Introduction To Scala Programming

Built-in Control Structures : Expressions

Qn. Suppose we have the questions.

1. Write a scala program that will run through a set of 10 marks and print out the marks and their total sum.
2. Write a scala program that will run through a set of 10 marks and print out the marks and their corresponding grade according to a given grading criteria.
3. Write a scala program that will scan through the current directory and print out the all the filesystem
4. Write a scala program that will scan through the current directory and print out the all the files on only if there are files

As you listen to the question above , we are being directed to what we already know : The use of **control structures.**

In Java, C and C++ we have encountered them: decision structures: **if, if/else, nested if/else, switch case** structures and Loops: **for loop, while loop, and do-while.**

Scala provide support for controls structures but has only **if , while,do - while , for , try , match , and function calls** control structures.

The reason Scala has so few is that it has provided a rich set of **function literals** . Instead of accumulating one higher-level control structure after another in the base syntax, Scala accumulates function literals in libraries.

NOTE

- One thing you will notice is that almost all of Scala's control structures **result in some value**. So all the above structures should end up resulting to a value.
- This makes it possible for programmers to use these result values to simplify their code, just as they use return values of functions.
- Without this design feature, the programmer must then create temporary variables just to hold results that are calculated inside a control structure.

- Removing these temporary variables makes the code a little simpler, and it also prevents many bugs.
- However the **while and do-while** constructs are called “**loops**,” not **expressions**, because they don’t result in an interesting value.

Imperative Style Vs Functional Style

Before we dive into the built in scala control structures let us briefly discuss the functional style of programming supported by scala

When we write an iteration code using the **while** ,the do-while and the **for** loops as in java , then we are programming in an **imperative** style.

In the imperative style, which is the style you normally use with languages like Java, C++, and C, we write one imperative command at a time, iterate with loops, and often **mutate** state shared between different functions.

Let us consider the very usual while loop case below-mentioned

Example 1: Writing a scala program that will scan through all the integer arguments input at the command line and print them out and their sums.

```
object LoopExample1 {
def main(args: Array[String]) ={
var i = 0
var sum = 0
while (i < args.length) {
 println(args(i))//prints out the ith command line argument
 sum =sum + Integer.parseInt(args(i)) // You understand this?
  i += 1  //increments i by one.
}
 println(sum)
   }//End of main
}//End of class
```

The sum variable in the above code is changed (**mutated**) in side the while loop.

With Scala you can program **imperatively** but it also enable you program in a **functional** style.

You are advised in this course that you become as comfortable with the functional style as you are with imperative style.

Functional programming is a style of programming that emphasizes writing applications using only **pure functions** and **immutable values**.

This means functional programming have strongly focuses on a desire to see their code as **math** :

to see the combination of their functions as a series of algebraic equations.

In that regard, you could say that functional programmers like to think of themselves as mathematicians.

That’s the driving desire that leads them to use *only* pure functions and immutable values, because that’s what you use in algebra.

Scala supports strongly **functional programming**

Pure Functions

One of Ups of Scala is that it enables you write a program in pure functions.

Alvin Alexander (In his book Functional Programming, Simplified) defines a *pure function* like as one with the following descriptions :

- The function’s output depends *only* on its input variables
- It doesn’t mutate any hidden state
- It doesn’t have any “back doors”: I.e It doesn’t read data from the outside world (including the console, web services, databases, files, etc.), or write data to the outside world

As a result of this definition, any time you call a pure function with the same input value(s), you’ll always get the same result.

Definition

> *A pure function is a function that depends only on its declared inputs and its internal algorithm to produce its output. It does not read any other values from “the outside world” — the world outside of the function’s scope — and it does not modify any values in the outside world.*

Examples of pure functions

Given that definition of pure functions, as you might imagine, methods like these in the *scala.math._* package are pure functions:

- **`abs`**
- **`ceil`**
- **`max`**
- **`min`**

These Scala `String` methods are also pure functions:

- **`isEmpty`**
- **`length`**
- **`substring`**

Many methods on the Scala collections classes also work as pure functions, including `drop`, `filter`, and `map`.

However real-world applications consist of a combination of **pure** and **impure** functions.

An application isn't very useful if it can't read or write to the outside world, so people make this recommendation:

> *Write the core of your application using pure functions, and then write an impure "wrapper" around that core to interact with the outside world.*

Consider the case of putting a layer of impure icing on top of a pure cake.

There are ways to make impure interactions with the outside world feel a little more pure.

Examples of impure functions

Conversely, the following functions are *impure* because they violate the definition.

The **`foreach`** method on collections classes is impure because it's only used for its **side effects,** such as printing to STDOUT.

> A great hint that `foreach` is impure is that it's method signature declares that it returns the type **`Unit`**. Because it returns nothing, logically the only reason you ever call it is to achieve some side effect.
>
> Similarly, *any* method that returns **`Unit`** is going to be an impure function.

Date and **time** related methods like `getDayOfWeek`, `getHour`, and `getMinute` are all impure because their output depends on something other than their input parameters. Their results rely on some form of hidden **I/O**, ***hidden inputs***.

In general, impure functions do one or more of these things:

- Read hidden inputs, i.e., they access variables and data not explicitly passed into the function as input parameters
- Write hidden outputs
- Mutate the parameters they are given
- Perform some sort of I/O with the outside world

Writing pure functions

Writing pure functions in Scala is one of the simpler parts about functional programming: You just write pure functions using Scala's method syntax. Here's a pure function that doubles the input value it's given:

```
def double(i: Int): Int = i * 2
```

Here's a pure function that calculates the sum of a list of integers (`List[Int]`):

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case head :: tail => head + sum(tail)
}
```

Even though we haven't covered recursion, if you can understand that code, you'll see that it meets my definition of a pure function.

In Built Scala Functions

Scala has a rich set of math functions in package scala.math

The package object `scala.math` contains methods for performing basic numeric operations such as elementary exponential, logarithmic, root and trigonometric functions.

All methods forward to java.lang.Math unless otherwise noted.

Examples include:

Minimum and Maximum

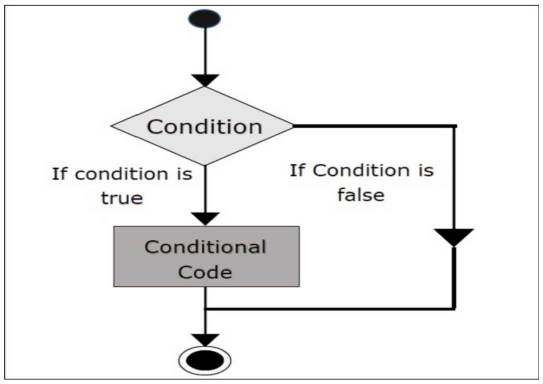
Find the min or max of two numbers. Note: scala.collection.IterableOnceOps has min and max methods which determine the min or max of a collection.

def max(x: Int, y: Int): Int
def max(x: Long, y: Long): Long
def max(x: Float, y: Float): Float
def max(x: Double, y: Double): Double
def min(x: Int, y: Int): Int
def min(x: Long, y: Long): Long
def min(x: Float, y: Float): Float
def min(x: Double, y: Double): Double

For a complete reference to scala math functions visist- **https://dotty.epfl.ch/api/scala/math.html**

The If expressions

Scala's if works just like in many other languages. It tests a condition and then executes one of two code branches depending on whether the condition holds true.



Syntax

The syntax of an 'if' statement is as follows.

