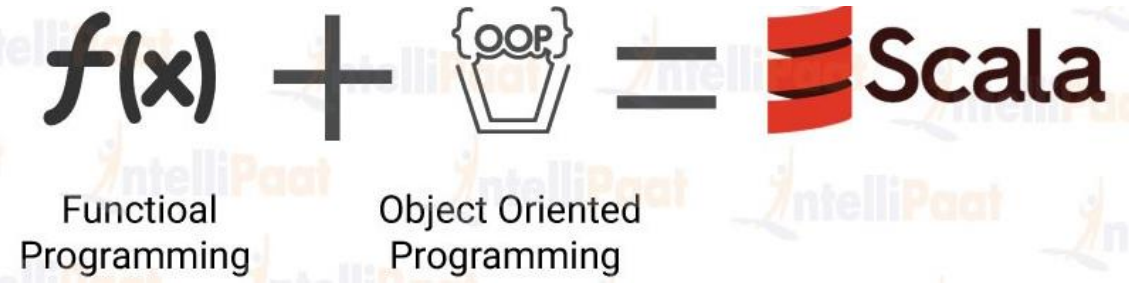
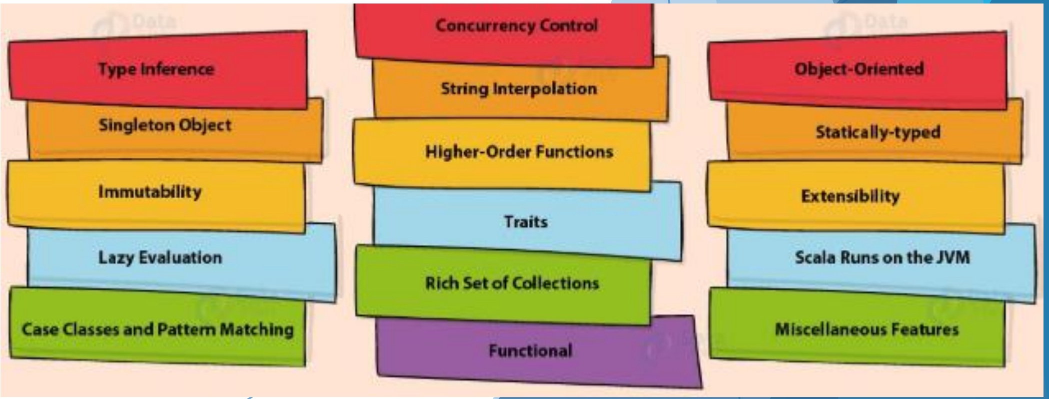
Scala for Spark

Introduction to Scala

- Scala is an acronym for “Scalable Language”
- Martin Osersky and his team started developing in 2001 and publically released in 2004

- Scala is a modern and multi-paradigm general purpose programming language
- Scala is developed as an object-oriented and functional programming language
- Scala is statically typed, being empowered with an expressive type system.
- Runs on the JVM
- Scala can execute Java code
- Concurrent and Synchronized processing is supported in Scala





Scala Features

Scala is object oriented:

- Scala is a pure object oriented language in the sense that every value is an object.
- Types and behavior of objects are described by classes and traits

Scala is functional:

- Scala is also a functional language in the sense that every function is a value and because every value is an object so ultimately every function is an object.
- Scala provides a lightweight syntax for defining anonymous functions, it supports higher order functions, it allows functions to be nested, and supports currying.

Scala is statically typed:

- Scala, unlike some of the other statically typed languages, does not expect you to provide redundant type information.
- You don't have to specify a type in most cases, and you certainly don't have to repeat it.

Scala runs on the JVM:

- Scala is compiled into Java Byte Code, which is executed by the Java Virtual Machine (JVM). This means that Scala and Java have a common run time platform.
- You can easily move from Java to Scala.

Scala Environment - Setup

- Scala can be installed on any UNIX flavored or Windows based system
- Java 1.8 or greater should be installed prior

STEP 1 : JAVA SETUP

- Set the JAVA_HOME environment variable and add the JDK's bin directory to your PATH variable
- To verify, type java –version and javac –version. It should show the version

STEP (2): SCALA SETUP:

- Next, you can download latest version of Scala from http://www.scala-lang.org/downloads
- C:\\> java jar < “latest scala installer.jar”>
- Above command will display an installation wizard, which will guide you to install scala on your windows machine.
- Open a new command prompt and type scala -version

Variables in Scala

- Variables are nothing but reserved memory locations to store values.
- This means that when you create a variable, you reserve some space in memory.

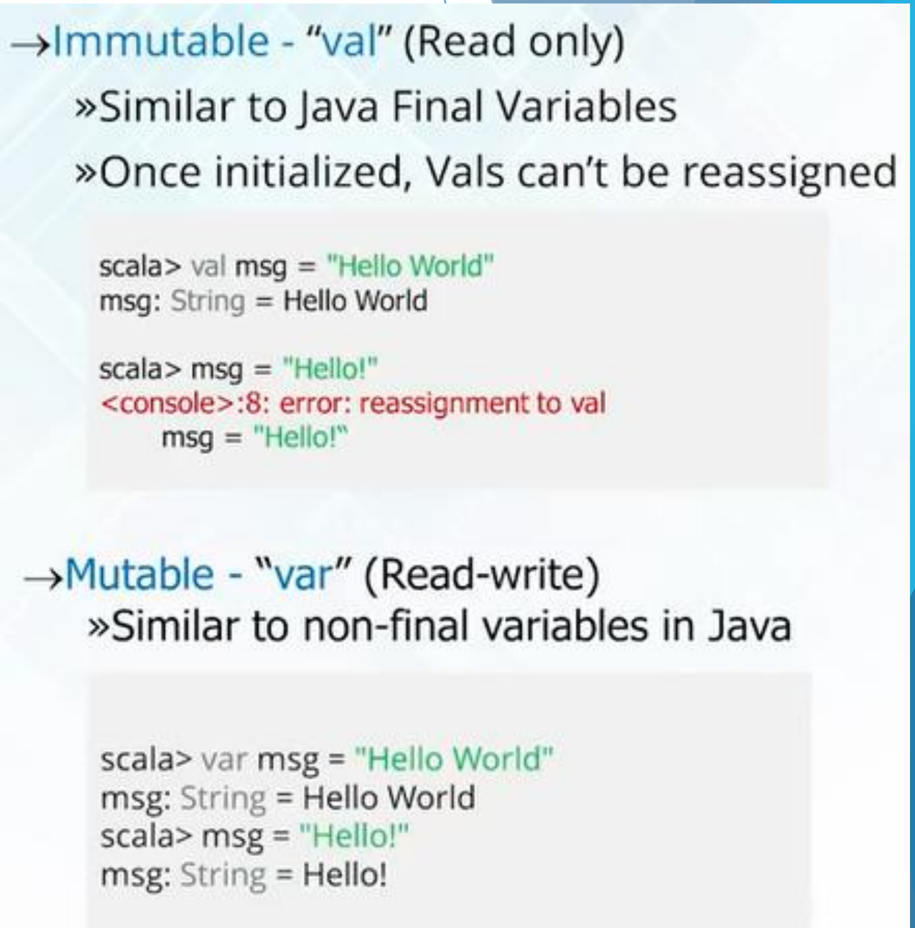
- Scala has two kinds of variables, vals and vars

Val:

- A val is similar to a final variable in Java.
- Once initialized, a val can never be reassigned
- It immutable variable

Var:

- A var is similar to a non final variable in Java.
- A var can be reassigned throughout its lifetime.
- It is mutable variable



→Immutable - "val" (Read only)

»Similar to Java Final Variables

»Once initialized, Vals can't be reassigned

```
scala> val msg = "Hello World"
msg: String = Hello World

scala> msg = "Hello!"
<console>:8: error: reassignment to val
       msg = "Hello!"
```

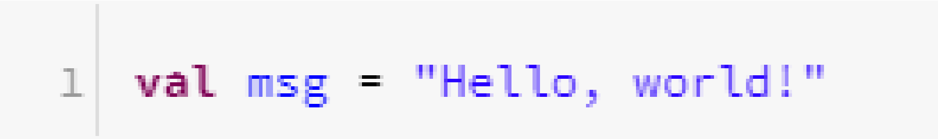
→Mutable - "var" (Read-write)

»Similar to non-final variables in Java

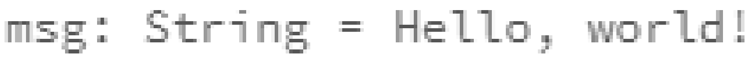
```
scala> var msg = "Hello World"
msg: String = Hello World
scala> msg = "Hello!"
msg: String = Hello!
```

Type Inference

- When we assign an initial value to a variable, the compiler infers its type based on the types of the subexpressions and the literals.
- This means that we don’t always have to declare the type of a variable.
- Once a type is assigned to a variable, it remains same for entire scope.
- Thus, Scala is statically Typed language



```scala
val msg = "Hello, world!"
```



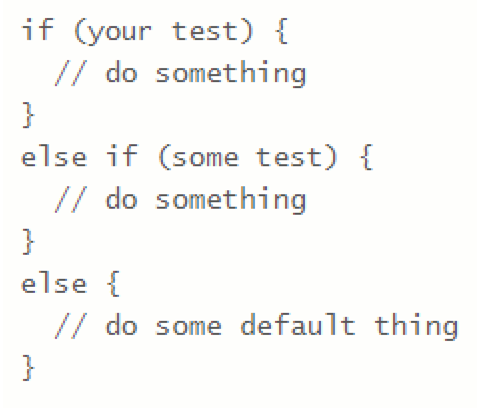
```
msg: String = Hello, world!
```

Control Structures - If Expression

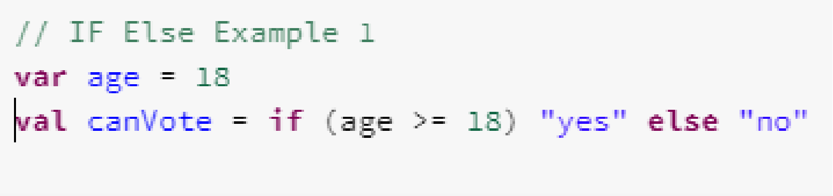
- It tests a condition and then executes one of two code branches depending on whether the condition holds true.
- An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements, there are few points to keep in mind.

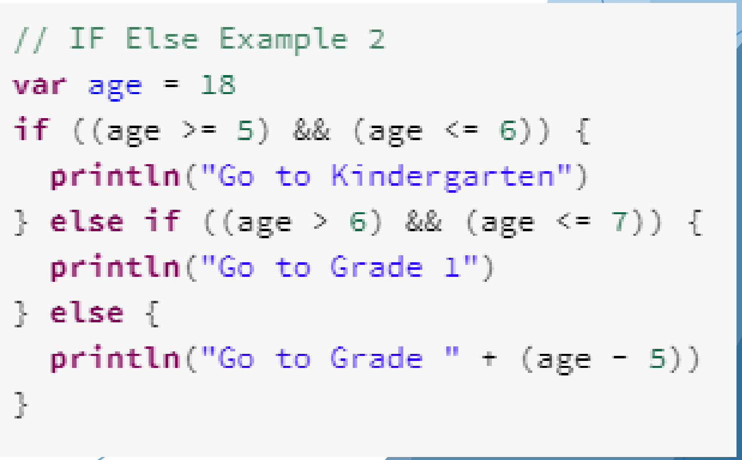
- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.



```
if (your test) {
  // do something
}
else if (some test) {
  // do something
}
else {
  // do some default thing
}
```



```
// IF Else Example 1
var age = 18
val canVote = if (age >= 18) "yes" else "no"
```



```
// IF Else Example 2
var age = 18
if ((age >= 5) && (age <= 6)) {
  println("Go to Kindergarten")
} else if ((age > 6) && (age <= 7)) {
  println("Go to Grade 1")
} else {
  println("Go to Grade " + (age - 5))
}
```

Loops

- A loop statement allows us to execute a statement or group of statements multiple times

while loop

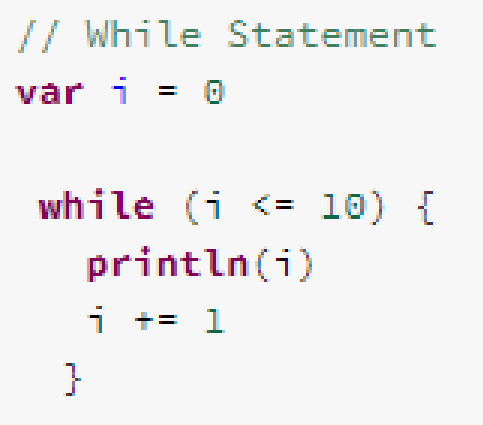
- Repeats a statement or group of statements while a given condition is true.
- It tests the condition before executing the loop body.

do-while loop

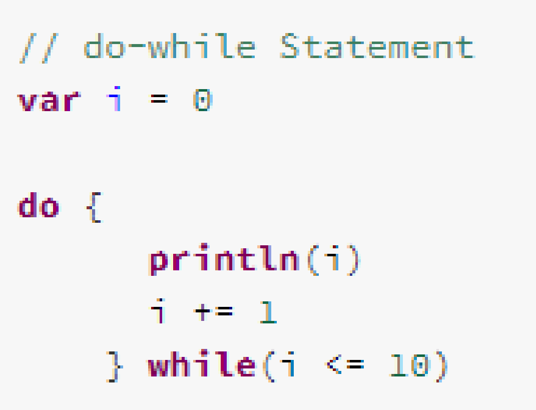
- Like a while statement, except that it tests the condition at the end of the loop body.

for loop

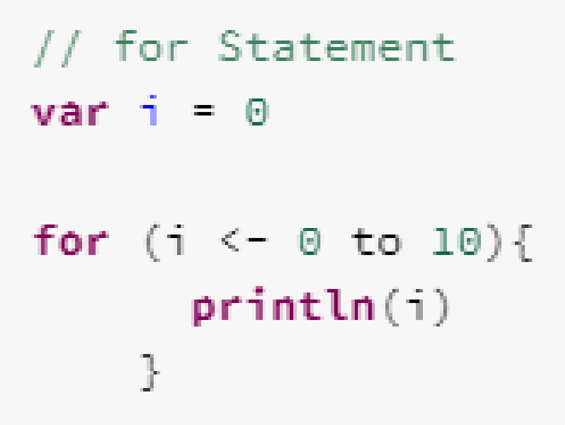
- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.



```
// While Statement
var i = 0

 while (i <= 10) {
   println(i)
   i += 1
  }
```



```
// do-while Statement
var i = 0

do {
      println(i)
      i += 1
    } while(i <= 10)
```



```
// for Statement
var i = 0

for (i <- 0 to 10){
      println(i)
    }
```

Reading from Terminal

- To read input from the terminal, there are a number of readXXX-methods.

- The methods are part of the scala.io.StdIn object, so you have to import them before you can use them
- import scala.io.StdIn.{readLine,readInt}

- readLine() reads a line and returns it as a string
- readInt() reads a line and returns it as an integer
- readDouble() reads a line and returns a floating point number
- readBoolean() reads a line and returns a Boolean ("yes", "y", "true", and "t" for true, and anything else for false);
- readChar() reads a line and returns the first character.

Scala Script

- A script is just a sequence of statements in a file that will be executed sequentially.

