in Scala is an expression that returns a value. This behavior is different from imperative languages, where you can just group a bunch of statements within if-else. This feature is useful for writing applications without mutable variables

Variables

- ✓ Statically Typed Language
- ✓ Type Inference
- ✓ Mutable – Var
- ✓ Immutable – Val
- ✓ Lazy Values
- ✓ Basic Types

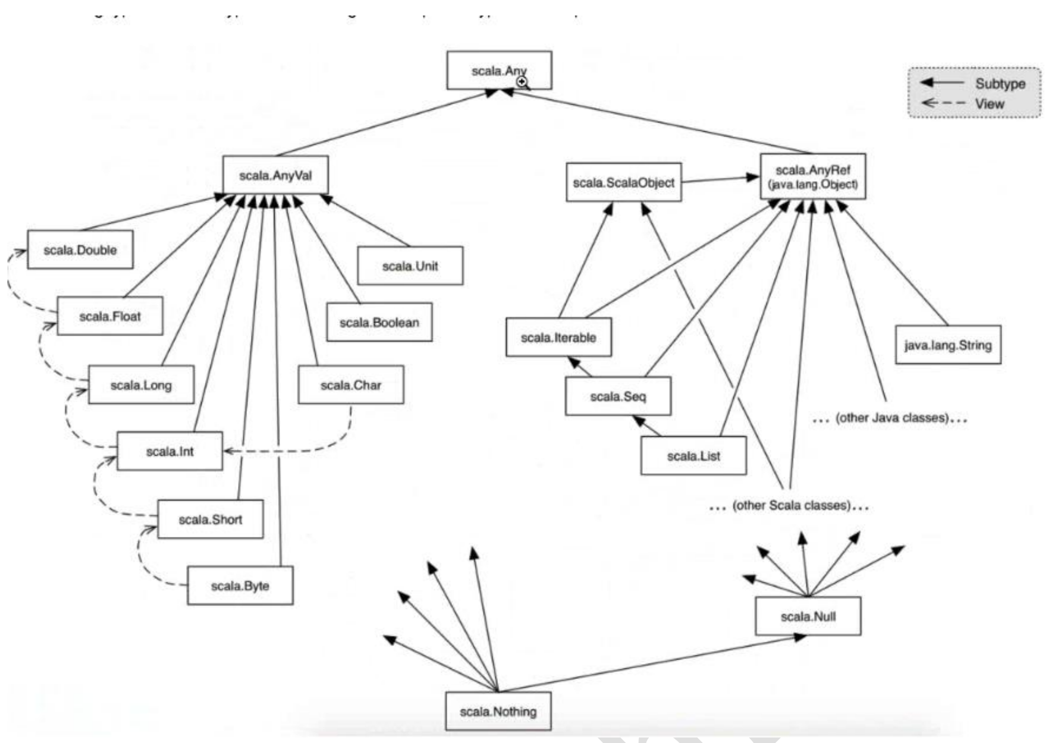


Variable Type	Description
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Float	32-bit single precision float
Double	64-bit double precision float
Char	16-bit unsigned Unicode character
String	A sequence of Chars
Boolean	true or false

Note that Scala does not have primitive types. Each type in Scala is implemented as a class. When a Scala application is compiled to Java bytecode, the compiler automatically converts the Scala types to Java's primitive types wherever possible to optimize application performance.

Control Structures

- ✓ If Else
- ✓ While
- ✓ For Loop



Functions

How to Define Function in Scala?

A function in Scala is defined with the keyword def. A function definition starts with the function name, which is followed by the comma-separated input parameters in parentheses along with their types. The closing parenthesis is followed by a colon, function output type, equal sign, and the function body in optional curly braces. An example is shown next.

```
def add(firstInput: Int, secondInput: Int): Int = {
  val sum = firstInput + secondInput
  return sum
}
```

Scala allows a concise version of the same function, as shown next.

```
def add(firstInput: Int, secondInput: Int) = firstInput + secondInput
```

The second version does the exact same thing as the first version. The type of the returned data is omitted since the compiler can infer it from the code. However, it is recommended not to omit the return type of a function.