```
if(Boolean_expression) {
   // Statements will execute if expression is
true
}
```

If the `Boolean` expression evaluates to true then the block of code inside the 'if' expression will be executed. If not, the first set of code after the end of the 'if' expression (after the closing curly brace) will be executed.

Try the following example program to understand conditional expressions (if expression) in Scala Programming Language.

Example1

```
object IfExample1 {
   def main(args: Array[String]) {
      var x = 10;
      if( x < 20 ){
         println("This is if statement");
      }
   }
}
```

What should be the name of the file for the code above?……………………….....

If-else Statement

An 'if' statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax

The syntax of a if...else is −

```
if(Boolean_expression){
   //Executes when the Boolean expression is true
} else{
   //Executes when the Boolean expression is false
}
```

Example2

```
object IfExample2 {
   def main(args: Array[String]) {
      var x = 30;
      if( x < 20 ){
         println(“X is not less than 20”);
      } else {
         println(x);
      }
   }
}
```

If-else-if-else Statement

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 ending else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax

The following is the syntax of an ‘if...else if...else’ is as follows −

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
} else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
} else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
} else {
   //Executes when the none of the above condition is true.
}
```

IfExample2a: To illustrate the use of if..else if ..else in Imperative Style

```
object IfExample2 {
   def main(args: Array[String]) ={
var x =Math.ceil(Math.random()*100);
      if( x >= 70 ){
         println(x+": A");
      } else if( x >= 60 ){
         println(x+": B");
      } else if( x >= 50 ){
         println(x+": C");
      } else if( x >= 40 ){
         println(x+": D");
      } else{
         println(x+": F");
      }
   }//End of main
}//End of class
```

If Example2b: To illustrate the use of if..else if ..else in Functional Style

```
object IfExample3 {
   def main(args: Array[String]) ={
val x =Math.ceil(Math.random()*100);
      val grd=if( x >= 70 ) x+": A"
       else if( x >= 60 ) x+": B"
        else if( x >= 50 ) x+": C"
         else if( x >= 40 ) x+": D"
          else x+": F"
           println(grd)
   }//End of main
}//End of class
```

IfExample3a: Checking if any argument is passed at the Command line in Imperative Style

```
var filename = "default.txt"
if (!args.isEmpty)
filename = args(0)
```

This code declares a variable, filename , and initializes it to a default value. It then uses an if expression to check whether any arguments were supplied to the program at the command line. If so, it changes the variable to hold the value specified in the argument list. If no arguments were supplied, it leaves the variable set to the default value.

IfExample3a: Checking if any argument is passed at the Command line in Functional Style.

Remember that Scala's if is an expression that results in a value.

This example shows how you can accomplish the same effect as the previous example, but without using any vars:

```
val filename =
if (!args.isEmpty) args(0)
else "default.txt"
```

This time, the if has two branches. If args is not empty, the initial element, args(0) , is chosen. Else, the default value is chosen. The if expression results in the **chosen value**, and the filename variable is initialized with that value.

The real advantage of this approach is that it uses a **val** instead of a **var** . ***Using a val reads to a functional style,*** and it helps you in much the same way as a **final** variable in Java.

It tells readers of the code that the variable will never change.

A second advantage to using a val instead of a var is that it better supports **equational reasoning**.

The introduced variable is equal to the expression that computes it, assuming the expression has no side effects.

Thus, any time you are about to write the variable name, you could instead write the expression. Instead of `println(filename)` , for example, you could just as well write this:

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

Though you can write it either way, Using vals helps you safely make this kind of refactoring as your code evolves over time.

Using The while Loop

The While loops

Scala’s while loop behaves as in other languages. It has a condition and a body, and the body is executed over and over as long as the condition holds true.

Qn

Write a scala program that will scan through all the integer arguments input at the command line and print them out and their sums.

Solution Discussion.

1. We will need a loop. **Which one**?
2. We should be able to convert the passed arguments from string to integers. **How?**

- For the loop we can strt with a loop
- For conversion we use scala **asInstanceOf**[Int] but this will convert it to an Integerr Object which is not the same as the primitive type (int) which is what we want.
- We can result to java (Remember scala code can also accept java code as they are all

interpreted to a java byte code.). In java we use :

i. **Integer.valueOf()** to Convert a String to an Integer This method returns the string as an **Integer object**. which is not the desired result.

ii. **Integer.parseInt()** to Convert a String to an Integer This method returns the string as a **primitive type** int which is what we want.

Example 1: Writing a scala program that will scan through all the integer arguments input at the command line and print them out and their sums.

```
object LoopExample1 {
def main(args: Array[String]) ={
var i = 0
var sum = 0
while (i < args.length) {
 println(args(i))//prints out the ith command line argument
 sum =sum + Integer.parseInt(args(i)) // You understand this?
  i += 1  //increments i by one.
}
 println(sum)
   }//End of main
}//End of class
```

Explanation
The **args.length** gives the length of the args array.
The while block contains three statements, **each indented two spaces**, the recommended indentation style for Scala.
Note that Java's ++i and i++ don't work in Scala. To increment in Scala, you need to say either i = i + 1 or i += 1 .

Note

- In Scala, as in Java, you must put the **boolean expression** for a **while** or an **if** in parentheses. (In other words, you can't say in Scala things like if i < 10 as you can in a language such as Ruby. We use **if (i < 10)** in Scala.)
- Another similarity to Java is that if a block has only one statement, you can optionally leave off the curly braces,
- Scala does use semicolons to separate statements as in Java, except that in Scala the semicolons are very often optional.

CLASS TASK
Modify the scala program above to now print the marks and print the corresponding grade in a tabular format using a **grading function** that receives marks and return the grade.

Example of Iterate using foreach

One of the main characteristics of a functional language is that functions are first class constructs.

For example, another (far more concise) way to print each command line argument is:

```
args.foreach(arg => println(arg))
```

In this code, you call the **foreach** method on args , and pass in a function.

In this case, you're passing in a function literal that takes one parameter named arg .

The body of the function is println(arg) .

In the previous example, the Scala interpreter infers the type of arg to be String, since String is the element type of the array on which you're calling foreach . If you want to be more explicit, you can mention the **type** name, but you must wrap the argument portion in parentheses (which is the normal form of the syntax anyway):

```
args.foreach((arg: String) => println(arg))
```

What about the ever popular for loop

You may have been accustomed to using the **for** loops in imperative languages such as Java or C.

Scala does not make use of the for loop but provide a **for expression** which is designed for all purposes to be used only a **functional style** .

Scala's **for expression** is a Swiss army knife of iteration. It lets you combine a few simple ingredients in different ways to express a wide variety of iterations.

- Simple uses that enable common tasks such as iterating through a sequence of integers.
- More advanced expressions can iterate over multiple collections of different kinds and can **filter** out elements based on arbitrary conditions, and can produce new collections.