- Put this into a file named hello.scala

```
println("Hello, world, from a script!")
```

- To run the script:

```
$ scala hello.scala
```

- Output:

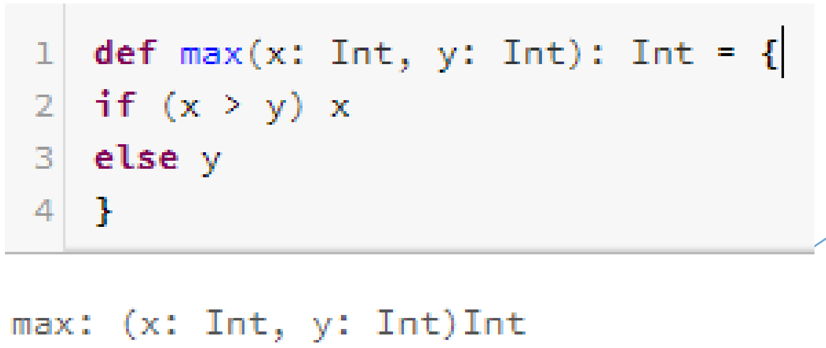
```
Hello, world, from a script!
```

Functions

- A function is a group of statements that perform a task.
- You can divide up your code into separate functions.

- Function definitions start with def followed by function’s name with by a comma separated list of parameters in parentheses and return type after parantheses
- A type annotation must follow every function parameter, preceded by a colon, because the Scala compiler does not infer function parameter types.
- Following the function's result type is an equals sign and pair of curly braces that contain the body of the function.

Syntax:
def **functionName**(parameters:typeofparameters):returntypeoffunction={
//statements to be executed
}



```
def max(x: Int, y: Int): Int = {
if (x > y) x
else y
}
```

```
max: (x: Int, y: Int)Int
```

Scala Collections

- Collections are containers of things.
- Lazy collections have elements that may not consume memory until they are accessed
- Collections may be mutable or immutable
- Immutable collections may contain mutable items.

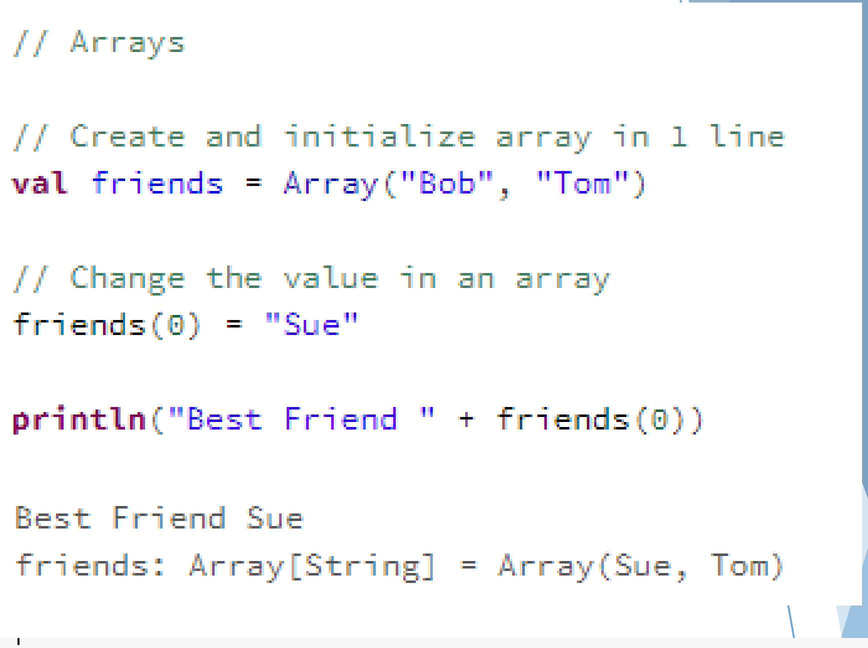
Collection Types:

- Arrays
- Lists
- Sets
- Maps
- Tuples
- Options

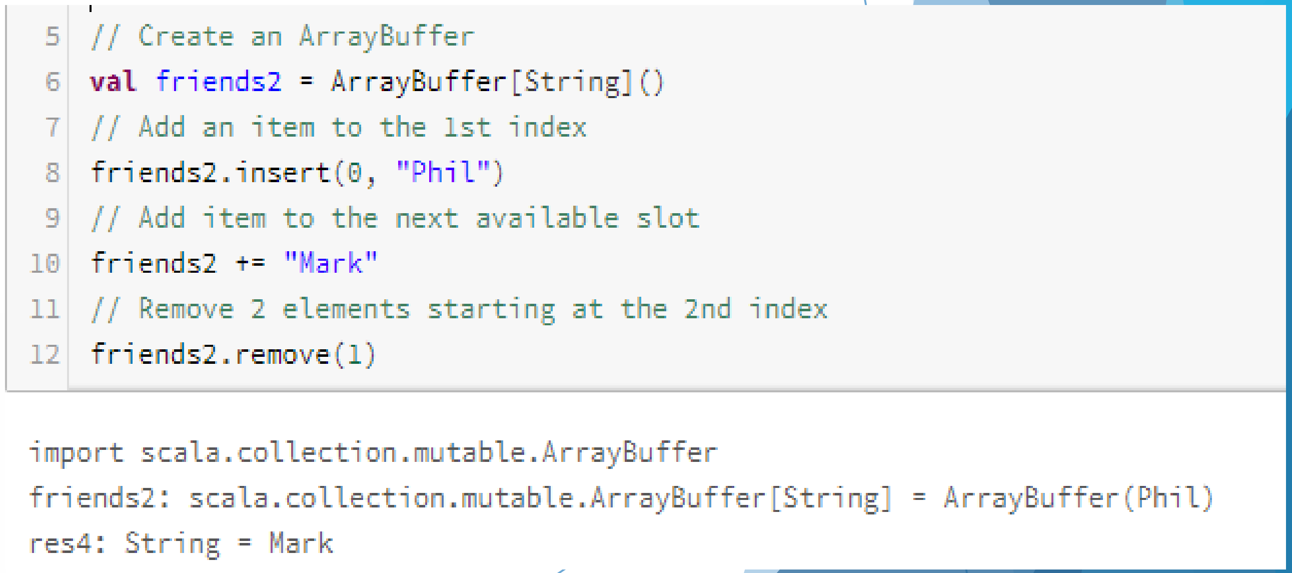
Arrays

- Arrays – Fixed Size
- ArrayBuffer – Variable Size

- It has elements of same type
- Ex: Array[String] contains only strings
- Arrays in Scala are accessed by placing the index inside parentheses, not square brackets as in Java.
- zeroth element of the array a(0)
- Although you can't change the length of an array after it is instantiated, you can change its element values. Thus, arrays are mutable objects.
- Keyword new is not required before the ArrayBuffer

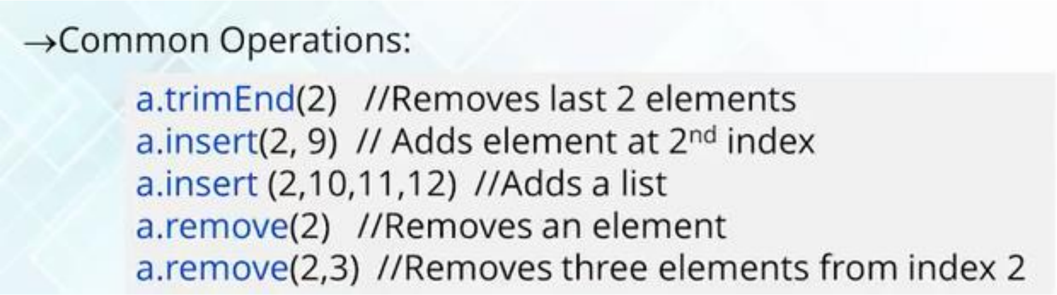


```
// Arrays

// Create and initialize array in 1 line
val friends = Array("Bob", "Tom")

// Change the value in an array
friends(0) = "Sue"

println("Best Friend " + friends(0))

Best Friend Sue
friends: Array[String] = Array(Sue, Tom)
```



```
// Create an ArrayBuffer
val friends2 = ArrayBuffer[String]()
// Add an item to the 1st index
friends2.insert(0, "Phil")
// Add item to the next available slot
friends2 += "Mark"
// Remove 2 elements starting at the 2nd index
friends2.remove(1)
```

```
import scala.collection.mutable.ArrayBuffer
friends2: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(Phil)
res4: String = Mark
```



→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
```

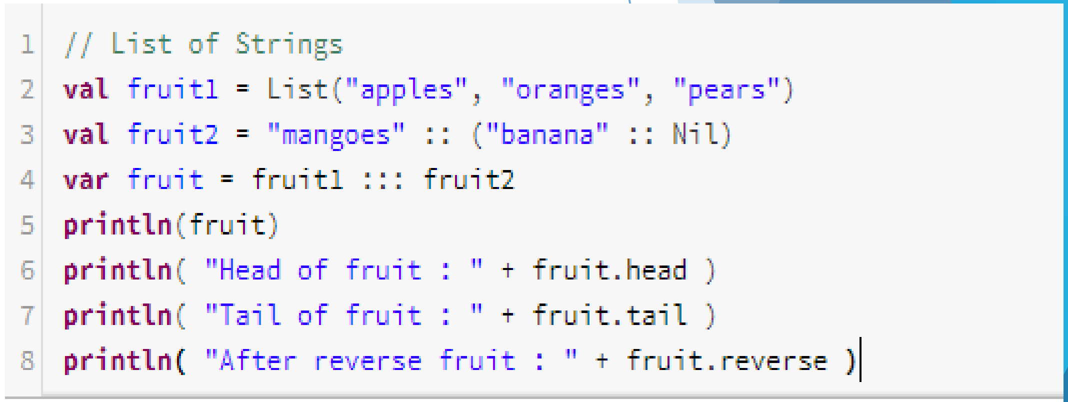
Lists

- Scala Lists are quite similar to arrays, all the elements of a list have the same type
- Lists are immutable, which means elements of a list cannot be changed by assignment

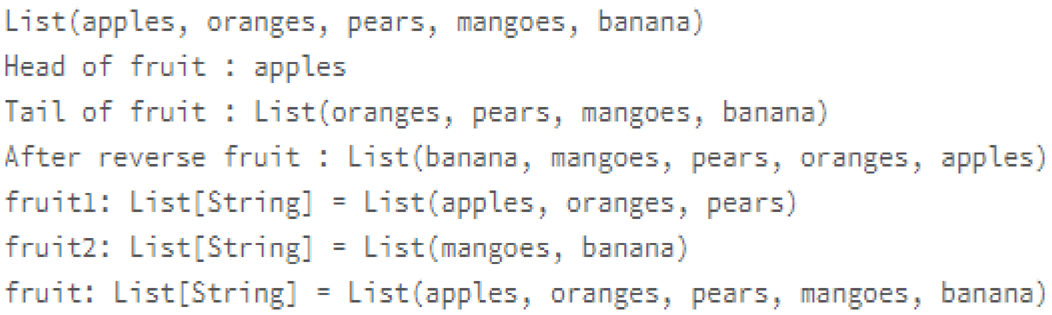
- head - This method returns the first element of a list.
- tail- This method returns a list consisting of all elements except the first.
- isEmpty -This method returns true if the list is empty otherwise false.

Concatenating Lists :

- You can use either ::: operator or List.:::() method or List.concat() method to add two or more lists
- List.tabulate() method to apply on all the elements of the list. Tabulate method, which can be used to create and populate a List



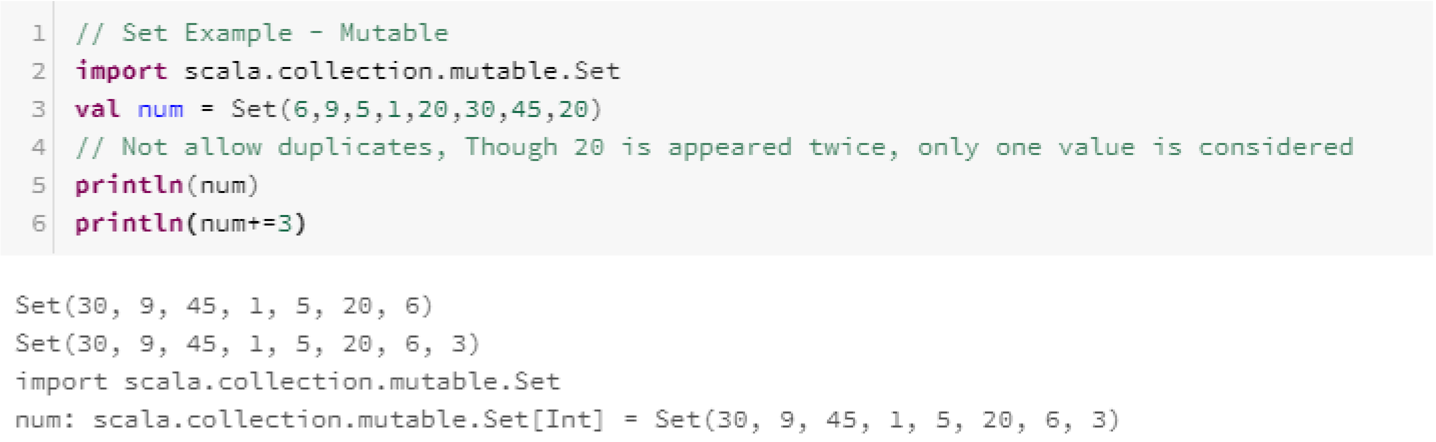
```scala
// List of Strings
val fruit1 = List("apples", "oranges", "pears")
val fruit2 = "mangoes" :: ("banana" :: Nil)
var fruit = fruit1 ::: fruit2
println(fruit)
println( "Head of fruit : " + fruit.head )
println( "Tail of fruit : " + fruit.tail )
println( "After reverse fruit : " + fruit.reverse )
```



```
List(apples, oranges, pears, mangoes, banana)
Head of fruit : apples
Tail of fruit : List(oranges, pears, mangoes, banana)
After reverse fruit : List(banana, mangoes, pears, oranges, apples)
fruit1: List[String] = List(apples, oranges, pears)
fruit2: List[String] = List(mangoes, banana)
fruit: List[String] = List(apples, oranges, pears, mangoes, banana)
```

Sets

- Scala Set is a collection of elements of the same type
- Set is unordered and can't have duplicate items
- There are two kinds of Sets, the immutable and the mutable.
- By default, Scala uses the immutable Set.
- If you want to use the mutable Set, you'll have to import scala.collection.mutable.Set class explicitly.



