Scala Class Note Book

Syllabus :-

1.Scala – Overview:

The Scala programming language is a newer, very interesting language, with a lot of new features compared to Java. The reason Scala is interesting to Java programmers is, that Scala is compiled to run on the Java Virtual Machine. In other words, Scala is compiled into Java bytecodes. This also means that you can use all Java classes in your Scala code. Even the Java classes you have developed yourself. This makes a transition to Scala cheaper, since a lot of Java code can be reused.

Some of the new, interesting features in Scala are closures, functions as objects, traits, concurrency mechanisms inspired by Erlang, and its support for DSL's (Domain Specific Languages).

Scala runs on the Java Virtual Machine

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 runtime platform. If you or your organization has standardized on Java, Scala is not going to be a total stranger in your architecture. It's a different language, but the same runtime.

Scala can Execute Java Code

Since Scala is compiled into Java Byte Bode, it was a no-brainer for the Scala language designer (Martin Odersky) to enable Scala to be able to call Java code from a Scala program. You can thus use all the classes of the Java SDK's in Scala, and also your own, custom Java classes, or your favourite Java open source projects.

Scala Has a Compiler, Interpreter and Runtime

Scala has both a compiler and an interpreter which can execute Scala code.

The Scala compiler compiles your Scala code into Java Byte Code which can then be executed by the scala command. The scala command is similar to the java command, in that it executes your compiled Scala code.

Since the Scala compiler can be a bit slow to startup, Scala has a compiler daemon you can run. This daemon keeps running, even when not compiling Scala code. You can then instruct the daemon to compile Scala code for you at will. This saves you the Scala compiler startup overhead when compiling.

The Scala interpreter runs your Scala code directly as it is, without you needing to compile it. The Scala interpreter may come in handy as a Scala script interpreter, a bit like a shell script interpreter on a Unix platform.

Scala Features

Scala has a set of features which differ from Java. Some of these are:

1. All types are objects.
2. Type inference.
3. Functions are objects.
4. Domain specific language (DSL) support.
5. Traits.
6. Closures.
7. Concurrency support inspired by Erlang.

Scala for the Web

One of the popular Scala web frameworks is called Lift. You can find it here:

- The Lift Framework

Other Frameworks are Play, Bowler .

Differences between Scala and Java

1) First and Major difference you will notice between Scala and Java is succinct and concise code. Scala drastically reduces the number of lines from a Java application by making clever use of type inference, treating everything as an object, function passing, and several other features.

2) Scala is designed to express common programming patterns in an elegant, concise and type-safe way. The language itself encourage you to write code in immutable style, which makes applying concurrency and parallelism easily.

3) One difference, which some might not notice is learning curve. Scala has a steep learning curve as compared to Java, my opinion may be slightly biased because I am from Java background, but with so much happening with little code, Scala can be really tricky to predict.

4) One of Scala's cool feature is built-in lazy evaluation, which allows deferring time-consuming computation until absolutely needed and you can do this by using a keyword called "lazy" .

5) Someone can argue that Java is more readable than Scala, because of really nested code in Scala. Since you can define functions inside the function, inside other functions, inside of an object inside of a class. The code can be very nested. Though sometimes it may improve clarity, but if written poorly it can be really tricky to understand.

6) One more difference between Scala and Java is that Scala supports Operator overloading. You can overload any operator in Java and you can also create new operators for any type, but as you already know, Java doesn't support Operator Overloading.

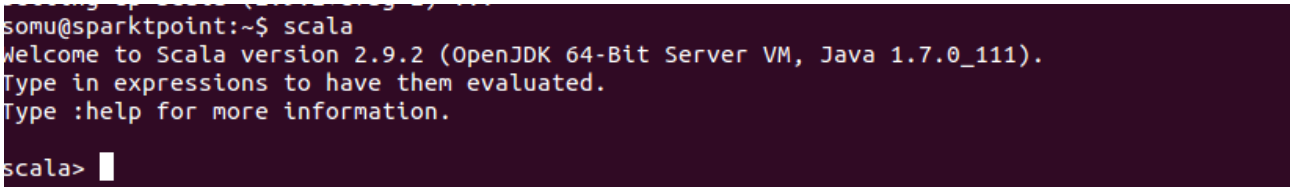
7) Another major difference between Java and Scala is that functions are objects in Java. Scala treats any method or function as they are variables. When means, you can pass them around like Object. You might have seen the code, where one Scala function is accepting another function. In fact, this gives the language enormous power.