Example 1 : Iterating through the args array and printing the values

```
for (arg <- args)
println(arg)
```

The parentheses after the " for " contain **arg <- args** . To the right of the **<-** symbol is the familiar args array and to the left of <- is " arg ", which a the name of a val variable,

Note It is not a not a var . (Because it is always a val , you just write " arg " by itself, not " val arg ".) Although arg may seem to be a var , because it will get a new value on each iteration, it really is a val : arg can't be reassigned inside the body of the for expression.

Instead, for each element of the args array, a new arg val will be created and initialized to the element value, and the body of the for will be executed.

Example 2: Iterating through a range of integer values

Output

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

```
scala> for (i <- 1 to 4)

println("Iteration "+ I)
```

If you don't want to include the upper bound of the range in the values that

are iterated over, use until instead of to :

```
scala> for (i <- 1 until 4)

println("Iteration "+ i)
```

```
Output

Iteration 1
Iteration 2
Iteration 3
```

Example 3: Iterating through a predefined sequence

```
sscala> val nums = Seq(1,2,3,4)

nums: Seq[Int] = List(1, 2, 3,4)
scala> for (n <- nums) println(n)
```

```
Output

1
2
3
4
```

Example 4: Iterating through and printing a list

```
val students = List(   "Jenifer",     "Micheal",     "James",     "Nancy",     "Regina")

for (s <- students) println(s)
```

`Seq` and **`List`** are two types of linear **collections**. In Scala these collection classes are preferred over `Array`. We will discuss collections later.

Example 3: Printing the files in the current directory

The simplest thing you can do with for is to iterate through all the elements of a collection. In this example we are iterating through an array of files in the current directory (**"."**)

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere)

println(file)
```

The I/O is performed using the Java API.

First, we create a java.io.File on the current directory, "." , and call its **listFiles** method which returns an array of File objects, one per directory and file contained in the current directory. We store the resulting array in the **filesHere** variable

With the **" file <- filesHere "** syntax, which is called a **generator**, we iterate through the elements of filesHere .

In each iteration, a new val named file is initialized with an element value.

The compiler infers the type of file to be File , because filesHere is an Array[File] .

For each iteration, the body of the for expression, println(file) , will be executed.

Because File ’s toString method yields the name of the file or directory, the names of all the files and directories in the current directory will be printed.

Example 3 Filtering Through a Collection

Sometimes you want to filter through a collection down to some subset. You can do this with a **for expression** by adding a **filter**:

A is an **if clause** inside the for ’s parentheses.

Example3 : Lists only those files in the current directory whose names end with “ .scala ”:

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere if file.getName.endsWith(".scala"))

println(file)
```

You could alternatively accomplish the same goal with this code:

```
for (file <- filesHere)

if (file.getName.endsWith(".scala"))

println(file)
```

Example4 : Nested Filtering

To use more filters depending on the application, you can add more **if clauses**.

Example: The code below prints only files and not directories. It does so by adding a filter that checks the file ’s isFile method. (Check java File I/o Methods)

Note that ff you add more than one filter on a **generator**, the filter’s if clauses must be separated by semicolons.

```
for ( file <- filesHere

if file.isFile;

if file.getName.endsWith(".scala")

) println(file)
```

Nested iteration The for Expression

We are used to nested loops in imperative programming and there are cases that require us to nest loops..

With the **for expression** we can use multiple nested generators (that is **multiple <- clauses**) to implement nested loops.

Example

The for expression shown below has two nested loops. The outer loop iterates through filesHere , and the inner loop iterates through fileLines(file) for any file that ends with .scala .

```
def fileLines(file: java.io.File) =scala.io.Source.fromFile(file).getLines.toList
def grep(pattern: String) =
for (
file <- filesHere
if file.getName.endsWith(".scala");
line <- fileLines(file)
if line.trim.matches(pattern)
) println(file +": "+ line.trim)
grep(".*gcd.*")
```

You can also use curly braces instead of parentheses to surround the generators and filters. One advantage to using curly braces is that you can leave off some of the semicolons that are needed when you use parentheses.

Example2.: Command Line Arguments

Type the following into a new file named `Example2.scala` :

```
object Example2 {
   def main(args: Array[String]) ={
      println("Hello, " + args(0)+ "!")
      var i = 0
// Printing argument values
      while (i < args.length) {
      println(args(i))
      i += 1
      }
   }
}
```

then run:

```
$ scala Example2.scala James John Wandera
```

Here "`James John and Wandera`" are passed as a command line argument, which is accessed in the script as args(i) .

The foreach method

Scala provides the **foreach method** for the purpose of iterating over a collection of elements and printing its contents that's available to most collections classes such as sequences, maps, and sets..

Example: To use `foreach` to print the previous list of students:

```
students.foreach(println)
```

Using `for` and `foreach` with Maps

You can also use `for` and `foreach` when working with a Scala `Map` (which is similar to a Java `HashMap`). For example, given this `Map` of movie names and ratings:

```
val ratings = Map(    "James"  -> 40,     "Jane"  -> 50,    "Joseph"  -> 70,)
```

You can print the movie names and ratings using `for` like this:

```
for ((name,rating) <- ratings) println(s"Movie: $name, Rating: $rating")
```

Here's what that looks like in the REPL:

```
scala> for ((name,rating) <- ratings) println(s"Name: $name, Marks: $rating")
```

In this example, `name` corresponds to each *key* in the map, and `rating` is the name that's assigned to each *value* in the map.

You can also print the ratings with `foreach` like this:

```
ratings.foreach {
    case(name, marks) => println(s"key: $name, value: $marks")
}
```

NOTE

The use of the **`for`** keyword and **`foreach`** method in this above cases is an illustration of their as tools for **side effects**. We used them to print the values in the collections to STDOUT using `println`.

The for Expressions

In Scala, the **`for` expression** (written as **for-expression**) is a different use of the **`for`** construct. While a ***for-loop*** is used for side effects (such as printing output), a ***for-expression*** is used to create new collections from existing collections.

Example

Given this list of integers:

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

You can create a new list of integers where all of the values are doubled, like this:

```
val doubledNums = for (n <- nums) yield n * 2
```

That expression can be read as, "For every number `n` in the list of numbers `nums`, double each value, and then assign all of the new values to the variable `doubledNums`."

```
scala> val doubledNums = for (n <- nums) yield n * 2
doubledNums: Seq[Int] = List(2, 4, 6)
```

As the REPL output shows, the new list `doubledNums` contains these values:

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

The result of the **for-expression** is that it creates a new variable named `doubledNums` whose values were created by doubling each value in the original list, `nums`.

Capitalizing a list of strings

You can use the same approach with a list of strings. For example, given this list of lowercase strings:

```
val names = List("adam", "david", "frank")
```

You can create a list of capitalized strings with this for-expression:

```
val ucNames = for (name <- names) yield name.capitalize
```

The REPL shows how this works:

```
scala> val ucNames = for (name <- names) yield name.capitalize
ucNames: List[String] = List(Adam, David, Frank)
```

Success! Each name in the new variable `ucNames` is capitalized.

The `yield` keyword

Using `yield` after `for` is the "secret key" that instruct the compiler to a new collection from the existing collection that is being iterated over in the for-expression."