```scala
// Set Example - Mutable
import scala.collection.mutable.Set
val num = Set(6,9,5,1,20,30,45,20)
// Not allow duplicates, Though 20 is appeared twice, only one value is considered
println(num)
println(num+=3)
```

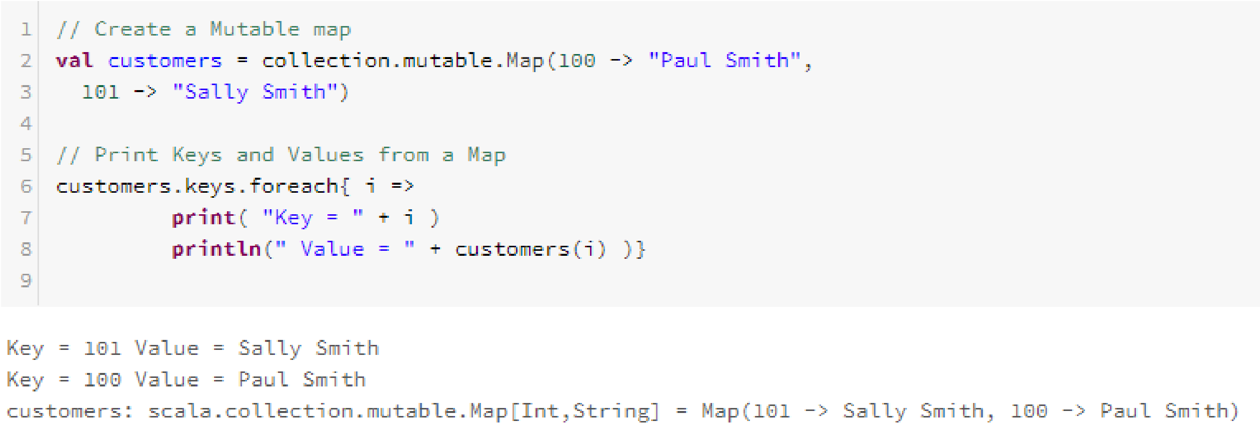
```
Set(30, 9, 45, 1, 5, 20, 6)
Set(30, 9, 45, 1, 5, 20, 6, 3)
import scala.collection.mutable.Set
num: scala.collection.mutable.Set[Int] = Set(30, 9, 45, 1, 5, 20, 6, 3)
```

Maps

- Maps are collections of key value pairs.
- Any value can be retrieved based on its key.
- Keys are unique in the Map, but values need not be unique
- Maps are also called Hash tables
- There are two kinds of Maps, the immutable and the mutable

- By default, Scala uses the immutable Map
- If you want to use the mutable Map, you'll have to import scala.collection.mutable.Map class explicitly

- We can use a foreach loop to walk through the keys and values of a Scala Map

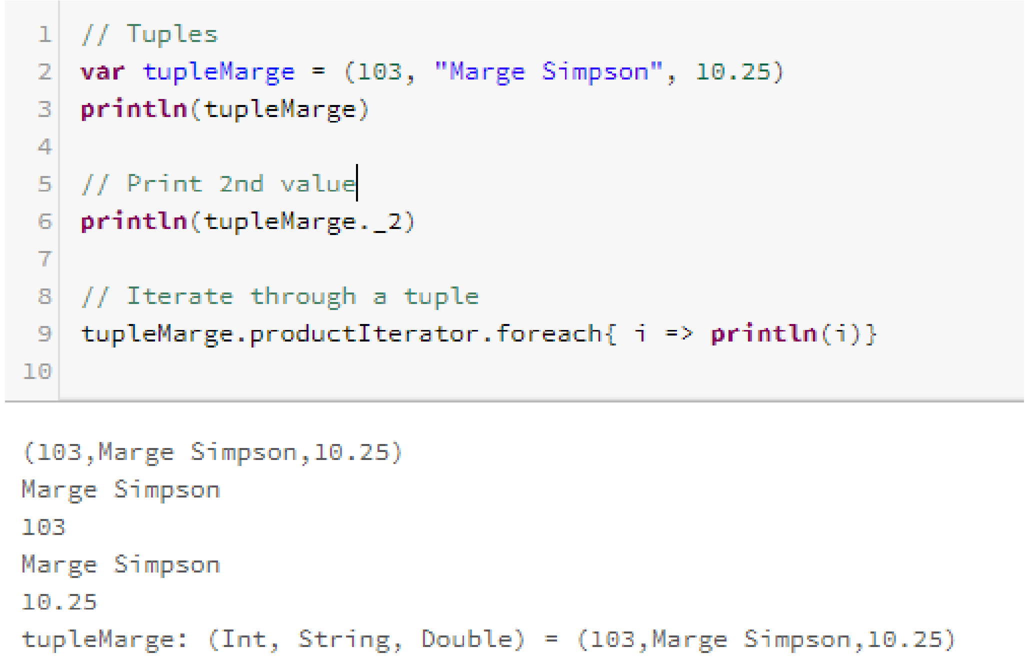


```scala
// Create a Mutable map
val customers = collection.mutable.Map(100 -> "Paul Smith",
  101 -> "Sally Smith")

// Print Keys and Values from a Map
customers.keys.foreach{ i =>
         print( "Key = " + i )
         println(" Value = " + customers(i) )}

```

```
Key = 101 Value = Sally Smith
Key = 100 Value = Paul Smith
customers: scala.collection.mutable.Map[Int,String] = Map(101 -> Sally Smith, 100 -> Paul Smith)
```

Tuples

- Like lists, tuples are immutable, but unlike lists, tuples can contain different types of elements.
- A List might be a List[Int] or a List[String], a tuple could contain both an integer and a string at the same time.
- You access the _1 field, which will produce the first element

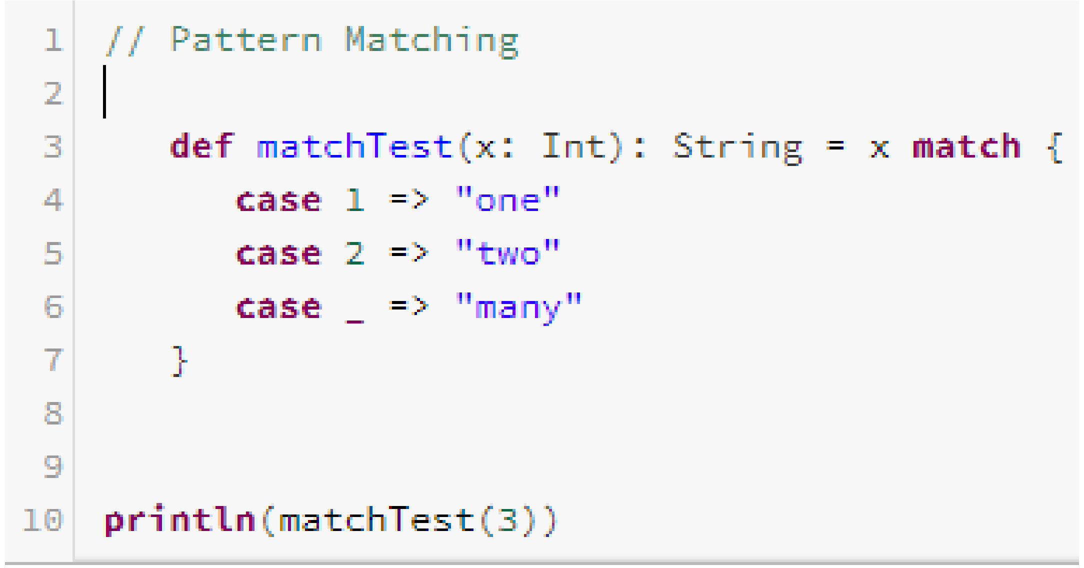


```
// Tuples
var tupleMarge = (103, "Marge Simpson", 10.25)
println(tupleMarge)

// Print 2nd value
println(tupleMarge._2)

// Iterate through a tuple
tupleMarge.productIterator.foreach{ i => println(i)}

```

```
(103,Marge Simpson,10.25)
Marge Simpson
103
Marge Simpson
10.25
tupleMarge: (Int, String, Double) = (103,Marge Simpson,10.25)
```

Pattern Matching

- Pattern matching is a way of checking the given sequence of tokens for the presence of the specific pattern
- It is similar to the switch statement of Java and C.
- Here, “match” keyword is used instead of switch statement
- To separate the pattern from the expressions, arrow symbol(=>) is used.

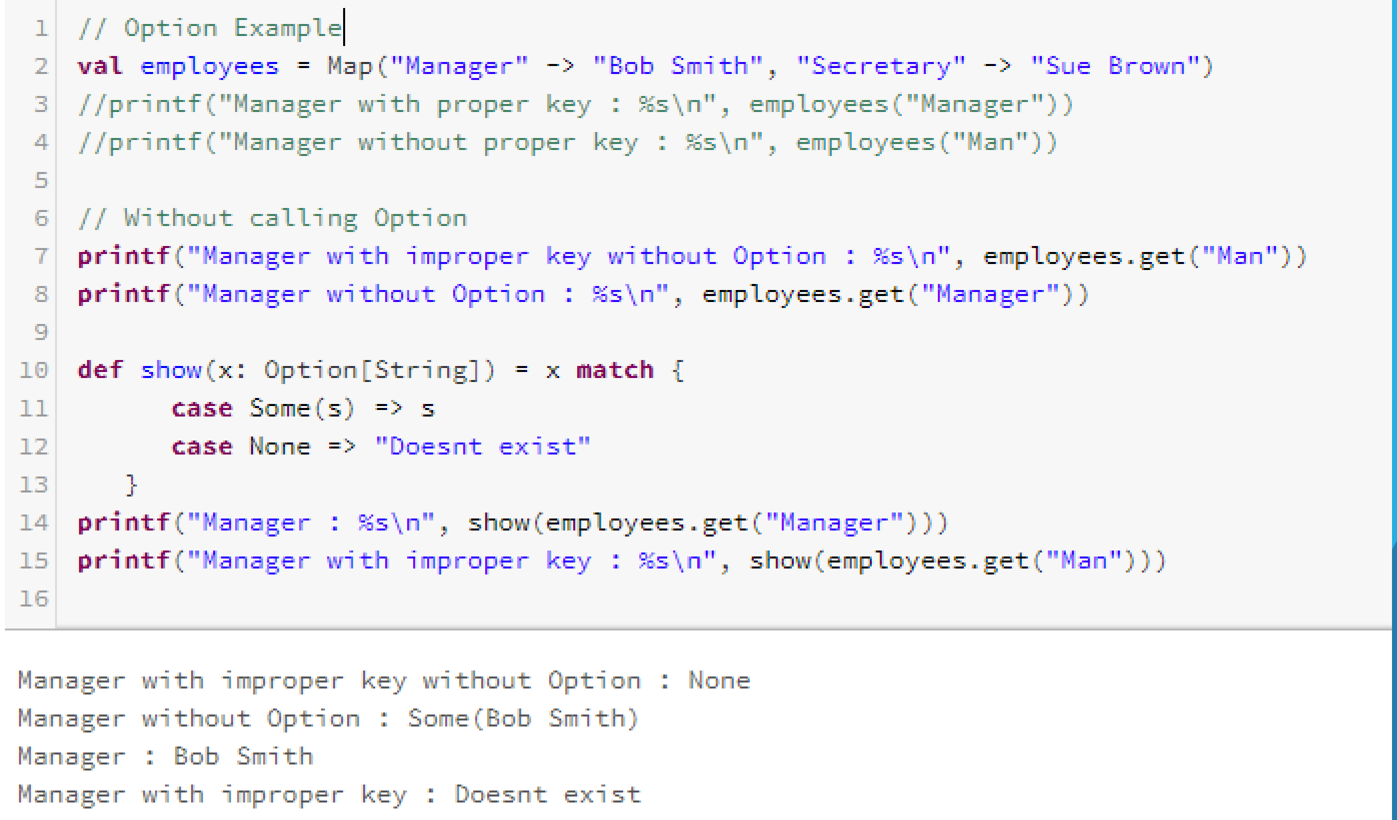


```
// Pattern Matching

   def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
   }


println(matchTest(3))
```

```
many
```

Options

- Scala Option[T] is a container for zero or one element of a given type.
- An Option[T] can be either Some[T] or None object, which represents a missing value.
- For instance, the get method of Scala's Map produces Some(value) if a value corresponding to a given key has been found, or None if the given key is not defined in the Map.

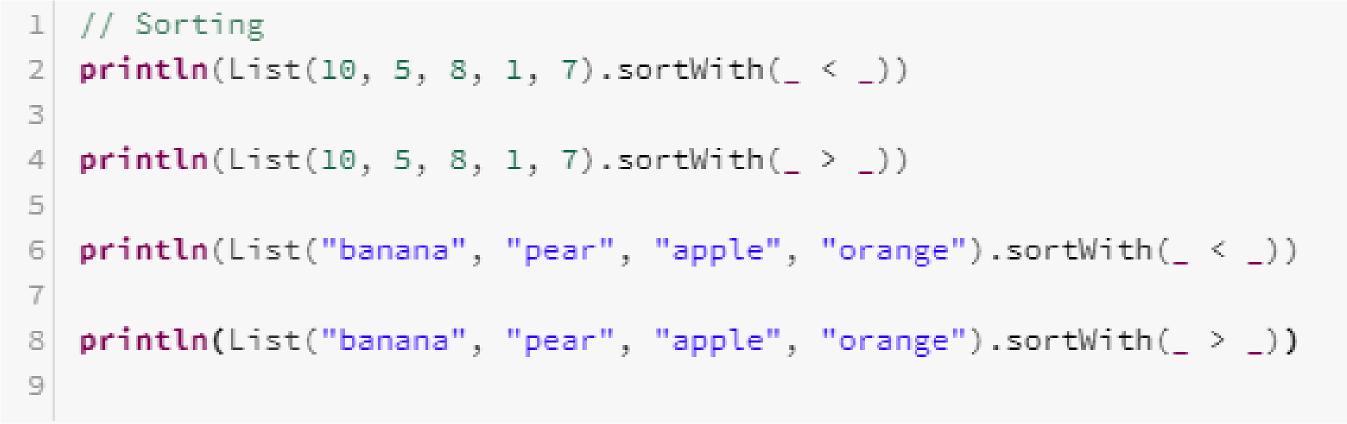


```scala
// Option Example
val employees = Map("Manager" -> "Bob Smith", "Secretary" -> "Sue Brown")
//printf("Manager with proper key : %s\n", employees("Manager"))
//printf("Manager without proper key : %s\n", employees("Man"))

// Without calling Option
printf("Manager with improper key without Option : %s\n", employees.get("Man"))
printf("Manager without Option : %s\n", employees.get("Manager"))

def show(x: Option[String]) = x match {
      case Some(s) => s
      case None => "Doesnt exist"
   }
printf("Manager : %s\n", show(employees.get("Manager")))
printf("Manager with improper key : %s\n", show(employees.get("Man")))

```

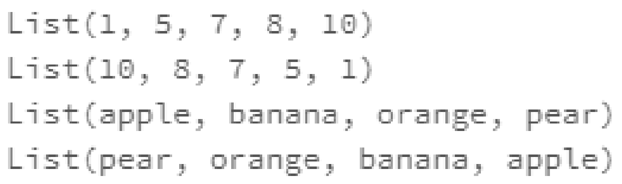
```
Manager with improper key without Option : None
Manager without Option : Some(Bob Smith)
Manager : Bob Smith
Manager with improper key : Doesnt exist
```

Sorting

- The sortWith method lets you provide your own sorting function.

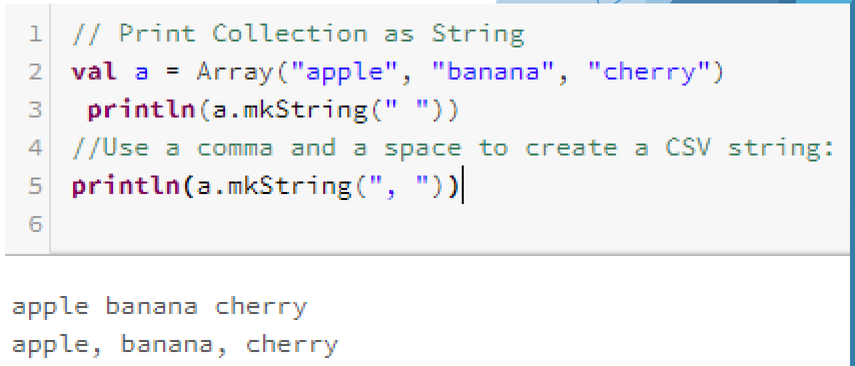


```
// Sorting
println(List(10, 5, 8, 1, 7).sortWith(_ < _))

println(List(10, 5, 8, 1, 7).sortWith(_ > _))

println(List("banana", "pear", "apple", "orange").sortWith(_ < _))

println(List("banana", "pear", "apple", "orange").sortWith(_ > _))

```



```
List(1, 5, 7, 8, 10)
List(10, 8, 7, 5, 1)
List(apple, banana, orange, pear)
List(pear, orange, banana, apple)
```

- Use the mkString method to print a collection as a String.