The curly braces are also omitted in this version. They are required only if a function body consists of more than one statement.

In addition, the keyword return is omitted since it is optional. Everything in Scala is an expression that returns a value. The result of the last expression, represented by the last statement, in a function body becomes the return value of that function.

Methods

A method is a function that is a member of an object. It is defined like and works the same as a function. The only difference is that a method has access to all the fields of the object to which it belongs.

Procedures

Procedure is a special type of function with no return and having Unit type as return

Local Functions

- ✓ A function defined inside another function or method is called a *local function*. It has access to the variables and input parameters of the enclosing function.
- ✓ A local function is visible only within the function in which it is defined. This is a useful feature that allows you to group statements within a function without polluting your application’s namespace.

Higher-Order Methods

A method that takes a function as an input parameter is called a *higher-order method*. Similarly, a high-order function is a function that takes another function as input. Higher-order methods and functions help reduce code duplication. In addition, they help you write concise code.

The following example shows a simple higher-order function.

```
def encode(n: Int, f: (Int) => Long): Long = {
  val x = n * 10
  f(x)
}
```

Function Literals

- ✓ A function literal is an unnamed or anonymous function in source code. It can be used in an application just like a string literal. It can be passed as an input to a higher-order method or function. It can also be assigned to a variable.
- ✓ A function literal is defined with input parameters in parenthesis, followed by a right arrow and the body of the function. The body of a functional literal is enclosed in optional curly braces. An example is shown next.

```
(x: Int) => {
            x + 100
}
```

The higher-order function encode defined earlier can be used with a function literal, as shown next.

```
val code = encode(10, (x: Int) => x + 100)
```

Closures

- ✓ The body of a function literal typically uses only input parameters and local variables defined within the function literal. However, Scala allows a function literal to use a variable from its environment.
- ✓ A closure is a function literal that uses a non-local non-parameter variable captured from its environment. Sometimes people use the terms *function literal* and *closure* interchangeably, but technically, they are not the same.

The following code shows an example of a closure.

```
def encodeWithSeed(num: Int, seed: Int): Long = {
  def encode(x: Int, func: (Int) => Int): Long = {
    val y = x + 1000
    func(y)
}

   val result = encode(num, (n: Int) => (n * seed))
result }
```

Classes

- ✓ A class is a template or blueprint for creating objects at runtime. An object is an instance of a class.
- ✓ A class is defined in source code, whereas an object exists at runtime. A class is defined using the keyword class.
- ✓ A class definition starts with the class name, followed by comma-separated class parameters in parentheses, and then fields and methods enclosed in curly braces.

An example is shown next.

```
class Car(mk: String, ml: String, cr: String) {
  val make = mk
  val model = ml
  var color = cr
24

def repaint(newColor: String) = {
   color = newColor
} }
```

An instance of a class is created using the keyword new.

```
val mustang = new Car("Ford", "Mustang", "Red")

val corvette = new Car("GM", "Corvette", "Black")
```

A class is generally used as a mutable data structure. An object has a state, which changes with time. Therefore, a class may have fields that are defined using var.

Since Scala runs on the JVM, you do not need to explicitly delete an object. The Java garbage collector automatically removes objects that are no longer in use.