Using a block of code after `yield`

The code after the `yield` expression can be as long as necessary to solve the current problem. For example, given a list of strings like this:

```
val names = List("_adam", "_david", "_frank")
```

- Imagine that you want to create a new list that has the capitalized names of each person.
- To do that, you first need to remove the underscore character at the beginning of each name, and then capitalize each name.
- To remove the underscore from each name, you call `drop(1)` on each `String`.
- After you do that, you call the `capitalize` method on each string.

Here's how you can use a for-expression to solve this problem:

```
val capNames = for (name <- names) yield {
    val nameWithoutUnderscore = name.drop(1)
    val capName = nameWithoutUnderscore.capitalize
    capName
}
```

If you put that code in the REPL, you'll see this result:

```
capNames: List[String] = List(Adam, David, Frank)
```

A shorter version of the solution

The above example you can also be writen in a more concise approach, which is the Scala style of doing things:

```
val capNames = for (name <- names) yield name.drop(1).capitalize
```

You can also put curly braces around the algorithm, if you prefer:

```
val capNames = for (name <- names) yield { name.drop(1).capitalize }
```

Scala match Expressions

Scala provide for a **`match`** expression.

In the most simple way you can use a `match` expression like a **Java `switch`** statement:

```
// i is an integer
i match {
    case 1  => println("January")
    case 2  => println("February")
    case 3  => println("March")
    case 4  => println("April")
    case 5  => println("May")
    case 6  => println("June")
    case 7  => println("July")
    case 8  => println("August")
    case 9  => println("September")
    case 10 => println("October")
    case 11 => println("November")
    case 12 => println("December")
    // catch the default with a variable so you can print it
    case _  => println("Invalid month")
}
```

As shown, with **a `match`** expression you write a number of `case` statements that you use to match possible values.

Here we match the integer values `1` through `12`. Any other value falls down to the _ case, which is the **catch-all, default case**.

`match` expressions are nice because they also return values, so rather than directly printing a string as in that example, you can assign the string result to a new value:

```
val monthName = i match {
    case 1  => "January"
    case 2  => "February"
    case 3  => "March"
    case 4  => "April"
    case 5  => "May"
    case 6  => "June"
    case 7  => "July"
    case 8  => "August"
    case 9  => "September"
    case 10 => "October"
    case 11 => "November"
    case 12 => "December"
    case _  => "Invalid month"
}
```

Using a `match` expression to yield a result like this is a common use.

Using A Match Expression As The Body Of A Method

This is a Example that that creates a function that uses a match expression.

It takes a `Boolean` value as an input parameter and returning a `String` message.

```
def convertBooleanToStringMessage(bool: Boolean): String = bool match {
    case true => "you said true"
    case false => "you said false"
}
```

The body of the method is just two `case` statements, one that matches `true` and another that matches `false`.

NOTE. Because those are the only possible `Boolean` values, there's no need for a default `case` statement.

This is how you call that method and then print its result:

```
val result = convertBooleanToStringMessage(true)
```

```
println(result)
```

Using a `match` expression as the body of a method is also a common use.

Handling alternate cases

`match` expressions are extremely powerful, and we'll demonstrate a few other things you can do with them.

1. **`match` expressions** let you handle multiple cases in a single `case` statement.

To demonstrate this, imagine that you want to evaluate "boolean equality" like the Perl programming language handles it: a `0` or a blank string evaluates to false, and anything else evaluates to true.

This is how you write a method using a `match` expression that evaluates to true and false in the manner described:

```
def isTrue(a: Any) = a match {
    case 0 | "" => false // Case 0 or empty
    case _ => true
}
```

Because the input parameter **`a`** is defined to be the `Any` type — which is the root of all Scala classes, like `Object` in Java — this method works with any data type that's passed in:

```
scala> isTrue(0)
res0: Boolean = false
scala> isTrue("")
res1: Boolean = false
scala> isTrue(1.1F)
res2: Boolean = true

scala> isTrue(new java.io.File("/etc/passwd"))
res3: Boolean = true
```

The key part of this solution is that this one `case` statement lets both `0` and the empty string evaluate to `false`:

```
case 0 | "" => false
```

Example2:

```
val evenOrOdd = i match {
    case 1 | 3 | 5 | 7 | 9 => println("odd")
    case 2 | 4 | 6 | 8 | 10 => println("even")
    case _ => println("some other number")
}
```

Example3 that shows how to handle multiple strings in multiple `case` statements:

```
cmd match {
    case "start" | "go" => println("starting")
    case "stop" | "quit" | "exit" => println("stopping")
    case _ => println("doing nothing")
}
```

2. Using `If` Expressions In `Case` Statements

Another great thing about `match` expressions is that you can use `if` expressions in `case` statements for powerful pattern matching. In this example the second and third `case` statements both use `if` expressions to match ranges of numbers:

```
count match {
    case 1 => println("one, a lonely number")
    case x if x == 2 || x == 3 => println("two's company, three's a crowd")
    case x if x > 3 => println("4+, that's a party")
    case _ => println("i'm guessing your number is zero or less")
}
```

Scala doesn't require you to use parentheses in the `if` expressions, however you can use them if you think that makes them more readable:

```
count match {
    case 1 => println("one, a lonely number")
    case x if (x == 2 || x == 3) => println("two's company, three's a crowd")
    case x if (x > 3) => println("4+, that's a party")
    case _ => println("i'm guessing your number is zero or less")
}
```

3. You can also write the code on the right side of the => on multiple lines if you think is easier to read. Here's one example:

```
count match {
    case 1 =>
        println("one, a lonely number")
    case x if x == 2 || x == 3 =>
        println("two's company, three's a crowd")
    case x if x > 3 =>
        println("4+, that's a party")
    case _ =>
        println("i'm guessing your number is zero or less")
}
```

Here's a variation of that example that uses curly braces:

```
count match {
    case 1 => {
        println("one, a lonely number")
    }
    case x if x == 2 || x == 3 => {
        println("two's company, three's a crowd")
    }
    case x if x > 3 => {
        println("4+, that's a party")
    }
    case _ => {
        println("i'm guessing your number is zero or less")
    }
}
```

Here are a few other examples of how you can use `if` expressions in `case` statements. First, another example of how to match ranges of numbers:

```
i match {
  case a if 0 to 9 contains a => println("0-9 range: " + a)
  case b if 10 to 19 contains b => println("10-19 range: " + b)
  case c if 20 to 29 contains c => println("20-29 range: " + c)
  case _ => println("Hmmm...")
}
```

Lastly, this example shows how to reference class fields in `if` expressions:

```
stock match {
  case x if (x.symbol == "XYZ" && x.price < 20) => buy(x)
  case x if (x.symbol == "XYZ" && x.price > 50) => sell(x)
  case x => doNothing(x)
}
```

A Case Problem

You need to match one or more patterns in a Scala *match expression*, and the pattern may be a constant pattern, variable pattern, constructor pattern, sequence pattern, tuple pattern, or type pattern.

Solution

Define a `case` statement for each pattern you want to match.