2.Scala – Environment / Installation Steps:

The below instructions are for Linux machine

1. Make sure you have Java installed.
2. Download Scala from:http://www.scala-lang.org/download/
3. Extract the .tgz file.
4. Move directory to your /usr/bin/scala
5. Add /usr/bin/scala/bin to your /etc/paths file.
6. Reload your terminal.
7. Scala you can install linux command Line. Help of this command.

```
somu@sparktpoint:~$ sudo apt-get install scala
```



Run scala in Terminal

>scala

Executing Scala codes:

Using the interactive interpreter

The easiest way to execute a single line of Scala code is to use the interactive interpreter with which statements and definitions are evaluated on the fly. We start the interactive interpreter using the Scala command line tool “scala” which is located in the bin folder in the folder where Scala is installed. With the proper environment variable in place you can just type “scala” to start it. Using the interactive interpreter we can do our first Hello world by using the println method.

C:\Users\somu>scala

Welcome to Scala version 2.8.0.RC7 (Java HotSpot(TM) Client VM, Java 1.6.0_20).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello world")
Hello world

scala>

If a line of code that we type in to the interpreter doesn’t cause a console output the interpreter will print out the type and value of the statement.

scala> 1+1
res1: Int = 2

To quite the interpreter we type exit.

scala> exit

C:\Users\somu>

Executing Scala code as a script

Another way to execute Scala code is to type it into a text file and save it with a name ending with ".scala". We can then execute that code by typing "scala filename". For instance, we can create a file named hello.scala with our Hello world in it:

To execute it we specify the filename as a parameter to the Scala command line tool.

```
>scala hello.scala
Hello world!
```

Comments in Scala

Single line comments :

single line comments begin with two forward slashes (//).

Multi-line Comments

/* and ends with */.

3. Scala – Basics:

If you have a good understanding on Java, then it will be very easy for you to learn Scala
The biggest syntactic difference between Scala and Java is that the ';' line end character is optional.
Case Sensitivity – Scala is case-sensitive, which means identifier Hello and hello would have different meaning in Scala.

- Class Names – For all class names, the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

  Example – class MyFirstScalaClass.

- Method Names – All method names should start with a Lower Case letter. If multiple words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

  Example – def myMethodName()

- Program File Name – Name of the program file should exactly match the object name. When saving the file you should save it using the object name (Remember Scala is case-sensitive) and append '.scala' to the end of the name. (If the file name and the object name do not match your program will not compile).

  Example – Assume 'HelloWorld' is the object name. Then the file should be saved as 'HelloWorld.scala'.

- def main(args: Array[String]) – Scala program processing starts from the main() method

  which is a mandatory part of every Scala Program.

Scala Identifiers

All Scala components require names. Names used for objects, classes, variables and methods are called identifiers. A keyword cannot be used as an identifier and identifiers are case-sensitive. Scala supports four types of identifiers.

Alphanumeric Identifiers

An alphanumeric identifier starts with a letter or an underscore, which can be followed by further letters, digits, or underscores. The '$' character is a reserved keyword in Scala and should not be used in identifiers.

Following are legal alphanumeric identifiers –

age, salary, _value, __1_value

Following are illegal identifiers –

$salary, 123abc, -salary

Operator Identifiers

An operator identifier consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #.

Following are legal operator identifiers –

+ ++ ::: <?> :>

The Scala compiler will internally "mangle" operator identifiers to turn them into legal Java identifiers with embedded $ characters. For instance, the identifier :-> would be represented internally as $colon$minus$greater.

Mixed Identifiers

A mixed identifier consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier.

Following are legal mixed identifiers –

unary_+, myvar_=

Here, unary_+ used as a method name defines a unary + operator and myvar_= used as method name defines an assignment operator (operator overloading).

Literal Identifiers

A literal identifier is an arbitrary string enclosed in back ticks (` . . . `).

Following are legal literal identifiers –

`x` `<clinit>` `yield`

Scala Keywords

The following list shows the reserved words in Scala. These reserved words may not be used as constant or variable or any other identifier names.