Class in Scala is very much Similar to Java or C++

```
class Ctr {

        private var value=0 //Fields must be initialized

                def incr() {value +=1}

                def curr() = value

            }
```

- ✓ In scala a class in NOT declared as public
- ✓ A source file can contain multiple classes
- ✓ All of the classes could be public

Previous class could be used in usual way :

```
val ctr1 = new Ctr            //Or new Ctr() is also valid
ctr1.incr()
print(ctr1.curr)
```

```
scala>val ctr1 = new Ctr
ctr1: Ctr = Ctr@6689ef79

scala>ctr1.incr()
scala>print(ctr1.curr)
```

Parameter less method could be called with or without parentheses
Using any form is programmer's choice
However,as convention

- Use() for mutator method
- Use no parentheses for accessor method

Properties: Getters & Setters

- ✓ Getters and Setters are better to expose class properties
- ✓ In Java, we typically keep the instance variables as private and private and expose the public getters and setters

```
public class Duck
                {
                   private int size;
                   public int getSize() {return size;}
                   public void setSize(int size)
                {
                   (if size >0) this.size = size
                 }
                }
```

Scala provides the getters and setters for every field by default

We define a public field

```
Class Duck {
              var size = 1
             }
```

Example: Scala

```
scala> class Duck {
           var size = 1
           }
defined class Duck
```

- ✓ Scala generate a class for the JVM with a private size variable and public getter and setter methods
- ✓ If the field is declared as private, the getters and setters would be private
- ✓ The getters and setter methods in previous case would be:
    - size and size_=

Example:

```
scala> var f = new Duck
f: Duck = Duck@3ec9cf38

scala> f.size = 10
f.size: Int = 10
scala> println(f.size)
10
```

Properties with Only Getters

Sometimes we need read-only properties

There are two possibilities:

- ✓ The property value never changes
- ✓ The value is changed indirectly

For the first case, we declare the property as val. Scala treats it as final variable and thus generates only getter or setter

In second case, you need to declare the field as private and provide the getter, as explained below:

Semicolons are optional in Scala:

```
scala> class Counter
    {
            private var value = 0
                def incr() {value +=1}
                def current = value
            }
defined class Counter
```

Primary Constructor

- ✓ Every class in Scala has a primary constructor
- ✓ Primary constructor isn't defined by this method

The parameters for primary constructor are placed immediately after the class name:

```
class Duck(val size:Int, val age:Int)
{
//...Code for intitialization
}
```

The Primary constructor executes all the statements in the class definition, as explained below:

```
class Duck(val size:Int, val age:Int)
{
println("Inside duck constructor")
def desc = "Duck of age" +age+ "is of size" + size
}
```

Scala Example:

```
scala>class Duck(val size:Int, val age:Int)
    {
          def desc = "Duck of age" +age+ "is of size" + size
    }
defined class Duck
```

Auxiliary Constructor

- ✓ We can have as many constructors as we need
- ✓ There are two types of constructors in Scala:
  - Auxiliary Constructor
  - Primary Constructor

- ✓ The auxiliary constructors in scala are called this. This is different from other language, where constructors have the same name as the class.

- ✓ Each auxiliary constructor must start with a call to either a previously defined auxiliary constructor or the primary constructor.

```
class Duck {
        private var size = 0;
        private var age = 0;
        def this(size:Int)(
                this()  //Calls the primary constructor
                this.size = size
                }
                def this(size:Int, age:Int)
                {
                        this(size)  //calls previous auxiliary constructor
                        this.age
                 }
            }
```

Singletons

- ✓ One of the common design patterns in object-oriented programming is to define a class that can be instantiated only once.
- ✓ A class that can be instantiated only once is called a *singleton*. Scala provides the keyword object for defining a singleton class.

```
object DatabaseConnection {
  def open(name: String): Int = {
... }

  def read (streamId: Int): Array[Byte] = {
    ...
}

  def close (): Unit = {
    ...
} }
```

Case Classes

A case class is a class with a case modifier. An example is shown next.

case class Message(from: String, to: String, content: String)

>Creates Factory method automatically

-val request = Message("harry", "sam", "fight")

>All input parameters defined implicitly treated as Val

>4 Methods: toString, hashCode, equals, and copy

>Useful for nonmutable objects and pattern matching

Pattern Matching

- ✓ Pattern matching is a Scala concept that looks similar to a switch statement in other languages.
- ✓ However, it is a more powerful tool than a switch statement. It is like a Swiss Army knife that can be used for solving a number of different problems.