The following method shows examples of many different types of patterns you can use in `match` expressions:

```
def echoWhatYouGaveMe(x: Any): String = x match {

    // constant patterns
    case 0 => "zero"
    case true => "true"
    case "hello" => "you said 'hello'"
    case Nil => "an empty List"
    // sequence patterns
    case List(0, _, _) => "a three-element list with 0 as the first
element"
    case List(1, _*) => "a list beginning with 1, having any number
of elements"
    case Vector(1, _*) => "a vector starting with 1, having any
number of elements"

    // tuples
    case (a, b) => s"got $a and $b"
    case (a, b, c) => s"got $a, $b, and $c"

    // constructor patterns
    case Person(first, "Alexander") => s"found an Alexander, first
name = $first"
    case Dog("Suka") => "found a dog named Suka"

    // typed patterns
    case s: String => s"you gave me this string: $s"
    case i: Int => s"thanks for the int: $i"
    case f: Float => s"thanks for the float: $f"
    case a: Array[Int] => s"an array of int: ${a.mkString(",")}"
    case as: Array[String] => s"an array of strings: $
{as.mkString(",")}"
    case d: Dog => s"dog: ${d.name}"
    case list: List[_] => s"thanks for the List: $list"
    case m: Map[_, _] => m.toString

    // the default wildcard pattern
    case _ => "Unknown"
}
```

You can test this `match` expression in a variety of ways. For the purposes of this example, I created the following object to test the `echoWhatYouGaveMe` method:

```
object LargeMatchTest extends App {

    case class Person(firstName: String, lastName: String)
    case class Dog(name: String)

    // trigger the constant patterns
    println(echoWhatYouGaveMe(0))
    println(echoWhatYouGaveMe(true))
    println(echoWhatYouGaveMe("hello"))
    println(echoWhatYouGaveMe(Nil))
    // trigger the sequence patterns
    println(echoWhatYouGaveMe(List(0,1,2)))
    println(echoWhatYouGaveMe(List(1,2)))
    println(echoWhatYouGaveMe(List(1,2,3)))
    println(echoWhatYouGaveMe(Vector(1,2,3)))

    // trigger the tuple patterns
    println(echoWhatYouGaveMe((1,2)))         // two element tuple
    println(echoWhatYouGaveMe((1,2,3)))       // three element tuple

    // trigger the constructor patterns
    println(echoWhatYouGaveMe(Person("Melissa", "Alexander")))
   println(echoWhatYouGaveMe(Dog("Suka")))

    // trigger the typed patterns
    println(echoWhatYouGaveMe("Hello, world"))
    println(echoWhatYouGaveMe(42))
    println(echoWhatYouGaveMe(42F))
    println(echoWhatYouGaveMe(Array(1,2,3)))
    println(echoWhatYouGaveMe(Array("coffee", "apple pie")))
    println(echoWhatYouGaveMe(Dog("Fido")))
    println(echoWhatYouGaveMe(List("apple", "banana")))
    println(echoWhatYouGaveMe(Map(1->"Al", 2->"Alexander")))

    // trigger the wildcard pattern
    println(echoWhatYouGaveMe("33d"))
}
```

try/catch/finally Expressions

Like Java, Scala has a **try/catch/finally** construct to let you catch and manage exceptions.

The main difference is that for consistency, Scala uses the same syntax that **`match`** expressions use: `case` statements to match the different possible exceptions that can occur.

A try/catch example

In this example, `openAndReadAFile` is a method that does what its name implies: it opens a file and reads the text in it, assigning the result to the variable named `text`:

```
var text = ""
```

```
try {
    text = openAndReadAFile(filename)
} catch {
    case e: FileNotFoundException => println("Couldn't find that file.")
    case e: IOException => println("Had an IOException trying to read that file")
}
```

Scala uses the *java.io.** classes to work with files, so attempting to open and read a file can result in both a `FileNotFoundException` and an `IOException`. Those two exceptions are caught in the `catch` block of this example.

try, catch, and finally

The Scala try/catch syntax also lets you use a `finally` clause, which is typically used when you need to close a resource. Here's an example of what that looks like:

```
try {
    // your scala code here
}
catch {
    case foo: FooException => handleFooException(foo)
    case bar: BarException => handleBarException(bar)
    case _: Throwable => println("Got some other kind of Throwable exception")
} finally {
    // your scala code here, such as closing a database connection
    // or file handle
}
```

Revision QUESTIONS

In an organization an employee must pay tax which is based on the employee total earnings less the legal deductions. The PAYEE is based on the following brackets

Bracket	**Tax**
x<=20000	0%
20000<x<=30000	10%
30000<x<=40000	20%
40000<x<=50000	30%
50000<x and above	40%

1. Use the if/else structure to create an anonymous function that receives an employee taxable income and return the tax payable.
2. Consider a case of 10 employees in an organization each earning taxable incomes ranging from 35000 and 350000. Write a program that receives the taxable incomes for the ten employees from the command line and for each employee print the income and the corresponding PAYEE. Hint use the while loop.
3. Modify the program above (2) such that it now randomly generates the taxable incomes for the ten employees .
4. Modify the program above (2) such that it now works with any number of employees. Use

foreach method.

5. Write a Scala program to check whether a given positive number is a multiple of 3 or a multiple of 7 or a multiple of 11
6. Write a Scala program to check the gender of two people such one of one is a male and the other a female.
7. Write a Scala program to check two given integers whether either of them is in the range 1..100 inclusive
8. Write a Scala program to check whether three given integer values are in the range 20..70 inclusive. Return true if 1 or more of them are in the said range otherwise false.
9. Write a Scala program to check whether two given positive integers have the same last digit.

Scala - Strings

Every application should be able to handle strings. In deed string processing is one of the most common in all applications.

TASKS

Name application areas that require string processing.

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.

In Scala, as in Java, a string is an **immutable** object, that is, an object that cannot be modified. On the other hand, objects that can be modified, like arrays, are called **mutable** objects. We will discuss some basic string processing and also present important methods of **java.lang.String** class.

Creating a String

The following code can be used to create a String −

```
var greeting = "Hello world!";
or
var greeting:String = "Hello world!";
```

Whenever compiler encounters a string literal in the code, it creates a String object with its value, in this case, "Hello world!". String keyword can also be given in alternate declaration as shown above.

Example

```
object StringDemo1 {
   val greeting: String = "Hello, world!"
   def main(args: Array[String]) {
      println( greeting )
   }
}
```

As mentioned earlier, String class is immutable. String object once created cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters then use **String Builder Class** available in Scala!.

String Length

Methods used to obtain information about an object are known as **accessor** methods. One accessor method that can be used with strings is the **length()** method, which returns the number of characters contained in the string object.

Use the following code segment to find the length of a string −

Example

```
object StringDemo2 {
   def main(args: Array[String]) {
      var palindrome = "Dot saw I was Tod";
      var len = palindrome.length();
      println( "String Length is : " + len );
   }
}
```

Concatenating Strings

The String class includes a method for concatenating two strings −

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in −

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in −

```
"Hello," + " world" + "!"
```

Which results in −

```
"Hello, world!"
```

The following lines of code to find string length.