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	Null
object	override	package	private
protected	return	sealed	super
this	throw	trait	Try
true	type	val	Var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

4.Scala – Variables

Scala variables come in two shapes. Values and variables. A value variable is really just a constant, meaning once assigned, you cannot change its value. It is immutable in other words. A regular variable on the other hand, is mutable, meaning you can change its value.

Declaring var and val Variables

Here is how you declare these two types of variables:

```
var myVar : Int = 0;

val myVal : Int = 1;
```

The first variable, myVar, is declared using the keyword var. This means that it is a variable that can change value.

The second variable, myVal, is declared using the keyword val. This means that it is a value, a constant, that cannot change value once assigned.

Variable Types and Type Inference

The examples earlier in this text specified the type of the variable after the variable name, separated by a colon. In the example below I have repeated the declarations from earlier, and marked the type declarations in bold:

```
var myVar : Int = 0;

val myVal : Int = 1;
```

The type of a variable is specified after the variable name, and before any initial value.

When you assign an initial value to a variable, the Scala compiler can figure out the type of the

variable based on the value assigned to it. This is called type inference. Therefore, you could write these variable declarations like this:

```
var myVar = 0;
val myVal = 1;
```

If, however, you did not assign an initial value to the variable, the compiler cannot figure out what type it is. Therefore, you have to explicitly specify the type if you do not assign an initial value to the variable. Here is how it looks:

```
var myVar :Int;
val myVal :Int;
```

Fields, Parameters and Local Variables

Variables in Scala can exist in 3 different roles. As fields, as method parameters and as local variables. There are no static variables in Scala.

Fields are variables that belong to an object. The fields are accessible from inside every method in the object. Fields can also be accessible outside the object, depending on what access modifiers the field is declared with. Fields can be both val's and var's.

Method parameters are variables which values are passed to a method when the method is called. Method parameters are only accessible from inside the method - but the objects passed in may be accessible from the outside, if you have a reference to the object from outside the method. Method parameters are always val's.

Local variables are variables declared inside a method. Local variables are only accessible from inside the method, but the objects you create may escape the method if you return them from the method. Local variables can be both var's and val's.

Examples for variables:

Assign multiple variables

Using Tuples, it is possible to assign multiple values to multiple variables (either var or val).

```
var x, y, z = 0

var c, python, java = false

println(x, y, z, c, python, java)
```

```
scala> var (x, y, z, c, python, java) = (1, 2, 3, true, false, "no!")
x: Int = 1
y: Int = 2
z: Int = 3
c: Boolean = true
python: Boolean = false
java: String = no!

scala> println(x, y, z, c, python, java)
(1,2,3,true,false,no!)

scala>
```

5. Scala Data Types

Scala comes with the following built-in data types which you can use for your Scala variables:

Type	Value Space
Boolean	true or false
Byte	8 bit signed value
Short	16 bit signed value
Char	16 bit unsigned Unicode character
Int	32 bit signed value
Long	64 bit signed value
Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float
String	A sequence of characters

Here is an example of how to declare a variable to be of a certain type:

var myInt : Int

var myString : String

Multi-Line Strings

A multi-line string literal is a sequence of characters enclosed in triple quotes """ ... """.

Example:

"""the present string

spans three
lines."""

6. Classes & 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. Through the object you can use all functionalities of the defined class.

Scala is, among other things, an object oriented language. This means that you can define classes in Scala, and instantiate objects of these classes. I expect that you know the basics of object oriented programming, when reading this text.

A Scala class is a template for Scala objects. That means, that a class defines what information objects of that class holds, and what behaviour (methods) it exposes. A class can contain information about:

- Fields
- Constructors
- Methods

- Superclasses (inheritance)
- Interfaces implemented by the class
- etc.

In this text I will focus on just the fields, constructors and methods. The other aspects of Scala classes will be covered in separate texts.

The Basic Class Definition

Declaring and instantiating classes

Here is a simple class definition in Scala:

```
class MyClass {

}
```

This class is not very interesting. I will add some more to it throughout this text.

Fields

A field is a variable that is accessible inside the whole object. This is contrast to local variables, which are only accessible inside the method in which they are declared. Here is a simple field declaration:

```
class MyClass {
  var myField : Int = 0;
}
```

This declaration defines a field of type Int and initializes it to the value 0.