One simple use of pattern matching is as a replacement for a multi-level if-else statement.

```
def f(x: Int, y: Int, operator: String): Double = {
  operator match {
    case "+" => x + y
    case "-" => x - y
    case "*" => x * y
    case "/" => x / y.toDouble
} }

val sum = f(10,20, "+")
val product = f(10, 20, "*")
```

Operators

Scala provides a rich set of operators for the basic types. However, it does not have built-in operators. In Scala, every type is a class and every operator is a method. Using an operator is equivalent to calling a method. Consider the following example.

```
val x = 10
val y = 20
val z = x + y
```

+ is not a built-in operator in Scala. It is a method defined in class Int. The last statement in the preceding code is same as the following code.

val z = x.+(y)

Traits

A *trait* represents an interface supported by a hierarchy of related classes. It is an abstraction mechanism that helps development of modular, reusable, and extensible code

A trait looks similar to an abstract class. Both can contain fields and methods. The key difference is that a class can inherit from only one class, but it can inherit from any number of traits.

An example of a trait is shown next.

```
trait Shape {
  def area(): Int
}

class Square(length: Int) extends Shape {
  def area = length * length
}

class Rectangle(length: Int, width: Int) extends Shape {
  def area = length * width
}

val square = new Square(10)
val area = square.area
```

Tuples

A *tuple* is a container for storing two or more elements of different types. It is immutable; it cannot be modified after it has been created. It has a lightweight syntax, as shown next.

```
val twoElements = ("10", true)
val threeElements =  ("10", "harry", true)
```

An element in a tuple has a one-based index. The following code sample shows the syntax for accessing elements in a tuple.

```
val first = threeElements._1
val second = threeElements._2
val third = threeElements._3
```

Option Type

An Option is a data type that indicates the presence or absence of some data. It represents optional values. It can be an instance of either a case class called Some or singleton object called None. An instance of Some can store data of any type. The None object represents absence of data.

The following code shows sample usage.

```
def colorCode(color: String): Option[Int] = {
  color match {
    case "red" => Some(1)
    case "blue" => Some(2)
    case "green" => Some(3)
    case _ => None
} }

val code = colorCode("orange")
code match {
  case Some(c) => println("code for orange is: " + c)
  case None => println("code not defined for orange")
}
```

The Option data type helps prevent null pointer exceptions. In many languages, null is used to represent absence of data.

Collections

A collection is a container data structure. It contains zero or more elements. Collections provide a higher- level abstraction for working with data. They enable declarative programming. With an easy-to-use interface, they eliminate the need to manually iterate or loop through all the elements.

Categories:

- ✓ Sequences
- ✓ Maps
- ✓ Sets

Sequences

A *sequence* represents a collection of elements in a specific order. Since the elements have a defined order, they can be accessed by their position in a collection. For example, you can ask for the *nth* element in a sequence.

Array

An *Array* is an indexed sequence of elements. All the elements in an array are of the same type. It is a mutable data structure; you can update an element in an array. However, you cannot add an element to an array after it has been created. It has a fixed length.

```
val arr = Array(10, 20, 30, 40)
arr(0) = 50
val first = arr(0)
```

>Fetching by Index
>Updating by Index

Array Buffer

- ✓ Variable Length Arrays(Array Buffers)
- ✓ Similar to Java ArrayLists

```
  import scala.collection.mutable.ArrayBuffer
  val a= ArrayBuffer[Int]()
  a+=1
  a+=(2,3,5)
  a++=Array(6,7,8)
```

```
scala>import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala>   val a= ArrayBuffer[Int]()
a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> a +=1
res16: a.type = ArrayBuffer(1)

scala>a+= (2,3,5)
res17: a.type = ArrayBuffer(1,2,3,5)

scala> a++=Array(6,7,8)
res18: a.type = ArrayBuffer(1,2,3,5,6,7,8)
```