Example

```
object StringDemo3 {
   def main(args: Array[String]) {
      var str1 = "Dot saw I was ";
      var str2 =  "Tod";
      println("Dot " + str1 + str2);
   }
}
```

Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Try the following example program, which makes use of printf() method −

Example

```
object StringDemo4 {
   def main(args: Array[String]) {
      var floatVar = 12.456
      var intVar = 2000
      var stringVar = "Hello, Scala!"
      var fs = printf("The value of the float variable is " + "%f, while
the value of the integer " + "variable is %d, and the string" + "is %s",
floatVar, intVar, stringVar);

      println(fs)
   }
}
```

Output

```
The value of the float variable is 12.456000,
while the value of the integer variable is 2000,
and the string is Hello, Scala!()
```

String Interpolation

String Interpolation is the new way to create Strings in Scala programming language. This feature supports the versions of Scala-2.10 and later. String Interpolation: The mechanism to embed variable references directly in process string literal.

There are three types (interpolators) of implementations in String Interpolation.

The ‘s’ String Interpolator

The literal ‘s’ allows the usage of variable directly in processing a string, when you prepend ‘s’ to it. Any String variable with in a scope that can be used with in a String. The following are the different usages of ‘s’ String interpolator.

The following example code snippet for the implementation of ‘s’ interpolator in appending String variable ($name) to a normal String (Hello) in println statement.

```
val name = “James”
println(s “Hello, $name”) //output: Hello, James
```

String interpolater can also process arbitrary expressions. The following code snippet for Processing a String (1 + 1) with arbitrary expression (${1 + 1}) using ‘s’ String interpolator. Any arbitrary expression can be embedded in ‘${}’.

```
println(s “1 + 1 = ${1 + 1}”) //output: 1 + 1 = 2
```

Try the following example program of implementing ‘s’ interpolator.

Example

```
object StringDemo5 {
   def main(args: Array[String]) {
      val name = "James"
      println(s"Hello, $name")
      println(s"1 + 1 = ${1 + 1}")
   }
}
```

Output

```
Hello, James
1 + 1 = 2
```

The ‘ f ’ Interpolator

The literal ‘f’ interpolator allows to create a formatted String, similar to printf in C language. While using ‘f’ interpolator, all variable references should be followed by the **printf** style format specifiers such as `%d, %i, %f, etc.`

Let us take an example of append floating point value (height = 1.9d) and String variable (name = “James”) with normal string. The following code snippet of implementing ‘f’ Interpolator. Here $name %s to print (String variable) James and $height%2.2f to print (floating point value) 1.90.

```
val height = 1.9d
val name = "James"
println(f"$name%s is $height%2.2f meters tall") //James is 1.90 meters tall
```

It is type safe (i.e.) the variable reference and following format specifier should match otherwise it is showing error. The ‘ f ’ interpolator makes use of the String format utilities (format specifiers) available in Java. By default means, there is no % character after variable reference. It will assume as %s (String).

‘raw’ Interpolator

The ‘raw’ interpolator is similar to ‘s’ interpolator except that it performs no escaping of literals within a string. The following code snippets in a table will differ the usage of ‘s’ and ‘raw’ interpolators. In outputs of ‘s’ usage ‘\n’ effects as new line and in output of ‘raw’ usage the ‘\n’ will not effect. It will print the complete string with escape letters.

Example ‘s’ interpolator usage

```
object Demo {
   def main(args: Array[String]) {
    println(s"Result = \n a \n b")
   }
}
```

Output −

```
Result =
a
b
```

Example ‘raw’ interpolator usage

```
object Demo {
   def main(args: Array[String]) {
   println(raw"Result = \n a \n b")
   }
}
```

Output −

```
Result = \n a \n b
```

String Methods

Following is the list of methods defined by **java.lang.String** class and can be used directly in your Scala programs

Sr.No	Methods with Description
1	**char charAt(int index):**Returns the character at the specified index.
2	**int compareTo(Object o):** Compares this String to another Object.
3	**int compareTo(String anotherString):** Compares two strings lexicographically.
4	**int compareToIgnoreCase(String str):** Compares two strings lexicographically, ignoring case differences. I.e In terms of which is longer or shoter or equal in length
5	**String concat(String str):** Concatenates the specified string to the end of this string.
6	**boolean contentEquals(StringBuffer sb):** Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	**static String copyValueOf(char[] data):** Returns a String that represents the character sequence in the array specified.
8	**static String copyValueOf(char[] data, int offset, int count):** Returns a String that represents the character sequence in the array specified.
9	**boolean endsWith(String suffix):**Tests if this string ends with the specified suffix.

10 **boolean equals(Object anObject):**Compares this string to the specified object.

11 **boolean equalsIgnoreCase(String anotherString):**Compares this String to another String, ignoring case considerations.

12 **byte getBytes():**Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

13 **byte[] getBytes(String charsetName):**Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

14 **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin):**Copies characters from this string into the destination character array.

15 **int hashCode():**Returns a hash code for this string.

16 **int indexOf(int ch):**Returns the index within this string of the first occurrence of the specified character.

17 **int indexOf(int ch, int fromIndex):**Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

18 **int indexOf(String str):** Returns the index within this string of the first occurrence of the specified substring.

19 **int indexOf(String str, int fromIndex):** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

20 **String intern():** Returns a canonical representation for the string object.

21 **int lastIndexOf(int ch):** Returns the index within this string of the last occurrence of the specified character.

22 **int lastIndexOf(int ch, int fromIndex):** Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

23 **int lastIndexOf(String str):** Returns the index within this string of the rightmost occurrence of the specified substring.

24 **int lastIndexOf(String str, int fromIndex):** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

25 **int length():** Returns the length of this string.

26 **boolean matches(String regex):** Tells whether or not this string matches the given regular expression.

27 **boolean regionMatches(boolean ignoreCase, int toffset, String other, int offset, int len):** Tests if two string regions are equal.

28 **boolean regionMatches(int toffset, String other, int offset, int len):** Tests if two string regions are equal.

29 **String replace(char *oldChar*, char newChar):** Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

30 **String replaceAll(String regex, String replacement:** Replaces each substring of this string that matches the given regular expression with the given replacement.

31 **String replaceFirst(String regex, String replacement):** Replaces the first substring of this string that matches the given regular expression with the given replacement.

32 **String[] split(String regex):** Splits this string around matches of the given regular expression.

33 **String[] split(String regex, int limit):** Splits this string around matches of the given regular expression.

34 **boolean startsWith(String prefix):** Tests if this string starts with the specified prefix.

35 **boolean startsWith(String prefix, int toffset):** Tests if this string starts with the specified prefix beginning a specified index.

36 **CharSequence subSequence(int beginIndex, int endIndex):** Returns a new character sequence that is a subsequence of this sequence.

37 **String substring(int beginIndex):** Returns a new string that is a substring of this string.

38 **String substring(int beginIndex, int endIndex):** Returns a new string that is a substring of this string.

39 **char[] toCharArray():** Converts this string to a new character array.

40 **String toLowerCase():** Converts all of the characters in this String to lower case using the rules of the default locale.

41 **String toLowerCase(Locale locale):** Converts all of the characters in this String to lower case using the rules of the given Locale.

42 **String toString():** This object (which is already a string!) is itself returned.

43 **String toUpperCase():** Converts all of the characters in this String to upper case using the rules of the default locale.

44 **String toUpperCase(Locale locale):** Converts all of the characters in this String to upper case using the rules of the given Locale.

45 **String trim():** Returns a copy of the string, with leading and trailing whitespace omitted.