Scala's type inference can figure out the type of a variable, based on the value assigned to it. Therefore, you could actually omit the type in the field declaration above, like this:

```
class MyClass {
  var myField = 0;
}
```

Since 0 is by default assumed to be an Int, the Scala compiler can infer the type of the myField based on the 0 assigned to it.

Constructors

In scala constructors are declared like this:

```
class MyClass {
   var myField : Int = 0;


      def this(value : Int) = {
      this();
      this.myField = value;
      }

 }
```

This example defines a constructor which takes a single parameter, and assigns its value to the field myField.

Notice the = between the parameters and the constructor body (the {). This equals sign must be present in constructors.

Also notice the explicit call to the no-arg constructor, this();. All constructors except the no-arg constructor must always call another constructor in the beginning of its body.

Methods

In Scala methods in a class are defined like this:

```
class MyClass {
  var myField = 0;

  def getMyField() : Int = {
    return this.myField;
  }

}
```

The above example defines a method called getMyField. The return type, Int, is declared after the method name. Inside the { and } the method body is declared. The method currently just returns the myField field. Notice the = sign between the Int and {. Methods that return a value should have this equals sign there.

I will cover methods and functions in more detail in a separate text. Here I have just shown you how to declare a method inside a class.

Here is a method that doesn't return anything, but instead modifies the internal state (field) of the object. Notice how this addToMyField() method does not have the equals sign, and no return type specified.

```
    class MyClass {
      var myField = 0;

      def getMyField() : Int = {
        return this.myField;
      }

        def addToMyField(value : Int) {
        this.myField += value;
        }

    }
```

Eg:

```
class Greeter {
   def SayHi() = println("Hello world!")
}

val greeter = new Greeter()
greeter.SayHi()
```

In the above script we first declare a class named Greeter with a single method, SayHi. We then create a new variable named greeter whose SayHi method we finally invoke.

Considering that we looked at how methods are defined in the previous part there's not really anything strange or funky in the class definition. The same goes for the instantiation of the class. Just like in C# and Java we create a new instance using the new keyword. We assign it to a variable which we create using the val keyword. That's something new though. In Scala we create variables either using the val keyword or the var keyword. Using val we get a read-only variable that's immutable. In other words, once it's value is assigned it can't change. If we on the other hand create a variable using the var keyword we get a mutable variable whose value we can later change. In this case an immutable variable is fine, but if we were to reassign the variable later on

```
class Greeter {
   def SayHi() = println("Hello world!")
}

val greeter = new Greeter()
greeter.SayHi()
greeter = new Greeter()
```

then we'd get a compilation error:

```
c:\learningscala\Classes and methods>scala helloWithClass.scala
c:\learningscala\Classes and methods\helloWithClass.scala:7: error: reassignment
to val
greeter = new Greeter()


one error found
```

If we'd change the greeter variable to instead be a var, like in the snippet below, the code would compile and run just fine.

```
class Greeter {
   def SayHi() = println("Hello world!")
}

var greeter = new Greeter()
greeter.SayHi()
greeter = new Greeter()
```

Constructors in Scala

Let's say that we want to provide the message that the Greeter class writes to the console upon instantiation. Then we'd need a constructor. Constructors are a bit different in Scala than in C# or Java. In Scala the primary constructor is the class' body and it's parameter list comes right after the class name. So, to provide the message that Greeter will print using a constructor we can modify our code to look like this:

```
class Greeter(message: String) {
  def SayHi() = println(message)
}

val greeter = new Greeter("Hello world!")
greeter.SayHi()
```

To a C# or Java programmer (or at least to me) it looks like the message variable is a method parameter and it certainly is a parameter to the constructor. But, as proved by the fact that we can use it in the SayHi method, it is also a field in the class. That's actually not entirely true though. The parameter isn't a field, but the compiler will automatically create a field for us and take core of setting it's value. Anyhow, the field is by default a val (immutable) and it's access is public. That means that the above class definition in Scala is close to functionally equivalent to the below class definition in C#.

```
public class Greeter
{
  public readonly string message;

  public Greeter(string message)
  {
    this.message = message;
  }

  public void SayHi()
  {
    Console.WriteLine(message);
  }
}
```

How strange! But also a bit intriguing right? After all we accomplished what in C# took six lines (not counting lines with just curly braces) in just two lines in Scala. From a Java or C# perspective one might be a bit suspicious of that the message field was made public by default though. We'll look into why that is, and why it's sensible that it is that way, later.