Common Operations:

```
a.trimEnd(2) //Removes last 2 elements
```

```
a.insert(2,9) //Adds element at 2nd index
a.insert(2,10,11,12)  //Adds a list
a.remove(2)  //Removes an element
a.remove(2,3)  //Removes three elements from index 2
```

Traversing and Transformation:

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

scala>for(el<-a)
     print(el)
1
2
3
5
6

scala>for(el<-a if el%2 ==0) yield (2*el)
res19: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(4,12)
```

Tuples

Scala tuple combines a fixed number of items together so that they can be passed around as a whole. Unlike an array or list, a tuple can hold objects with different types but they are also immutable.

The following is an example of a tuple holding an integer, a string, and the console.

```
val t = (1, "hello", Console)
```

To access elements of a tuple t, you can use method t._1 to access the first element, t._2 to access the second, and so on. For example, the following expression computes the sum of all elements of t.

```
val sum = t._1 + t._2 + t._3 + t._4
```

List

A *List* is a linear sequence of elements of the same type. It is a recursive data structure, unlike an array, which is a flat data structure. In addition, unlike an array, it is an immutable data structure; it cannot be modified after it has been created. List is one of the most commonly used data structures in Scala and other functional languages.

. **val xs = List(10,20,30,40)**
. **val ys = (1 to 100).toList**
. **val zs = someArray.toList**

```
>Head
>tail
IsEmpty
```

Vector

The Vector class is a hybrid of the List and Array classes. It combines the performance characteristics of both Array and List. It provides constant-time indexed access and constant-time linear access. It allows both fast random access and fast functional updates.

An example is shown next.

val v1 = Vector(0, 10, 20, 30, 40)
val v2 = v1 :+ 50
val v3 = v2 :+ 60
val v4 = v3(4)
val v5 = v3(5)

Sets

Set is an unordered collection of distinct elements. It does not contain duplicates. In addition, you cannot access an element by its index, since it does not have one.

An example of a set is shown next.

val fruits = Set("apple", "orange", "pear", "banana")
Sets support two basic operations.

. contains: Returns true if a set contains the element passed as input to this method. (

. isEmpty: Returns true if a set is empty.

. vas s = Set(1, 2, 3)

Map

(*Map* is a collection of key-value pairs. In other languages, it known as a dictionary, associative array, or hash map. It is an efficient data structure for looking up a value by its key. It should not be confused with the map in Hadoop MapReduce. That map refers to an operation on a collection. (The following code snippet shows how to create and use a Map. (

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

. val indiaCapital = capitals("India")

merge Scala sequential collections (List, Vector, ArrayBuffer, Array, Seq)

- Use the ++= method to merge a sequence into a mutable sequence
- Use the ++ method to merge two mutable or immutable sequences
- Use collection methods like union, diff, and intersect

The ++= method

Use the ++= method to merge a sequence (any TraversableOnce) into a mutable collection like an ArrayBuffer:

scala> **val a = collection.mutable.ArrayBuffer(1,2,3)**

a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)

scala> **a ++= Seq(4,5,6)**

res0: a.type = ArrayBuffer(1, 2, 3, 4, 5, 6)

The ++ method

Use the ++ method to merge two mutable or immutable collections while assigning the result to a new variable:

scala> **val a = Array(1,2,3)** a: Array[Int] = Array(1, 2, 3)

scala> **val b = Array(4,5,6)** b: Array[Int] = Array(4, 5, 6) scala> **val c = a ++ b** c: Array[Int] = Array(1, 2, 3, 4, 5, 6)

union, intersect

You can also use methods like union and intersect to combine sequences to create a resulting sequence:

scala> **val a = Array(1,2,3,4,5)** a: Array[Int] = Array(1, 2, 3, 4, 5)

scala> **val b = Array(4,5,6,7,8)** b: Array[Int] = Array(4, 5, 6, 7, 8)

// elements that are in both collections

scala> **val c = a.intersect(b)** c: Array[Int] = Array(4, 5)

// all elements from both collections

scala> **val c = a.union(b)** c: Array[Int] = Array(1, 2, 3, 4, 5, 4, 5, 6, 7, 8)

// distinct elements from both collections

scala> **val c = a.union(b).distinct** c: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)

diff

The diff method results depend on which sequence it's called on:

scala> **val c = a diff b** c: Array[Int] = Array(1, 2, 3)

scala> **val c = b diff a** c: Array[Int] = Array(6, 7, 8)

The Scaladoc for the diff method states that it returns, "a new list which contains all elements of this list except some of occurrences of elements that also appear in that. If an element value x appears n times in that, then the first n occurrences of x will not form part of the result, but any following occurrences will."

The objects that correspond to most collections also have a concatmethod:

scala> **Array.concat(a, b)** res0: Array[Int] = Array(1, 2, 3, 4, 4, 5, 6, 7)

::: with List

If you happen to be working with a List, the ::: method prepends the elements of one list to another list:

scala> **val a = List(1,2,3,4)** a: List[Int] = List(1, 2, 3, 4)
scala> **val b = List(4,5,6,7)** b: List[Int] = List(4, 5, 6, 7)
scala> **val c = a ::: b** c: List[Int] = List(1, 2, 3, 4, 4, 5, 6, 7)

Higher-Order Methods on Collection Classes

The real power of Scala collections comes from their higher-order methods. A higher-order method takes a function as its input parameter. It is important to note that a higher-order method does not mutate a collection. This section discusses some of the most commonly used higher methods. The List collection is used in the examples, but all Scala collections support these higher-order methods.

map

The map method of a Scala collection applies its input function to all the elements in the collection and returns another collection. The returned collection has the exact same number of elements as the collection on which map was called. However, the elements in the returned collection need not be of the same type as that in the original collection.

An example is shown next.

val xs = List(1, 2, 3, 4)
val ys = xs.map((x: Int) => x * 10.0)

flatMap

The flatMap method of a Scala collection is similar to map. It takes a function as input, applies it to each element in a collection, and returns another collection as a result. However, the function passed to flatMap generates a collection for each element in the original collection. Thus, the result of applying the input function is a collection of collections. If the same input function were passed to the map method, it would return a collection of collections. The flatMap method instead returns a flattened collection.

The following example illustrates a use of flatMap.

val line = "Scala is fun"