```
// Print Collection as String
val a = Array("apple", "banana", "cherry")
 println(a.mkString(" "))
//Use a comma and a space to create a CSV string:
println(a.mkString(", "))

```

```
apple banana cherry
apple, banana, cherry
```

Map & FlatMap - Methods

- The map method is most commonly used with collections.
- We typically use it to iterate over a list, performing an operation on each element and adding the result to a new list.

- scala> **valfruits = List("apple", "banana", "orange")**

fruits: List[String] = List(apple, banana, orange)

- scala> **fruits.map(_.toUpperCase)**

res0: List[String] = List(APPLE, BANANA, ORANGE)

- The flatMap method acts as a shorthand to map a collection and then immediately

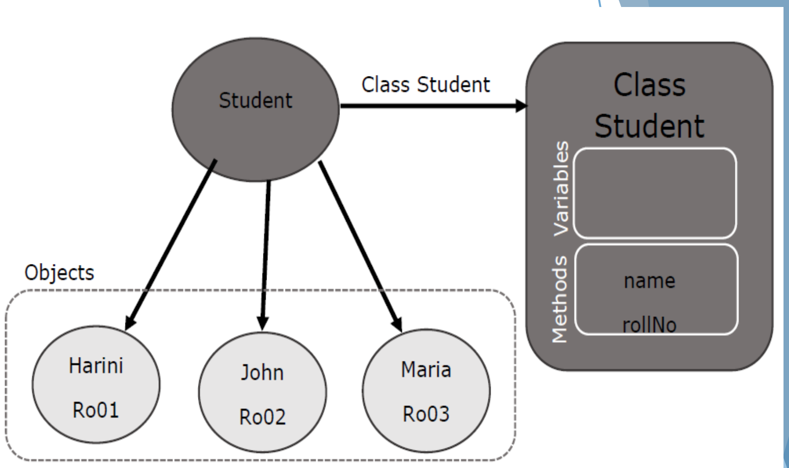
flatten it.

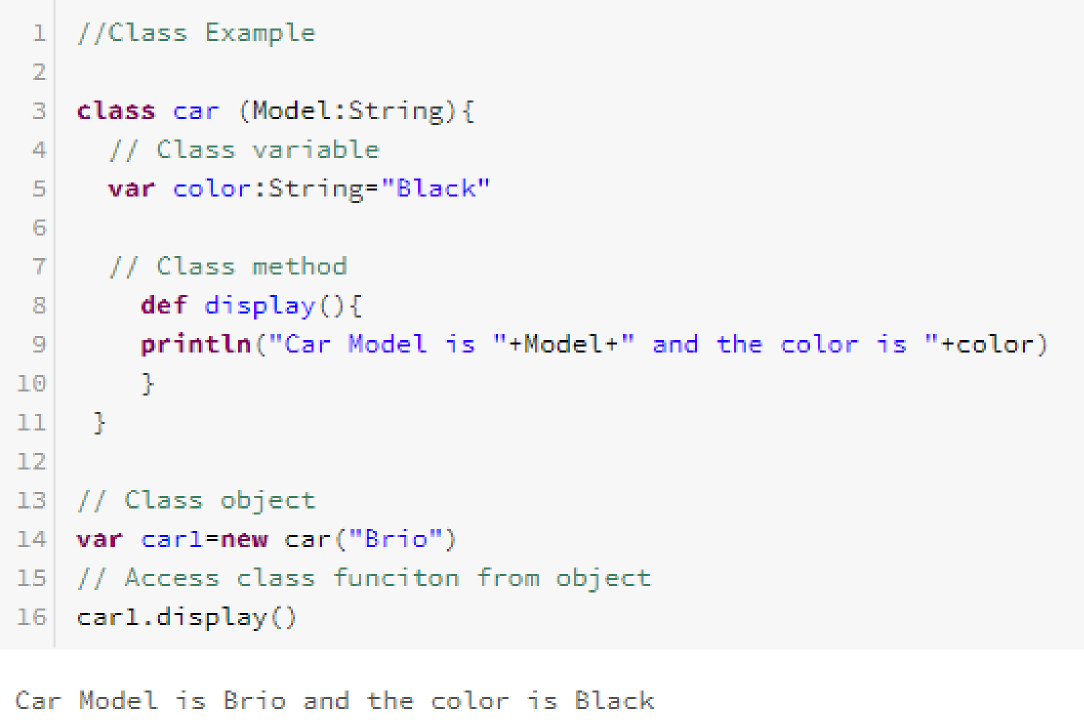
- This particular combination of methods is quite powerful.
- scala> **fruits.flatMap(_.toUpperCase)**

res1: Seq[Char] = List(A, P, P, L, E, B, A, N, A, N, A, O, R, A, N, G, E)

Classes and Objects

- A class is a blueprint for objects.
- Once you define a class, you can create objects from the class blueprint with the keyword new.
- Inside a class definition, you place fields and methods, which are collectively called members.
- Fields, which you define with either val or var, are variables
- Methods, which you define with def, contain executable code.
- protected variable means the field can only be accessed directly by methods defined in the class or by subclasses
- private fields can't be accessed by subclasses
- public fields can be accessed directly by anything



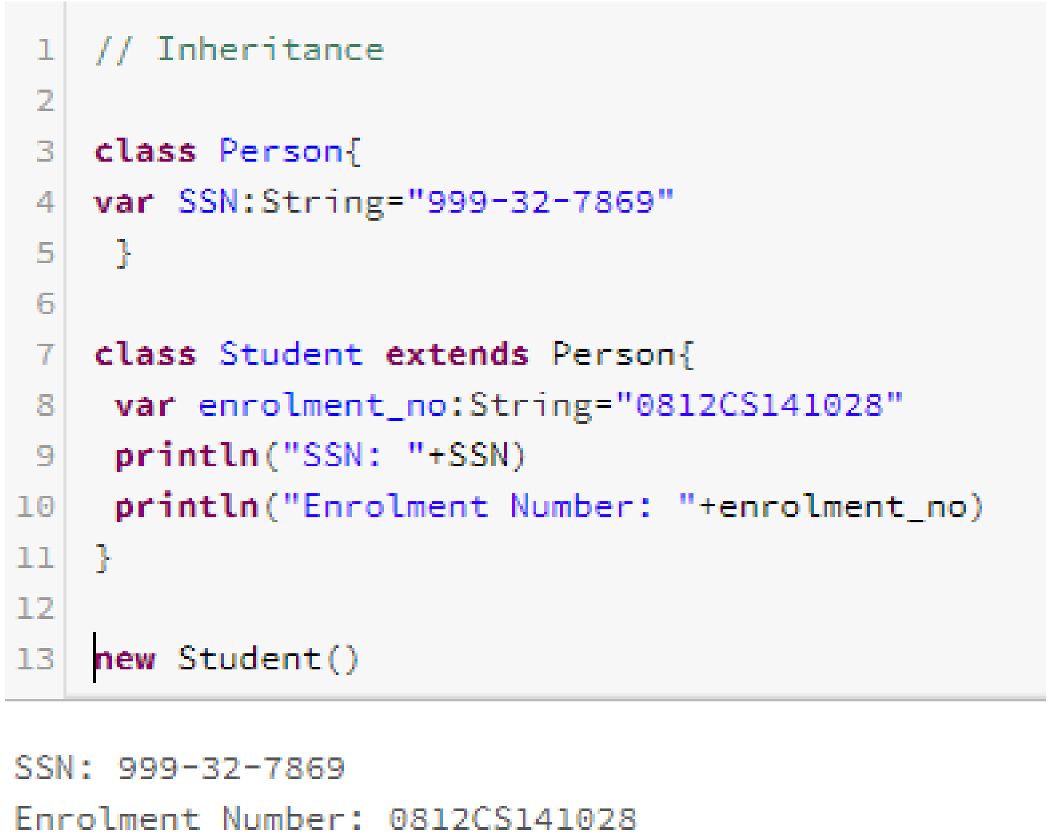


```
//Class Example

class car (Model:String){
  // Class variable
  var color:String="Black"

  // Class method
    def display(){
    println("Car Model is "+Model+" and the color is "+color)
    }
 }

// Class object
var car1=new car("Brio")
// Access class funciton from object
car1.display()
```

```
Car Model is Brio and the color is Black
```

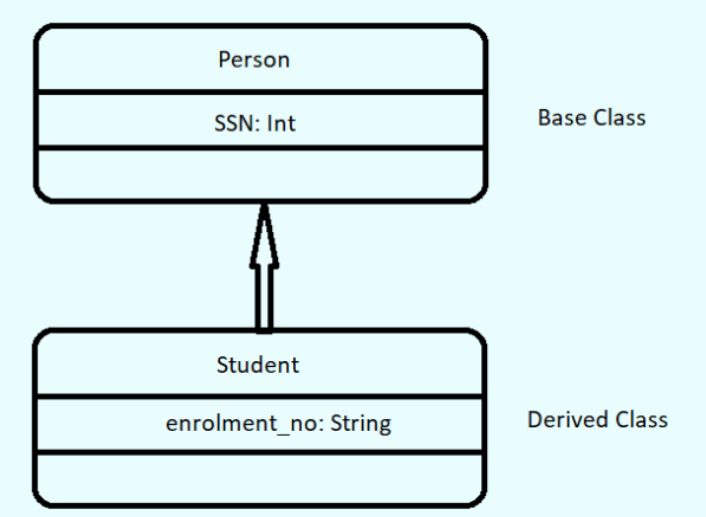
Inheritance

- When a class inherits from another, it means it extends another. We use the ‘extends’ keyword for this.
- This lets a class inherit members from the one it extends and lets us reuse code.
- A class that inherits from another gains all its fields and method



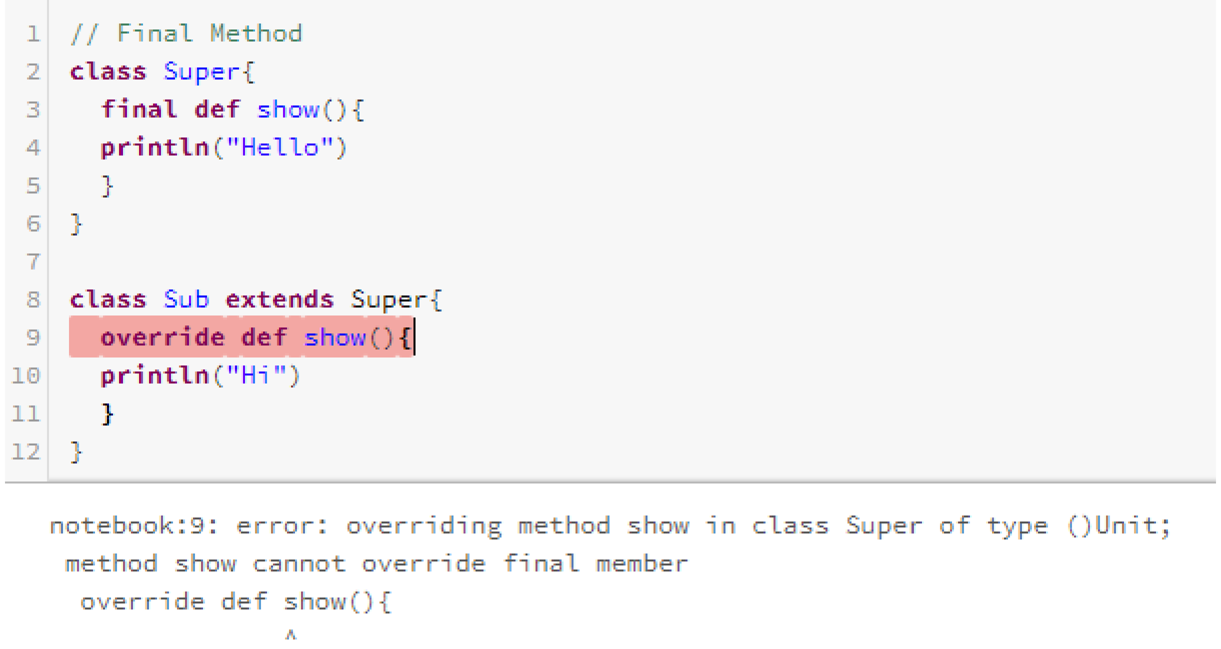
```
// Inheritance

class Person{
var SSN:String="999-32-7869"
 }

class Student extends Person{
 var enrolment_no:String="0812CS141028"
 println("SSN: "+SSN)
 println("Enrolment Number: "+enrolment_no)
}

new Student()
```

```
SSN: 999-32-7869
Enrolment Number: 0812CS141028
```



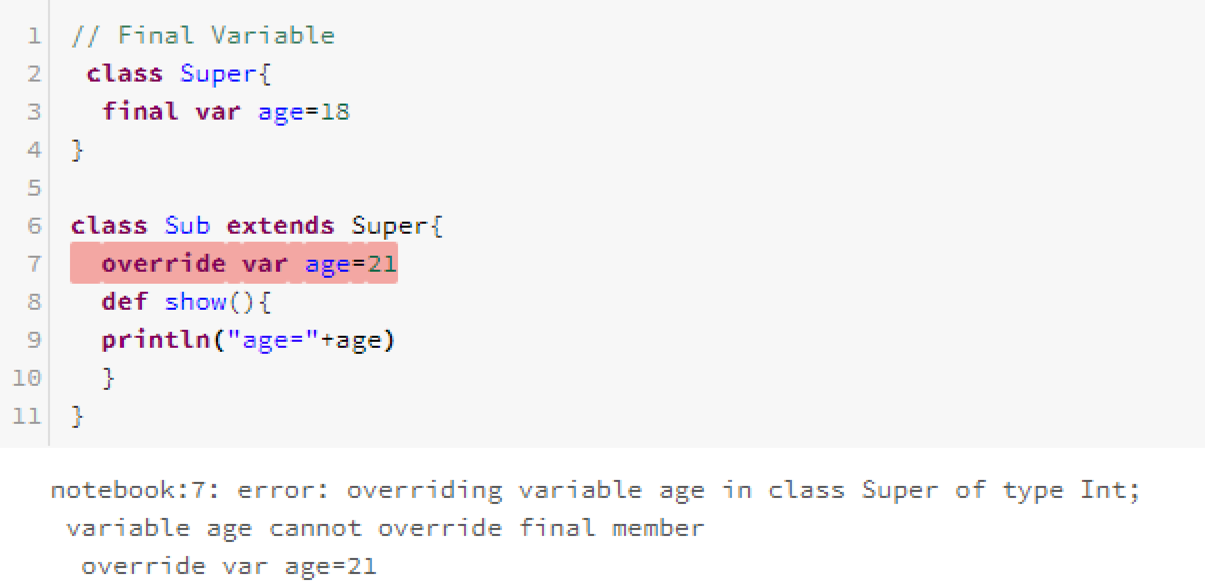
Final

- ‘final’ prevents a class from deriving members from its superclass.
- When we don’t want a class to be able to inherit a member from its superclass, we declare that member final
- This member may be a variable, a method, or even a class.