As I mentioned before the class' body is the primary constructor so if we wanted to perform some operations when an instance is created we would write that code directly in the class' body.

```
class Greeter(message: String) {
   println("A greeter is being instantiated")

   def SayHi() = println(message)
}
val greeter = new Greeter("Hello world!")
greeter.SayHi()
```

In the above example we added a call to println to print a message when a Greeter is being instantiated. Running this script produces the following output:

```
A greeter is being instantiated
Hello world!
```

If we would turn the message field into a var instead of a val we could also modify it when the class is instantiated.

```
class Greeter(var message: String) {
   println("A greeter is being instantiated")

   message = "I was asked to say " + message

   def SayHi() = println(message)
}

val greeter = new Greeter("Hello world!")
greeter.SayHi()
```

The above script produces this output:

```
A greeter is being instantiated
I was asked to say Hello world!
```

Abstract Classes :

Abstract classes

Abstract classes works just like in Java or C# and we declare them using the abstract keyword. To illustrate, let's modify our previous example so that the Greeter class is abstract, having a concrete implementation of the SayHi method that prints the value of an abstract field. We then create a class, SwedishGreeter, that extends Greeter and defines the message to say "Hej världen!" which means hello world in Swedish.

Eg:

```
abstract class Greeter {
   val message: String
   def SayHi() = println(message)
}

class SwedishGreeter extends Greeter{
   message = "Hej världen!"
}
```

```
val greeter = new SwedishGreeter()
greeter.SayHi()
```

7.Scala – Access Modifiers

Members of packages, classes, or objects can be labeled with the access modifiers private and protected.A member labeled private is visible only inside the class or object that contains the member definition. In Scala, this rule applies also for inner classes. This treatment is more consistent, but differs from Java.

```
class Outer {
  class Inner {
    private def f() { println("f") }
    class InnerMost {
      f() // OK }
    }
  }
  (new Inner).f() // error: f is not accessible
}
```

Access to protected members is also a bit more restrictive than in Java. In Scala, a protected member is only accessible from subclasses of the class in which the member is defined in same package. In Java such accesses are also possible from other classes in the same package.

```
package p {
  class Super {
    protected def f() { println("f") }
  }
  class Sub extends Super {
    f()
  }
  class Other {
    (new Super).f()  // error: f is not accessible
  }
}
```

Every member not labeled private or protected is public. There is no explicit modifier for public members.

A modifier of the form private[X] or protected[X] means that access is private or protected up to X, where X designates some enclosing package, class or singleton object.

```
package bobsrockets

package navigation {
  private[bobsrockets] class Navigator {
    protected[navigation] def useStarChart() {}
    class LegOfJourney {
      private[Navigator] val distance = 100
    }
    private[this] var speed = 200
  }
```

```
}
package launch {
  import navigation._
  object Vehicle {
    private[launch] val guide = new Navigator
  }
}
```

modifier	scope
no access modifier	public access
private[bobsrockets]	access within outer package
protected[navigation]	same as package visibility in Java
private[Navigator]	same as private in Java
private[LegOfJourney]	same as private in Scala
private[this]	access only from same object

Visibility and Companion Objects

In Scala there are no static members; instead you can have a companion object that contains members that exist only once.

Scala's access rules privilege companion objects and classes when it comes to private or protected accesses.

```
class Rocket {
  import Rocket.fuel
  private def canGoHomeAgain = fuel > 20
}

object Rocket {
  private def fuel = 10
  def chooseStrategy(rocket: Rocket) {
   if (rocket.canGoHomeAgain)
     goHome()
   else
     pickAStar()
  }
  def goHome() {}
  def pickAStar() {}
}
```

By contrast, a protected member in a companion object makes no sense, as singleton objects don't have any subclasses.

8.Operators :

Scala Operators – Arithmetic, Relational, Logical, Bitwise and Assignment Operators example

An operator tells the compiler to perform specific mathematical or logical operations. The following types of operators are supported in Scala programming language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Let's go through with each of them one by one with examples.