46 **static String valueOf(primitive data type x):** Returns the string representation of the passed data type argument.

Class Exercise

You are given the string " **Scala is just as cool in Programming Just as its Name is to Pronounce** "

1. Write a Scala program to print the number characters in the string above.
2. Write a Scala program to print the number characters in the string above that are uppercase.
3. Write a Scala program to print the number occurrence of every word in the string above.
4. Write a Scala program to print the word that is most common in the string above.
5. Write a Scala program to print the characters in the string above together with the corresponding character index position.
6. Write a Scala program to print the string above reversed at the character level.
7. Write a Scala program to print the string above reversed at the word level.
8. Write a Scala program to print the string above with every word characters reversed.
9. Write a Scala program to print a substring formed by the first character of every word in the string.
10. Write a Scala program to get the difference in length between two strings. If the difference is greater than 10 truncate the longer string to have the same length as the shorter one.
11. Write a Scala program to create a new string where '*' is added to the front of a given string. If

the string already begins with '*' return the string unchanged.

12. Write a Scala program to create a new string where '*’ and “#” are added to the front and end of a given string. If the string already begins with '*' and ends with “*”, return the string unchanged.

Scala - Arrays

Scala just likeJava provides array data structure, which stores a fixed-size sequential collection of elements of the same type.

An array is used to store a collection of data of the same type, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

We discuss how to declare array variables, create arrays, and process arrays using indexed variables. The index of the first element of an array is the number zero and the index of the last element of an array of size N is N-1.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array and you must specify the type of array the variable can reference.

The following is the syntax for declaring an array variable.

Syntax

```
var z:Array[String] = new Array[String](3)
or
var z = new Array[String](3)
```

Here, z is declared as an array of Strings that may hold up to three elements. Values can be assigned to individual elements or get access to individual elements, it can be done by using commands like the following −

Command

```
z(0) = "Zara"; z(1) = "Nuha"; z(4/2) = "Ayan"
```

Here, the last example shows that in general the index can be any expression that yields a whole number.

There is one more way of defining an array −

```
var z = Array("Zara", "Nuha", "Ayan")
```

Processing Arrays

When processing array elements, we often use loop contol structures because all of the elements in an array are of the same type and the size of the array is known.

Example

```
object ArrayDemo1 {
   def main(args: Array[String]) {
      var myList = Array(1.9, 2.9, 3.4, 3.5)

      // Print all the array elements
      for ( x <- myList ) {
         println( x )
      }

      // Summing all elements
      var total = 0.0;
      for ( i <- 0 to (myList.length - 1)) {
         total += myList(i);
      }
      println("Total is " + total);

      // Finding the largest element
      var max = myList(0);

      for ( i <- 1 to (myList.length - 1) ) {
         if (myList(i) > max) max = myList(i);
      }
      println("Max is " + max);
   }
}
```

Output

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Scala does not directly support various array operations and provides various methods to process arrays in any dimension. If you want to use the different methods then it is required to import **Array._** package.

Multi-Dimensional Arrays

There are many situations where you would need to define and use multi-dimensional arrays (i.e., arrays whose elements are arrays). For example, matrices and tables are examples of structures that can be realized as two-dimensional arrays.

Multi-dimensional arrays can be created from two dimensional to three, four and many more dimensional array according to your need.

Syntax

```
var arrayName = Array.ofDim[ArrayType](NoOfRows,NoOfColumns) or
var arrayName = Array(Array(element...), Array(element...), ...)
```

Example using ofDim

```
class ArrayExample{
    var arr = Array.ofDim[Int](2,2)// Creating multidimensional array
    arr(1)(0) = 15     // Assigning value
    def show(){
        for(i<- 0 to 1){// Traversing elements by using loop
           for(j<- 0 to 1){
                print(" "+arr(i)(j))
            }
            println()
        }
// Access elements by using index
        println("Third Element = "+ arr(1)(1))
    }
}

object MainObject{
    def main(args:Array[String]){
        var a = new ArrayExample()
        a.show()
    }
}
```

Output:

The following is the example of defining a two-dimensional array −

```
var myMatrix = ofDim[Int](3,3)
```

This is an array that has three elements each being an array of integers that has three elements.

Try the following example program to process a multi-dimensional array −

Example

```
import Array._
object ArrayDemo2 {
   def main(args: Array[String]) {
      var myMatrix = ofDim[Int](3,3)
      // build a matrix
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            myMatrix(i)(j) = j;
         }
      }
      // Print two dimensional array
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            print(" " + myMatrix(i)(j));
         }
         println();
      }
   }
}
```

Output

```
0 1 2
0 1 2
0 1 2
```

Multidimensional Array by using Array of Array

Apart from ofDim you can also create multidimensional array by using array of array. In this example, we have created multidimensional array by using array of array.

```
class ArrayExample{

    var arr = Array(Array(1,2,3,4,5), Array(6,7,8,9,10))
    def show(){
        for(i<- 0 to 1){ // Traversing elements using loop
```

```
           for(j<- 0 to 4){
                print(" "+arr(i)(j))
            }
            println()
        }
    }
}

object MainObject{
    def main(args:Array[String]){
        var a = new ArrayExample()
        a.show()
    }
}
```

Addition of Two Matrix Example

You can manipulate array elements in scala. Here, we are adding two array elements and storing result into third array.

```
class ArrayExample{

  var arr1 = Array(Array(1,2,3,4,5), Array(6,7,8,9,10))   // Creating multidimensional array
  var arr2 = Array(Array(1,2,3,4,5), Array(6,7,8,9,10))
  var arr3 = Array.ofDim[Int](2,5)
  def show(){
    for(i<- 0 to 1){             // Traversing elements using loop
      for(j<- 0 to 4){
        arr3(i)(j) = arr1(i)(j)+arr2(i)(j)
        print(" "+arr3(i)(j))
      }
      println()
    }
  }
}

object MainObject{
  def main(args:Array[String]){
    var a = new ArrayExample()
    a.show()
  }
}
```

Concatenate Arrays

The example below makes use of concat() method to concatenate two arrays. You can pass more than one array as arguments to concat() method.

Example

```
import Array._
object Demo {
   def main(args: Array[String]) {
      var myList1 = Array(1.9, 2.9, 3.4, 3.5)
      var myList2 = Array(8.9, 7.9, 0.4, 1.5)
      var myList3 =  concat( myList1, myList2)
      // Print all the array elements
      for ( x <- myList3 ) {
         println( x )
      }
   }
}
```

What is the Output?

Create Array with Range

Use of range() method to generate an array containing a sequence of increasing integers in a given range. You can use final argument as step to create the sequence; if you do not use final argument, then step would be assumed as 1.

Example : Creating an array of range (10, 20, 2) means creating an array with elements between 10 and 20 and range difference 2. Elements in the array will the be 10, 12, 14, 16, and 18.

Example: range (10, 20). Here range difference is not given so by default it assumes 1 element. It create an array with the elements in between 10 and 20 with range difference 1. Elements in the array are 10, 11, 12, 13, …, and 19.

Example

```
import Array._
object Demo {
   def main(args: Array[String]) {
      var myList1 = range(10, 20, 2)
      var myList2 = range(10,20)
      // Print all the array elements
      for ( x <- myList1 ) {
         print( " " + x )
      }
      println()
      for ( x <- myList2 ) {
         print( " " + x )
      }
   }
}
```

What will be the output of the program.