```
// Final Method
class Super{
  final def show(){
  println("Hello")
  }
}

class Sub extends Super{
  override def show(){
  println("Hi")
  }
}
```

```
notebook:9: error: overriding method show in class Super of type ()Unit;
 method show cannot override final member
  override def show(){
               ^
```

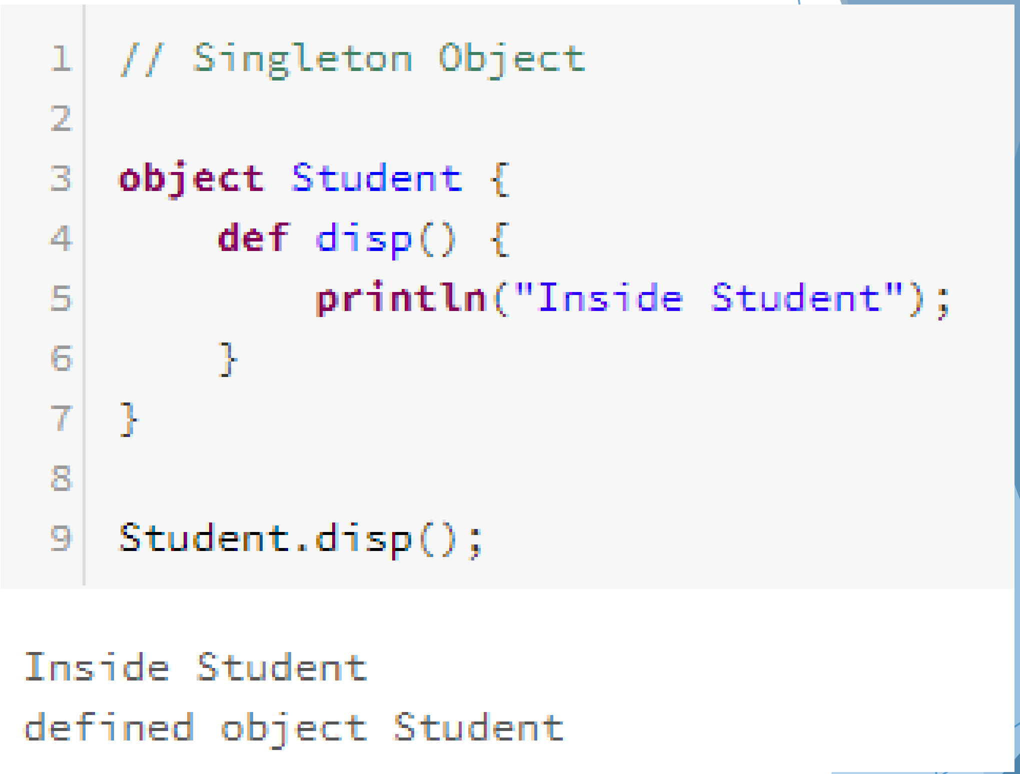


```
// Final Variable
 class Super{
  final var age=18
}

class Sub extends Super{
  override var age=21
  def show(){
  println("age="+age)
  }
}
```

```
notebook:7: error: overriding variable age in class Super of type Int;
 variable age cannot override final member
  override var age=21
```

Singleton Objects

- A singleton is a class that can have only one instance, i.e., Object
- You create singleton using the keyword object instead of class keyword
- No object is required to call methods declared inside singleton object
- A singleton object can extend classes and traits

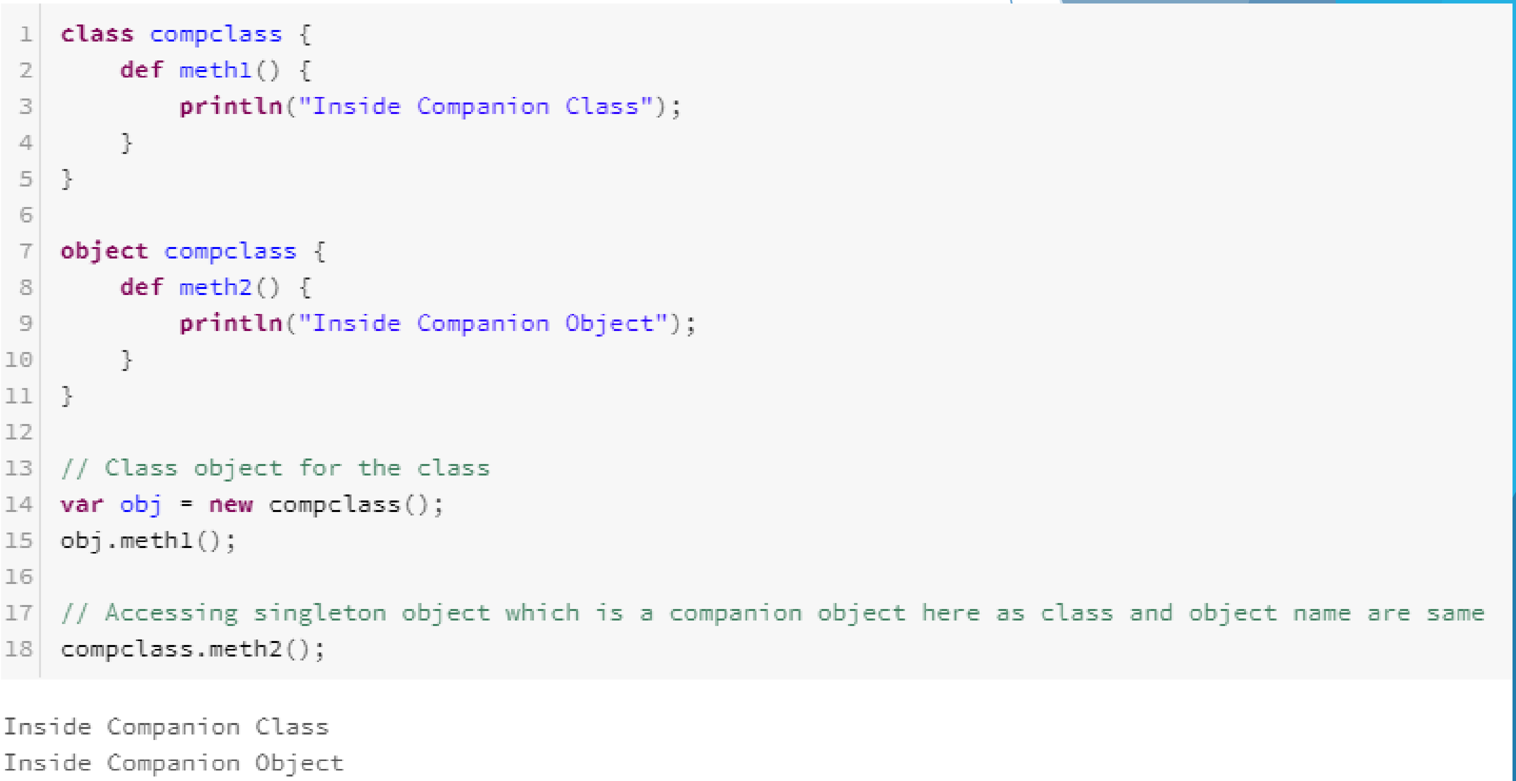


```
// Singleton Object

object Student {
    def disp() {
        println("Inside Student");
    }
}

Student.disp();
```

```
Inside Student
defined object Student
```

Companion Objects

- A Scala companion object is an object with the same name as a class
- We can call it the object's companion class.
- Either member of the pair can access its companion's private members.

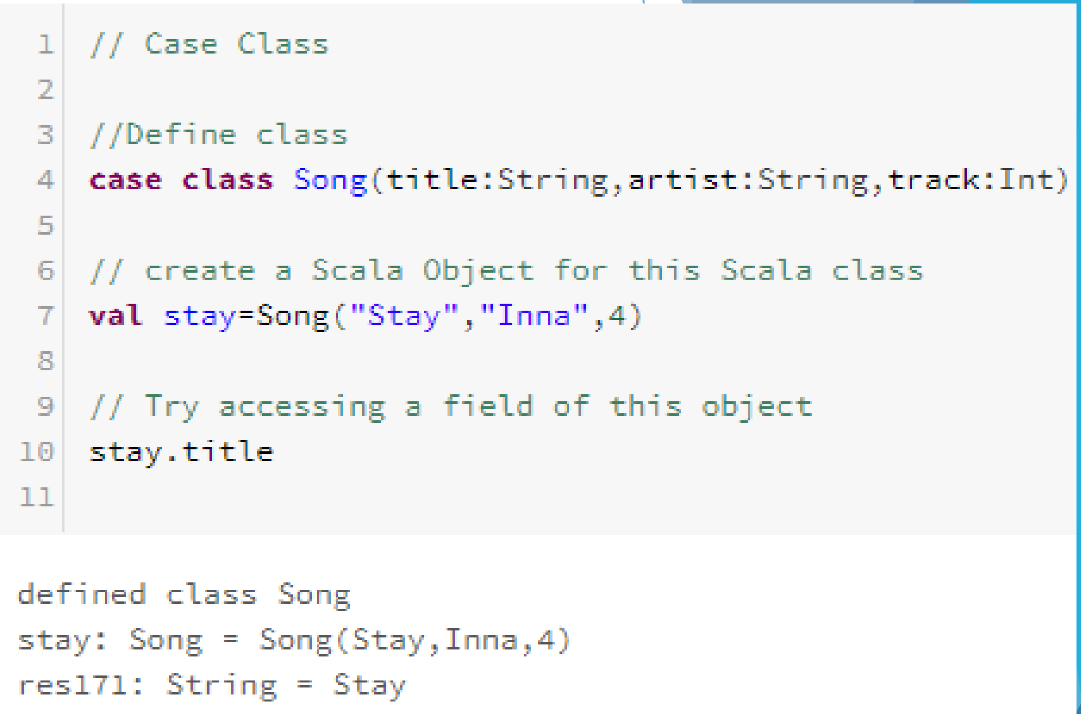


```
class compclass {
    def meth1() {
        println("Inside Companion Class");
    }
}

object compclass {
    def meth2() {
        println("Inside Companion Object");
    }
}

// Class object for the class
var obj = new compclass();
obj.meth1();

// Accessing singleton object which is a companion object here as class and object name are same
compclass.meth2();
```

```
Inside Companion Class
Inside Companion Object
```

Case Class

- A Scala Case Class is like a regular class, except it is good for modeling immutable data
- It give schema definition
- A scala case class also has all vals, which means they are immutable
- To create a Scala Object of a case class, we don’t use the keyword ‘new’

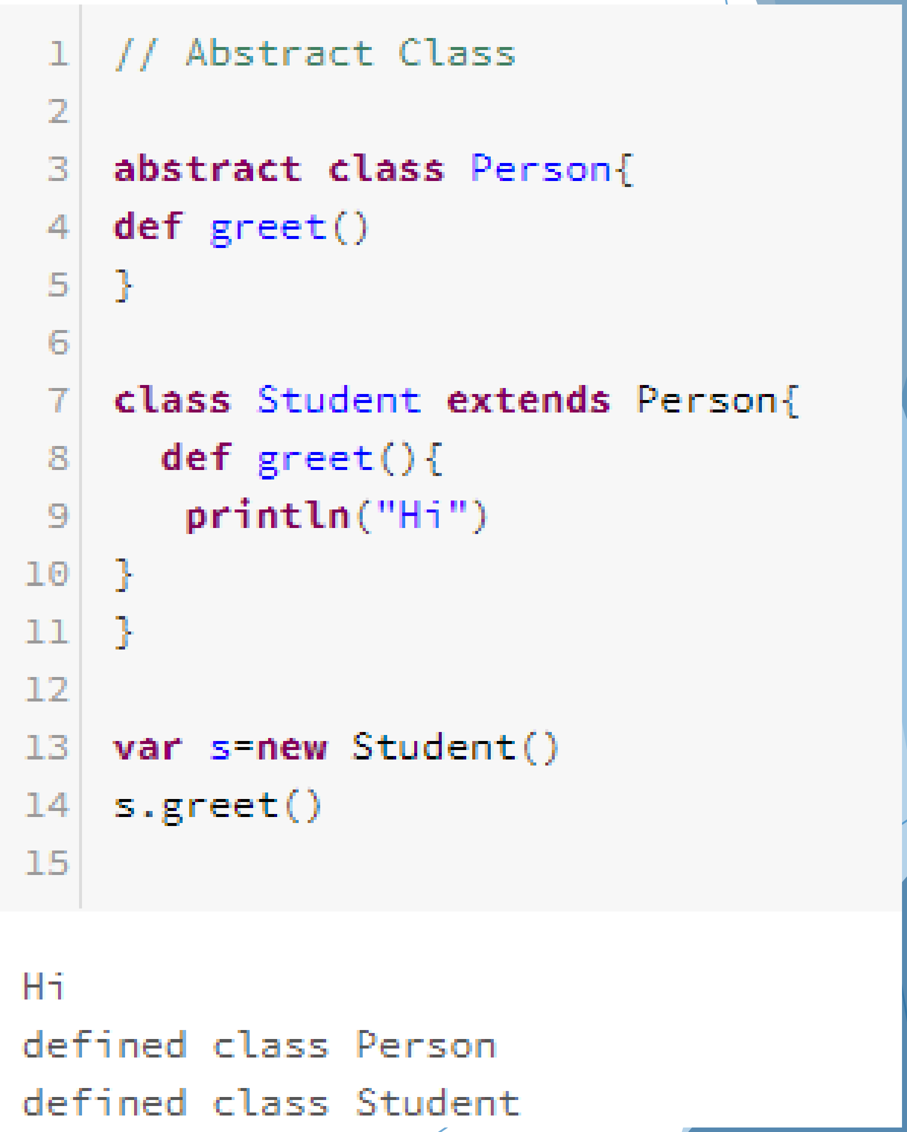


```
// Case Class

//Define class
case class Song(title:String,artist:String,track:Int)

// create a Scala Object for this Scala class
val stay=Song("Stay","Inna",4)