```
val SingleSpace = " "
val words = line.split(SingleSpace)
val arrayOfChars = words flatMap {_.toList}
```

The toList method of a collection creates a list of all the elements in the collection. It is a useful method for converting a string, an array, a set, or any other collection type to a list.

filter

The filter method applies a predicate to each element in a collection and returns another collection consisting of only those elements for which the predicate returned true. A predicate is function that returns a Boolean value. It returns either true or false.

```
val xs = (1 to 100).toList
val even = xs filter {_ %2 == 0}
```

foreach

The foreach method of a Scala collection calls its input function on each element of the collection, but does not return anything. It is similar to the map method. The only difference between the two methods is that map returns a collection and foreach does not return anything. It is a rare method that is used for its side effects.

```
val words = "Scala is fun".split(" ")
words.foreach(println)
```

reduce

The reduce method returns a single value. As the name implies, it reduces a collection to a single value. The input function to the reduce method takes two inputs at a time and returns one value. Essentially, the input function is a binary operator that must be both associative and commutative.

The following code shows some examples.

```
val xs = List(2, 4, 6, 8, 10)
val sum  = xs reduce {(x,y) => x + y}
val product  = xs reduce {(x,y) => x * y}
val max = xs reduce {(x,y) => if (x > y) x else y}
val min = xs reduce {(x,y) => if (x < y) x else y}
```

Here is another example that finds the longest word in a sentence

```
Val words = "Scala is fun" split(" ")
val longestWord=words reduce{(w1, w2) => if(w1.length >w2.length) w1 else w2}
```