Scala Passing Array into Function

Just as in java you can pass array as an argument to function during function call.

Example

```
class ArrayExample{
// A functions that recieves an array as an argument
    def show(arr:Array[Int])={
        for(a<-arr)      // Traversing array elements
            println(a)
        println("Third Element = "+ arr(2))        // Accessing elements by using index
    }
}
// Main Class
object MainObject{
    def main(args:Array[String])={
        var arr = Array(1,2,3,4,5,6)// creating single dim array
        var a = new ArrayExample() // Instantiating an object
        a.show(arr)// passing array as an argument in the function
    }
}
```

Scala Array Methods

Following are some of the important methods, you can use with array. As shown above, you would have to import **Array._** package before using any of the mentioned methods.

Sr.No	Methods with Description
1	**def apply(x: T, xs: T*): Array[T]:** Creates an array of T objects, where T can be Unit,

Double, Float, Long, Int, Char, Short, Byte, Boolean.

2 **def concat[T](xss: Array[T]*): Array[T]:** Concatenates all arrays into a single array.

def copy(src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int): Unit

3 Copy one array to another. Equivalent to Java's System.arraycopy(src, srcPos, dest, destPos, length).

4 **def empty[T]: Array[T]:** Returns an array of length 0

5 **def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]:** Returns an array containing repeated applications of a function to a start value.

6 **def fill[T](n: Int)(elem: => T): Array[T]:** Returns an array that contains the results of some element computation a number of times.

7 **def fill[T](n1: Int, n2: Int)(elem: => T): Array[Array[T]]:** Returns a two-dimensional array that contains the results of some element computation a number of times.

8 **def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]:** Returns an array containing repeated applications of a function to a start value.

9 **def ofDim[T](n1: Int): Array[T]:** Creates array with given dimensions.

10 **def ofDim[T](n1: Int, n2: Int): Array[Array[T]]:** Creates a 2-dimensional array

11 **def ofDim[T](n1: Int, n2: Int, n3: Int): Array[Array[Array[T]]]:** Creates a 3-dimensional array

12 **def range(start: Int, end: Int, step: Int): Array[Int]:** Returns an array containing equally spaced values in some integer interval.

13 **def range(start: Int, end: Int): Array[Int]:** Returns an array containing a sequence of increasing integers in a range.

14 **def tabulate[T](n: Int)(f: (Int)=> T): Array[T]:** Returns an array containing values of a given function over a range of integer values starting from 0.

15 **def tabulate[T](n1: Int, n2: Int)(f: (Int, Int) => T): Array[Array[T]]:** Returns a two-dimensional array containing values of a given function over ranges of integer values starting from 0.

Scala - Regular Expressions

We now discuss how Scala supports regular expressions through **Regex** class available in the scala.util.matching package.

Try the following example program where we will try to find out word **Scala** from a statement.

Example : find out the word Scala from a statement

```
import scala.util.matching.Regex

object RegexDemo1 {
   def main(args: Array[String]) {
      val pattern = "Scala".r
      val str = "Scala is Scalable and cool"
      println(pattern findFirstIn str)
   }
}
```

Output

```
Some(Scala)
```

We create a String and call the **r( )** method on it. Scala implicitly converts the String to a RichString and invokes that method to get an instance of Regex.

To find a first match of the regular expression, simply call the **findFirstIn()** method.

If instead of finding only the first occurrence we would like to find all occurrences of the matching word, we can use the **findAllIn( )** method which will return a collection of all matching words.

You can make use of the **mkString( )** method to concatenate the resulting list and you can use a **pipe (|)** to search small and capital case of Scala and you can use **Regex** constructor instead or **r()** method to create a pattern.

Try the following example program.

Example

```
import scala.util.matching.Regex
object RegexDemo2 {
   def main(args: Array[String]) {
      val pattern = new Regex("(S|s)cala")
      val str = "Scala is scalable and cool"
           println((pattern findAllIn str).mkString(","))
   }
}
```

Output

```
Scala,scala
```

If you would like to replace matching text, we can use **replaceFirstIn( )** to replace the first match or **replaceAllIn( )** to replace all occurrences.

Example

```
object RegexDemo3 {
   def main(args: Array[String]) {
      val pattern = "(S|s)cala".r
      val str = "Scala is scalable and cool"

      println(pattern replaceFirstIn(str, "Java"))
   }
}
```

Output

```
Java is scalable and cool
```

Forming Regular Expressions

Scala inherits its regular expression syntax from Java, which in turn inherits most of the features of Perl. Here are just some examples that should be enough as refreshers −

Following is the table listing down all the regular expression Meta character syntax available in Java.

Subexpression	Matches
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
\\A	Beginning of entire string
\\z	End of entire string
\\Z	End of entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more of the previous thing
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.

a|b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?: re)	Groups regular expressions without remembering matched text.
(?> re)	Matches independent pattern without backtracking.
\\w	Matches word characters.
\\W	Matches nonword characters.
\\s	Matches whitespace. Equivalent to [\t\n\r\f].
\\S	Matches nonwhitespace.
\\d	Matches digits. Equivalent to [0-9].
\\D	Matches nondigits.
\\A	Matches beginning of string.
\\Z	Matches end of string. If a newline exists, it matches just before newline.
\\z	Matches end of string.
\\G	Matches point where last match finished.
\\n	Back-reference to capture group number "n"
\\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\\B	Matches nonword boundaries.
\\n, \\t, etc.	Matches newlines, carriage returns, tabs, etc.
\\Q	Escape (quote) all characters up to \\E
\\E	Ends quoting begun with \\Q

Regular-Expression Examples

Example	Description
.	Match any character except newline
[Rr]uby	Match "Ruby" or "ruby"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit
\\d	Match a digit: [0-9]
\\D	Match a nondigit: [^0-9]
\\s	Match a whitespace character: [\t\r\n\f]
\\S	Match nonwhitespace: [^ \t\r\n\f]
\\w	Match a single word character: [A-Za-z0-9_]

\\W	Match a nonword character: [^A-Za-z0-9_]
ruby?	Match "rub" or "ruby": the y is optional
ruby*	Match "rub" plus 0 or more ys
ruby+	Match "rub" plus 1 or more ys
\\d{3}	Match exactly 3 digits
\\d{3,}	Match 3 or more digits
\\d{3,5}	Match 3, 4, or 5 digits
\\D\\d+	No group: + repeats \\d
(\\D\\d)+/	Grouped: + repeats \\D\d pair
([Rr]uby(,)?)+	Match "Ruby", "Ruby, ruby, ruby", etc.

Note − that every backslash appears twice in the string above. This is because in Java and Scala a single backslash is an escape character in a string literal, not a regular character that shows up in the string. So instead of '\', you need to write '\\' to get a single backslash in the string.

Example

```
import scala.util.matching.Regex
object RegexDemo4 {
   def main(args: Array[String]) {
      val pattern = new Regex("abl[ae]\\d+")
      val str = "ablaw is able1 and cool"

      println((pattern findAllIn str).mkString(","))
   }
}
```

Output

```
able1
```

Exercise Questions