// Try accessing a field of this object
stay.title

```

```
defined class Song
stay: Song = Song(Stay,Inna,4)
res171: String = Stay
```

Abstract Class

- Abstraction is the process to hide the internal details and showing only the functionality
- In Scala, an abstract class is constructed using the abstract keyword
- The method which does not contain body is known as an abstract method.
- Abstract class cannot be instantiated

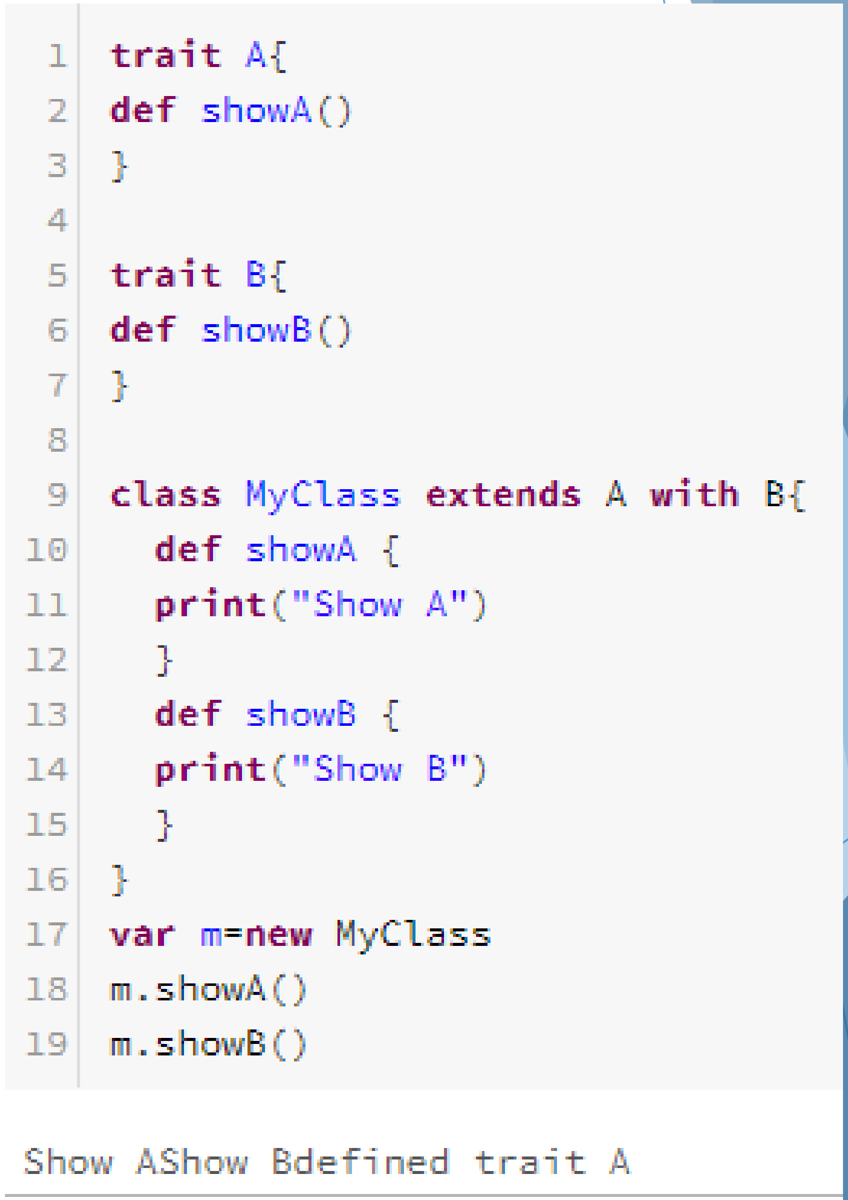


```
// Abstract Class

abstract class Person{
def greet()
}

class Student extends Person{
  def greet(){
   println("Hi")
}
}

var s=new Student()
s.greet()

```

```
Hi
defined class Person
defined class Student
```

Traits

- Traits are like interfaces in Java
- Traits are created using trait keywords
- Trait is a collection of abstract and non-abstract methods
- If a class implements multiple traits, it will extend the first trait (or a class, or abstract class), and then use with for other traits



```
trait A{
def showA()
}

trait B{
def showB()
}

class MyClass extends A with B{
  def showA {
  print("Show A")
  }
  def showB {
  print("Show B")
  }
}
var m=new MyClass
m.showA()
m.showB()
```

```
Show AShow Bdefined trait A
```

Access Modifiers

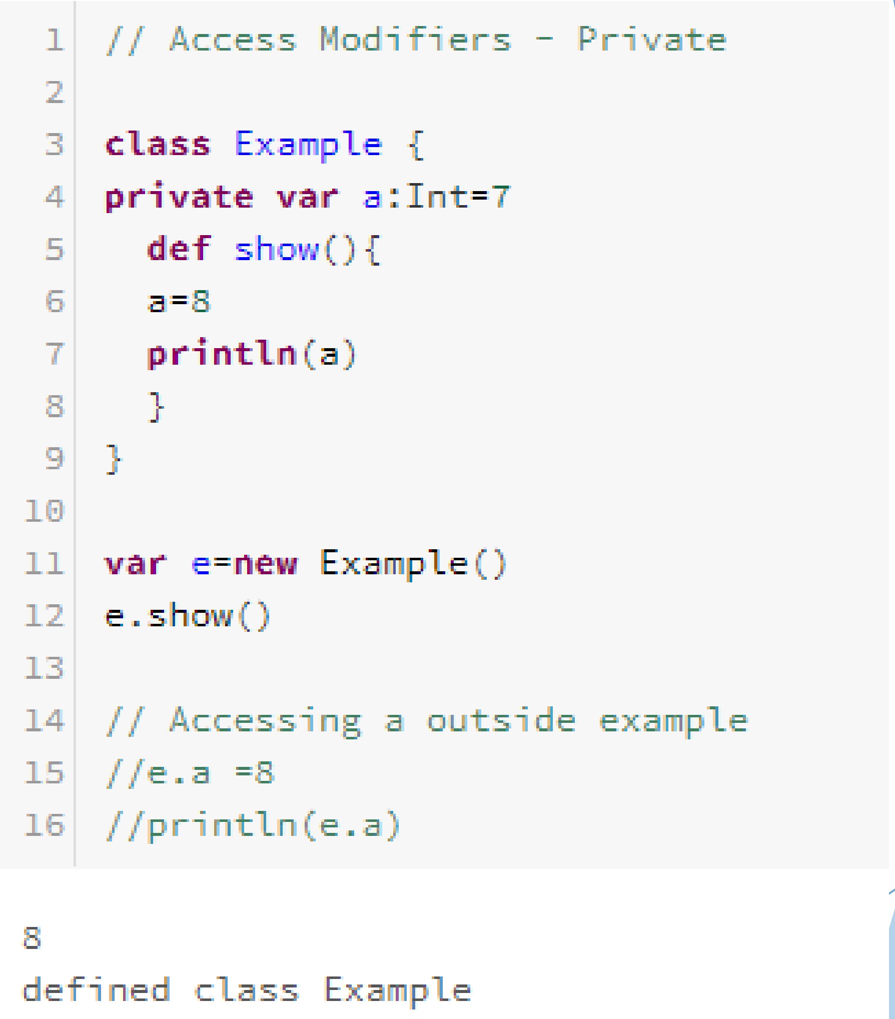
- Access Modifiers in scala are used to define the access field of members of packages, classes or objects in scala
- These modifiers will restrict accesses to the members to specific regions of code.

There are Three types of access modifiers available in Scala:

- Private
- Protected
- Public

Access Modifiers - Private Members

- When we declare a member as private, we can only use it inside its defining class or through one of its objects.
- To declare a member privately, we use the modifier “private”
- In this example, we declare the variable “a” to be private. This means that only the class Example can access it.

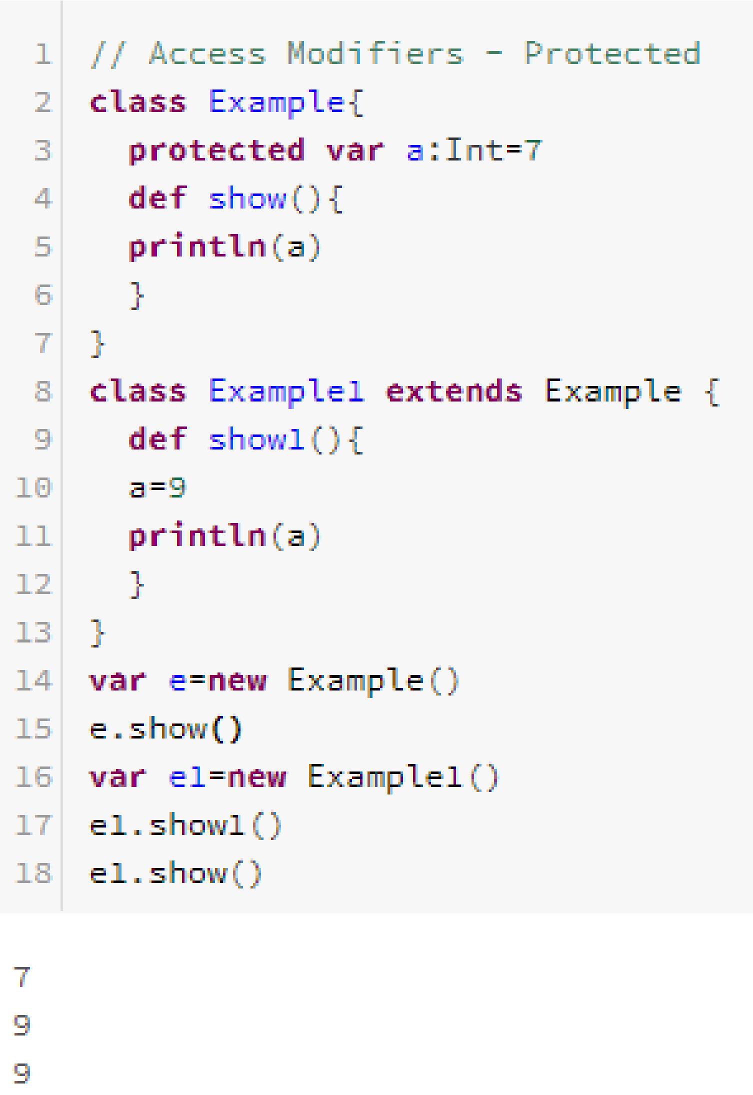


```
// Access Modifiers - Private

class Example {
private var a:Int=7
  def show(){
  a=8
  println(a)
  }
}

var e=new Example()
e.show()

// Accessing a outside example
//e.a =8
//println(e.a)
```

```
8
defined class Example
```

Access Modifiers – Protected Members

- We can only access protected members from within a class, from within its immediate subclasses, and from within companion objects.
- We use the modifier ‘protected’
- In the ex code, class Example1 inherits from Example, it can access the variable ‘a’, and also modify it

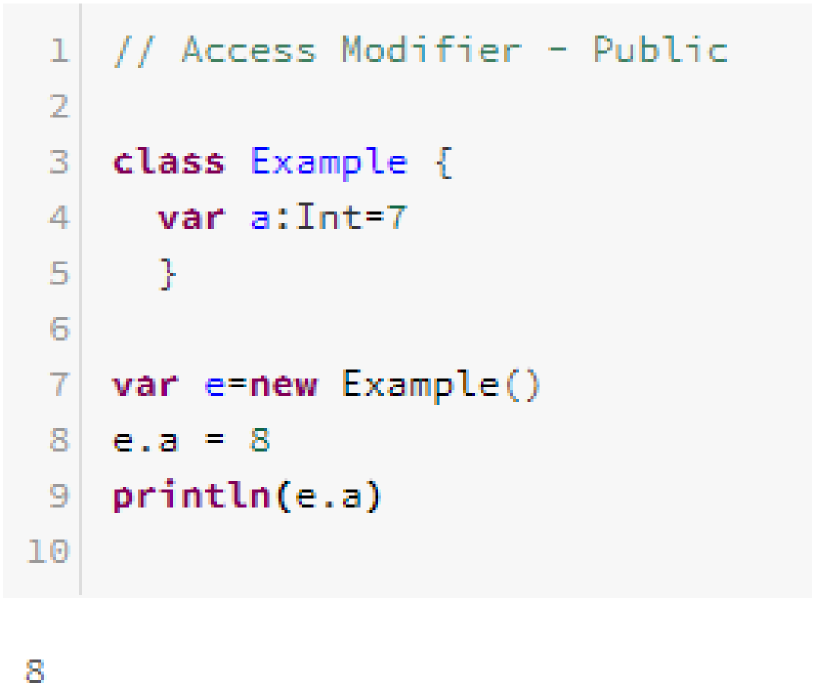


```
// Access Modifiers - Protected
class Example{
  protected var a:Int=7
  def show(){
  println(a)
   }
}
class Example1 extends Example {
  def show1(){
  a=9
  println(a)
   }
}
var e=new Example()
e.show()
var e1=new Example1()
e1.show1()
e1.show()
```

```
7
9
9
```

Access Modifiers – Public Members

- All members are default by public.
- If we do not accompany them with the modifiers ‘private’ or ‘protected’, they’re public.
- We can access these anywhere



```
// Access Modifier - Public

class Example {
  var a:Int=7
  }

var e=new Example()
e.a = 8
println(e.a)

```

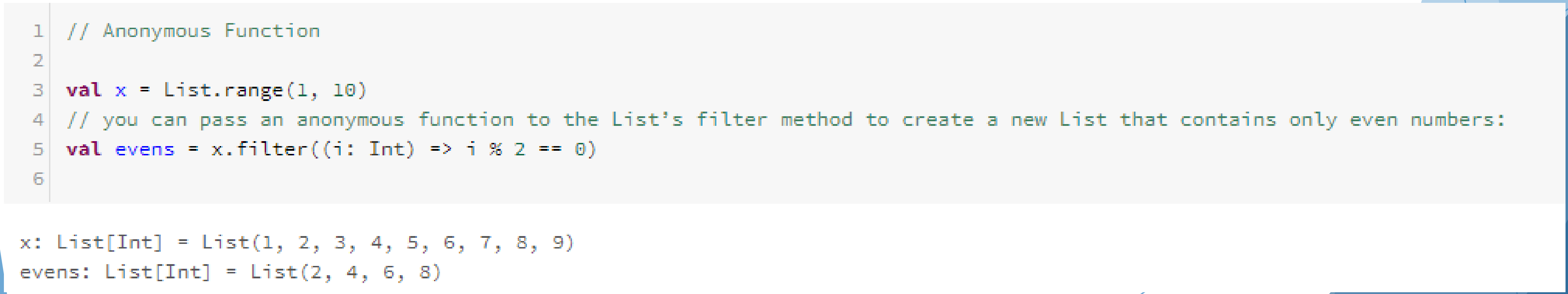
```
8
```

Anonymous Functions

- An anonymous function is also known as a function literal
- A function which does not contain a name is known as an anonymous function
- It is useful when we want to create an inline function.

Ex: (z:Int, y:Int)=> z*y

- In the above first syntax, => is known as a transformer.
- The transformer is used to transform the parameter-list of the left-hand side of the symbol into a new result using the expression present on the right-hand side.

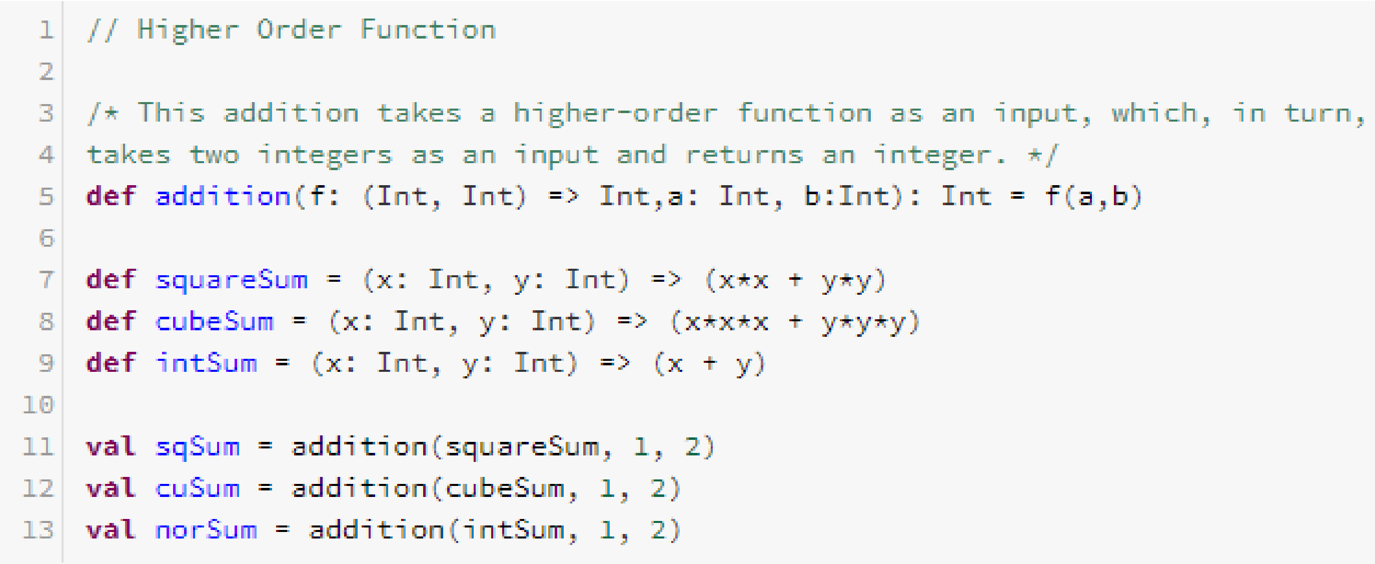


```scala
// Anonymous Function

val x = List.range(1, 10)
// you can pass an anonymous function to the List's filter method to create a new List that contains only even numbers:
val evens = x.filter((i: Int) => i % 2 == 0)

```

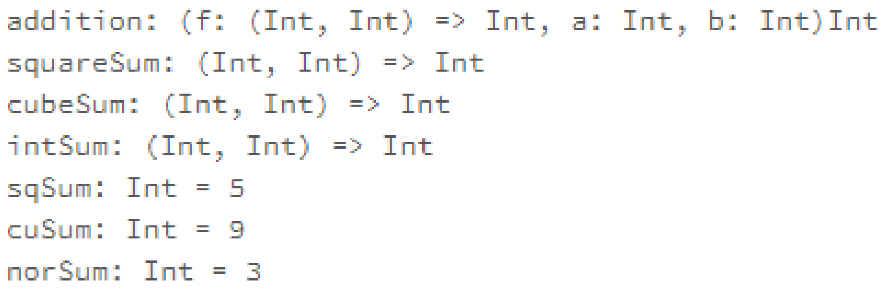
```
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
evens: List[Int] = List(2, 4, 6, 8)
```

Higher Order Functions

- Higher order function is a function that either takes a function as argument or returns a function.
- A function which works with function is called higher order function



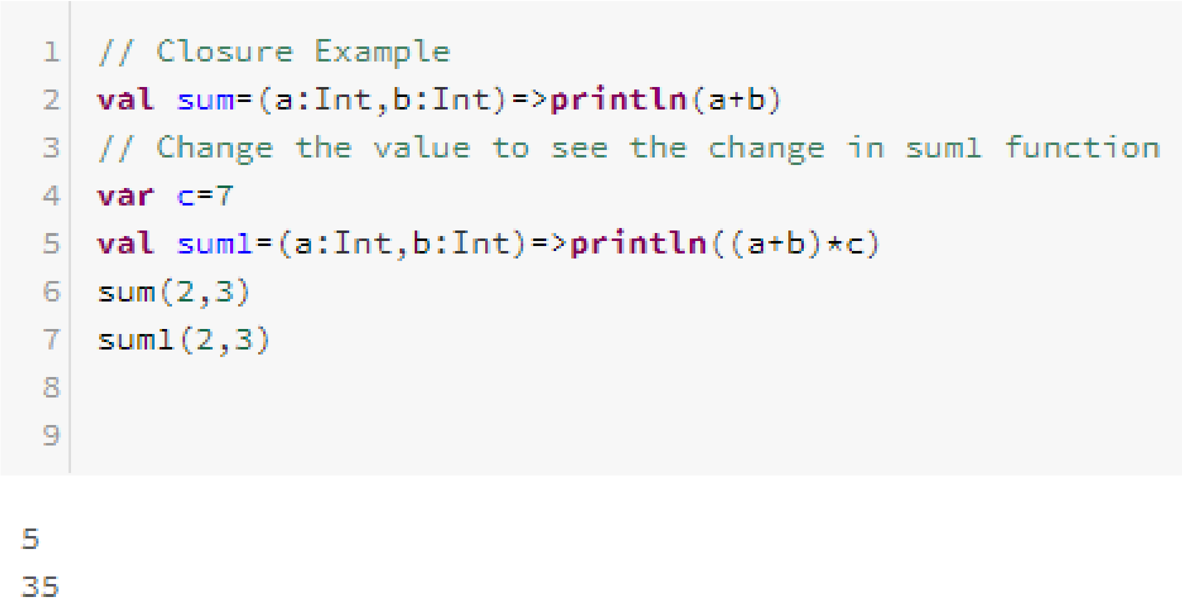
```
// Higher Order Function

/* This addition takes a higher-order function as an input, which, in turn,
takes two integers as an input and returns an integer. */
def addition(f: (Int, Int) => Int,a: Int, b:Int): Int = f(a,b)

def squareSum = (x: Int, y: Int) => (x*x + y*y)
def cubeSum = (x: Int, y: Int) => (x*x*x + y*y*y)
def intSum = (x: Int, y: Int) => (x + y)

val sqSum = addition(squareSum, 1, 2)
val cuSum = addition(cubeSum, 1, 2)
val norSum = addition(intSum, 1, 2)
```



```
addition: (f: (Int, Int) => Int, a: Int, b: Int)Int
squareSum: (Int, Int) => Int
cubeSum: (Int, Int) => Int
intSum: (Int, Int) => Int
sqSum: Int = 5
cuSum: Int = 9
norSum: Int = 3
```

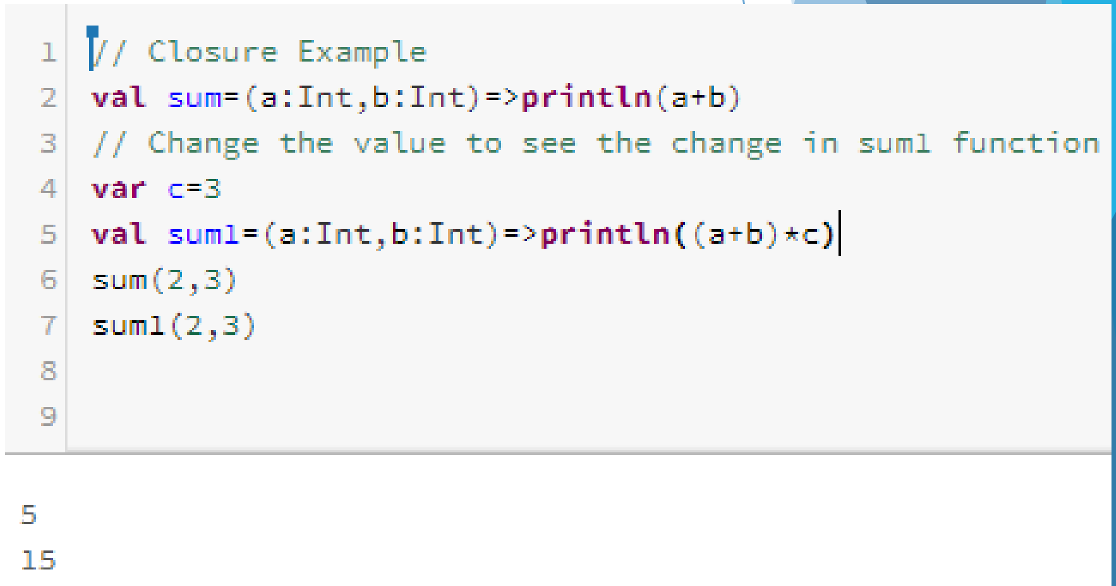
Closure

- A function whose return value depends on variable(s) declared outside it, is a closure.
- A free variable is any kind of variable which is not defined within the function and not passed as the parameter of the function
- If the value of the free variable changes, the value of the closure function changes
- Closure function takes the most recent state of the free variable and changes the value of the closure function accordingly.



```
// Closure Example
val sum=(a:Int,b:Int)=>println(a+b)
// Change the value to see the change in sum1 function
var c=7
val sum1=(a:Int,b:Int)=>println((a+b)*c)
sum(2,3)
sum1(2,3)
```

```
5
35
```

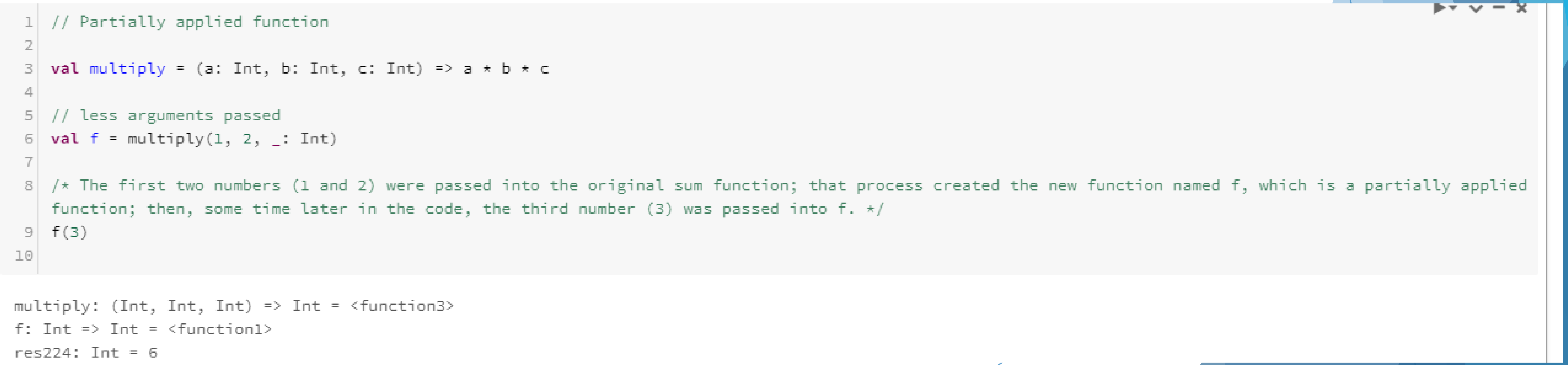


```
// Closure Example
val sum=(a:Int,b:Int)=>println(a+b)
// Change the value to see the change in sum1 function
var c=3
val sum1=(a:Int,b:Int)=>println((a+b)*c)
sum(2,3)
sum1(2,3)
```

```
5
15
```

Partially Applied Function

- A partially applied function is an expression in which you don't supply all of the arguments needed by the function.
- Instead, you supply some, or none, of the needed arguments.
- These arguments which are not passed to function we use hyphen(_) as placeholder.
- For the missing value it assigns a reference value to the variable
- When you apply the missed argument to this new function value, it will turn around and invoke the function, passing in all the required arguments.
- Pass to function less arguments than it has in its declaration. Scala returns a new function with rest of arguments that need to be passed.

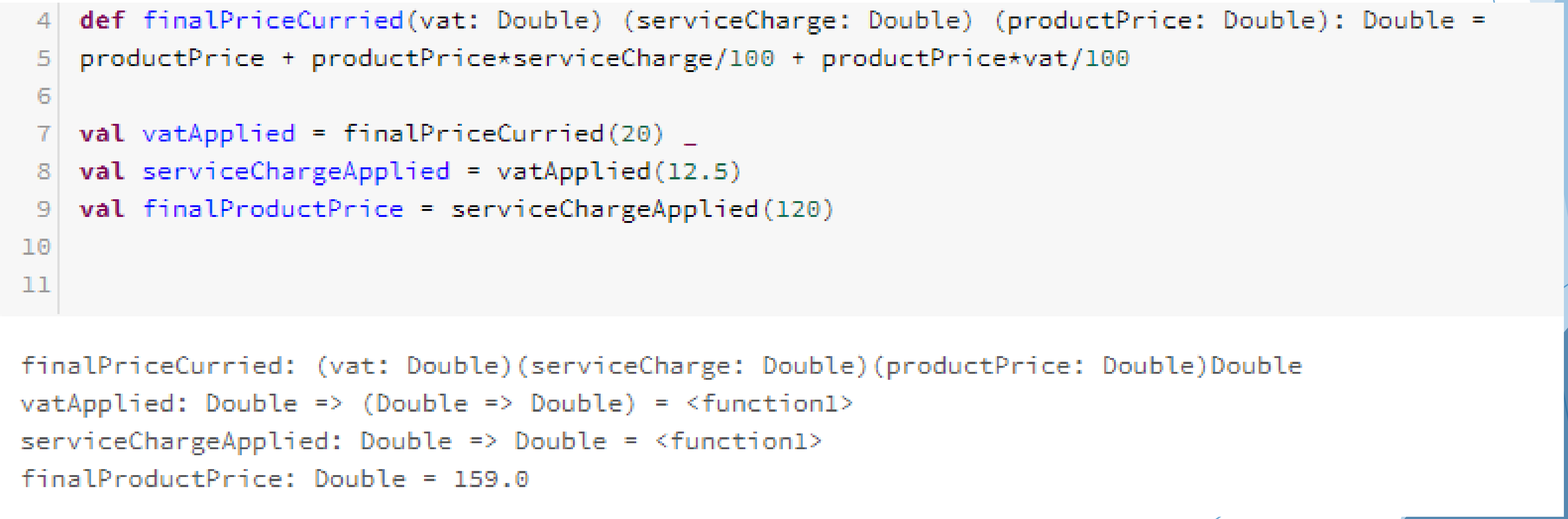


```scala
// Partially applied function

val multiply = (a: Int, b: Int, c: Int) => a * b * c

// less arguments passed
val f = multiply(1, 2, _: Int)

/* The first two numbers (1 and 2) were passed into the original sum function; that process created the new function named f, which is a partially applied function; then, some time later in the code, the third number (3) was passed into f. */
f(3)

```

```
multiply: (Int, Int, Int) => Int = <function3>
f: Int => Int = <function1>
res224: Int = 6
```

Currying Functions

- If a Scala function takes multiple parameters, we can transform it into a chain of functions where each takes a single parameter.
- Currying splits methods with multiple parameters into a chain of functions — each with one parameter.

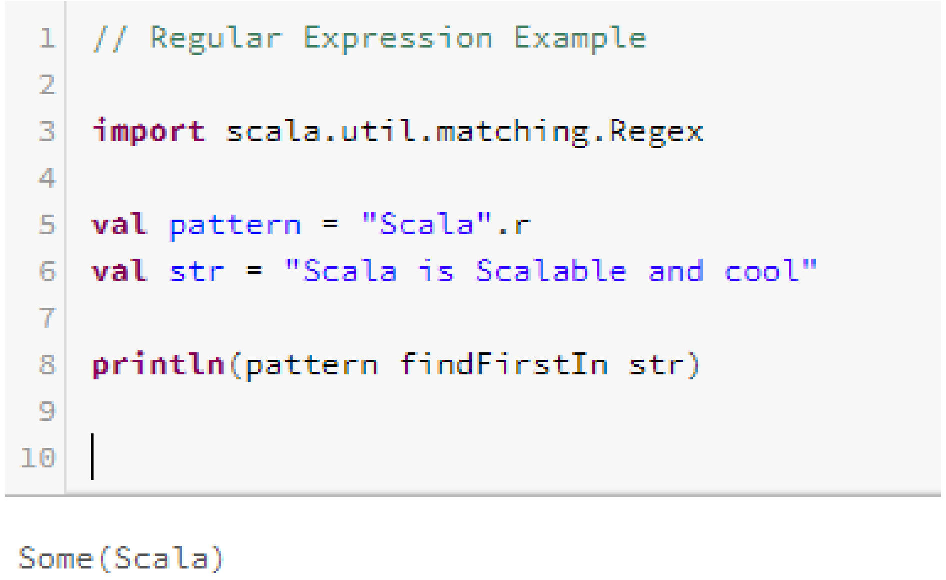


```scala
def finalPriceCurried(vat: Double) (serviceCharge: Double) (productPrice: Double): Double =
productPrice + productPrice*serviceCharge/100 + productPrice*vat/100

val vatApplied = finalPriceCurried(20) _
val serviceChargeApplied = vatApplied(12.5)
val finalProductPrice = serviceChargeApplied(120)


```

```
finalPriceCurried: (vat: Double)(serviceCharge: Double)(productPrice: Double)Double
vatApplied: Double => (Double => Double) = <function1>
serviceChargeApplied: Double => Double = <function1>
finalProductPrice: Double = 159.0
```

Regular Expressions

- Scala supports regular expressions through Regex class available in the scala.util.matching package.
- We create a String and call the r( ) method on it.

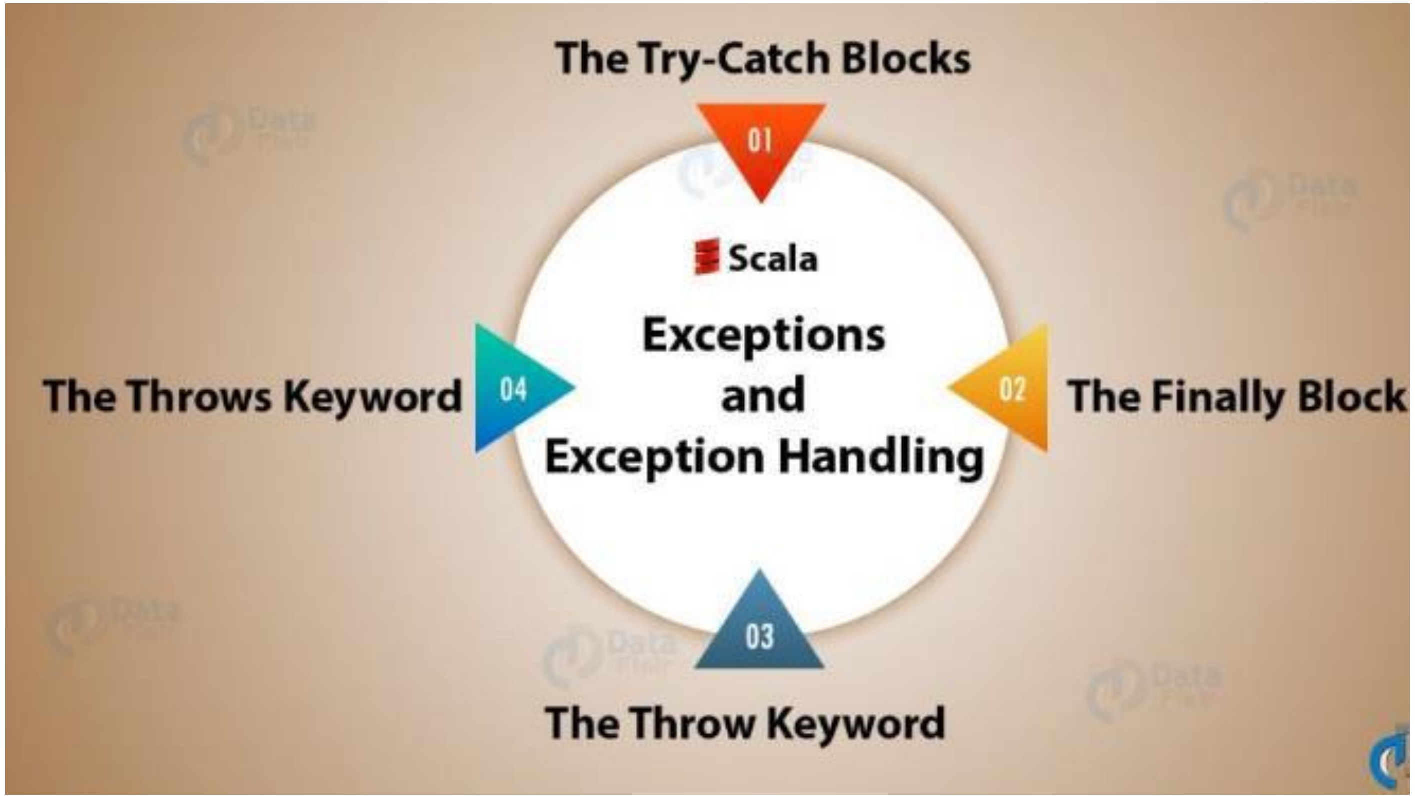


```
// Regular Expression Example

import scala.util.matching.Regex

val pattern = "Scala".r
val str = "Scala is Scalable and cool"

println(pattern findFirstIn str)

```

```
Some(Scala)
```

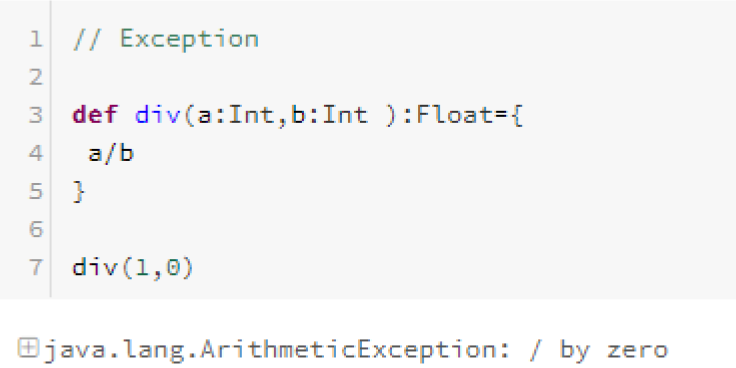
Exception Handling

- There may be situations your code may misfunction when you run it.
- These abnormal conditions may cause your program to terminate abruptly.
- Such runtime errors are called exceptions.



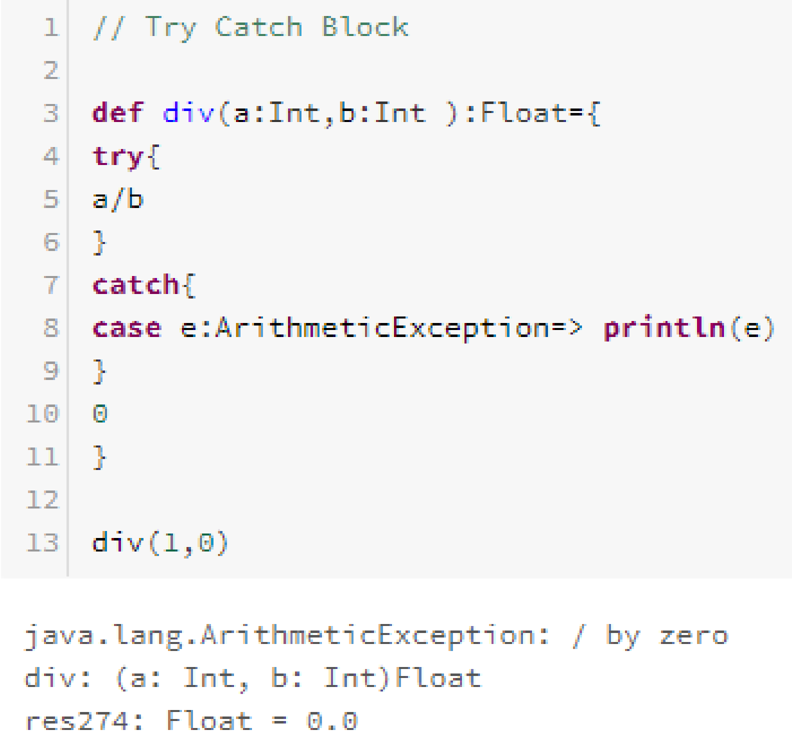
Try Catch Blocks:

- When we suspect that a line or a block of code may raise an exception in Scala, we put it inside a try block.
- What follows is a catch block.
- We can make use of any number of Try Catch Blocks in a program.



```scala
// Exception

def div(a:Int,b:Int ):Float={
 a/b
}

div(1,0)
```

```
java.lang.ArithmeticException: / by zero
```

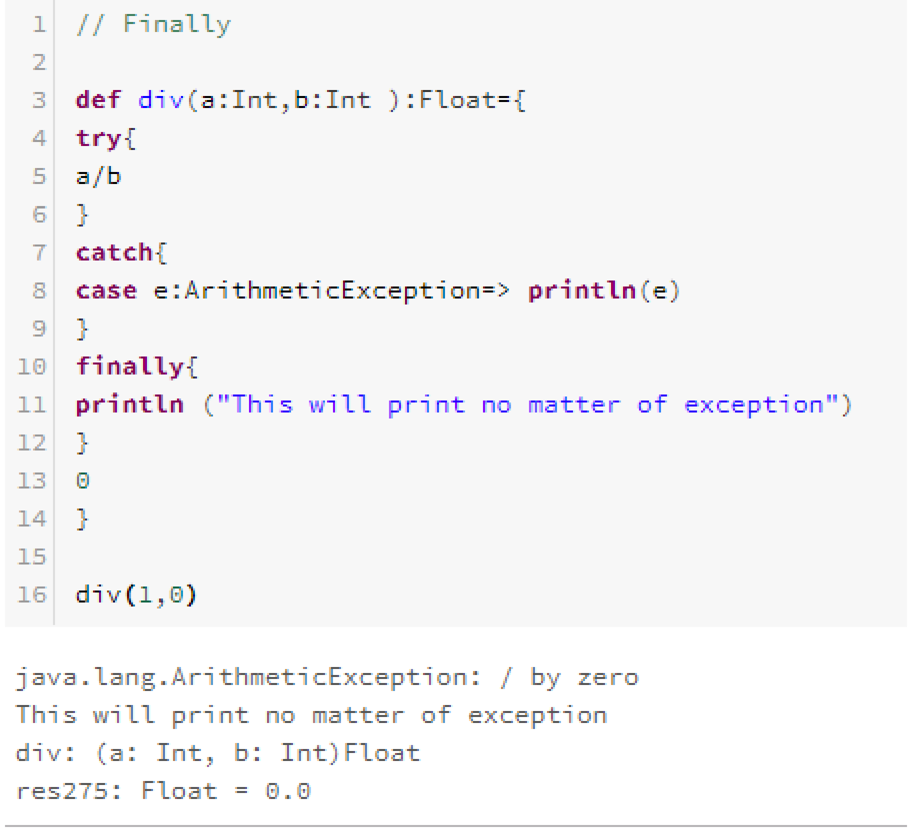


```scala
// Try Catch Block

def div(a:Int,b:Int ):Float={
try{
a/b
}
catch{
case e:ArithmeticException=> println(e)
}
0
}

div(1,0)
```

```
java.lang.ArithmeticException: / by zero
div: (a: Int, b: Int)Float
res274: Float = 0.0
```

Finally Block

- A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.
- Code to release all the resources. This could be a file, a network connection, or even a database connection.

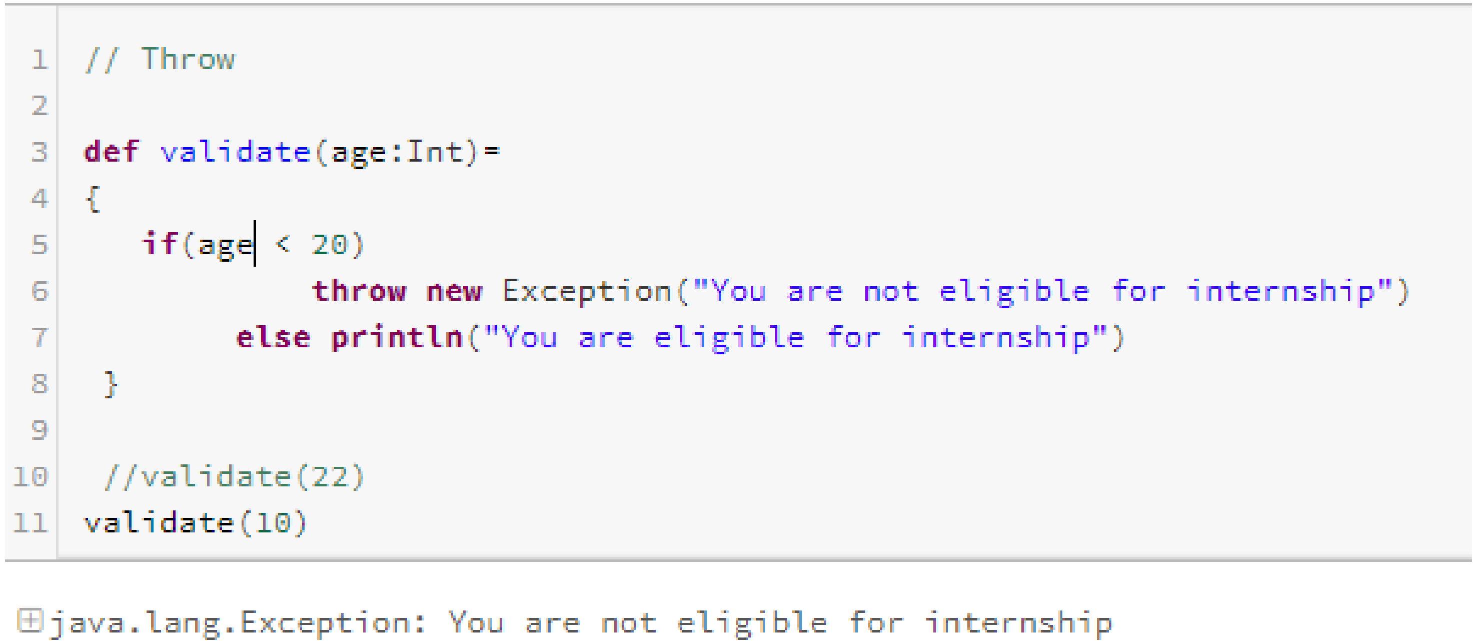


```
// Finally

def div(a:Int,b:Int ):Float={
try{
a/b
}
catch{
case e:ArithmeticException=> println(e)
}
finally{
println ("This will print no matter of exception")
}
0
}

div(1,0)
```

```
java.lang.ArithmeticException: / by zero
This will print no matter of exception
div: (a: Int, b: Int)Float
res275: Float = 0.0
```

Throw

- We can also explicitly throw a Scala exception in our code.
- We use the Throw Keyword for this.
- Let’s create a custom exception.



```
// Throw

def validate(age:Int)=
{
   if(age < 20)
            throw new Exception("You are not eligible for internship")
        else println("You are eligible for internship")
 }

 //validate(22)
validate(10)
```

```
⊞java.lang.Exception: You are not eligible for internship
```

Thank You

Keerthiga Barathan