Arithmetic Operators

Arithmetic operators include the basic operations like Addition, Subtraction, Multiplication, Division and Modulus. The arithmetic operators supported are +,-,*,/ and %.

Operator	Description
+	Adds two operands
–	Subtracts the second operand from the first
*	Multiplies two operands
/	Divides the numerator by the denominator
%	Returns the remainder

Examples for Arithmetic operators:

```
object Arithmetic {

def main(args:Array[String]) {
var x = 40;
var y = 20;

println("Addition of x + y = " + (x + y));

println("Subtraction of x - y = " + (x - y));

println("Multiplication of x * y = " + (x * y));

println("Division of x / y = " + (x / y));

println("Modulus of x % y = " + (x % y));

}
}
```

We are defining an object Arithmetic with the main method which accepts argument of String type of array. x and y variables are declared with values 40 and 20 respectively. We are printing the result by performing operations x+y, x-y, x*y, x/y, x%y.

Note that in Scala unlike in Java we can directly create an object without first creating a class.

Run the above code after creating object Arithmetic by typing Arithmetic.main(null) in Scala shell.

Output:

```
scala> Arithmetic.main(null)
Addition of x + y = 60
Subtraction of x - y = 20
Multiplication of x * y = 800
Division of x / y = 2
Modulus of x % y = 0
```

Relational Operators

Relational operators include ==,!=,>,<,>= and <=. The following table shows the list of relational operators

Operator	Description
==	Checks whether the two operands are equal or not and returns true if they are equal.
!=	Checks if the two operands are equal or not and returns true if they are not equal.
>	Checks if the first operand is greater than the second and returns true if the first operand is greater than the second operand.
<	Checks if the first operand is lesser than the second and returns true if the first operand is lesser than the second operand.
>=	Checks whether the first operand is greater than or equal to the second operand and returns true if the first operand is greater than or equal to the second operand.
<=	Checks whether the first operand is lesser than or equal to the second operand and returns true if the first operand is lesser than or equal to the second operand.

example for relational operators

```
object Relational {

def main(args:Array[String]) {
var x = 10;
var y = 20;

println("Equality of x == y is : " + (x == y));

println("Not Equals of x != y is : " + (x !=y));

println("Greater than of x > y is : " + (x > y));

println("Lesser than of x < y is : " + (x < y));

println("Greater than or Equal to of x >= y is : " + (x >= y));

println("Lesser than or Equal to of x <= y is : " + (x <= y));

}
}
```

We are defining an object Relational with the main method which accepts argument of String type of array. x and y variables are declared with values 10 and 20 respectively. We are printing the result by performing operations x ==y, x!=y, x>y, x<y, x>=y and x <=y.
Run the above code by typing Relational.main(null)

Logical Operators

Logical operators include !, || and &&. The following table shows the list of logical operators supported.

Operator	Description
!	Logical NOT operand which reverses the logical state of the operand.Returns true if the condition is satisfied
||	Logical OR operand which returns true if any of the two operands is non zero
&&	Logical AND operand which returns true if both the operand are non zero

Consider an example;

```
object Logical {

def main(args: Array[String]) {
var x = false
var y = true

println("Logical Not of !(x && y) = " + !(x && y) );

println("Logical Or of x || y = " + (x || y) );

println("Logical And of x && y = " + (x &&y) );

}
}
```

We are defining an object Logical with the main method which accepts argument of String type of array. x and y are boolean variables with values false and true respectively. We are printing the result by performing operations !(x && y), x || y and x && y.

Run the above code by typing Logical.main(null)

Output:

```
scala> Logical.main(null)
Logical Not of !(x && y) = true
Logical Or of x || y = true
Logical And of x && y = false
```

Bitwise operators

include &, |, ~, ^, <<, >> and >>>. The following table shows the bitwise operators supported in Scala.

Operator	Description
&	Binary AND operator copies the bit to the result if the operator exists in both the operands.
|	Binary OR operator copies the bit to the result if the operator exists in either of the operands.
~	Binary Ones Complement Operator is unary and has effects of the flipping bits.
^	Binary XOR operator copies if the bit is set in one of the operand but not both.
<<	Binary Left Shift operator.The left operand value is moved left by the number of bits specified in the right operand.
>>	Binary Right Shift operator.The left operand value is moved right by the number of bits specified by the right operand.
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled with zeros.

Consider an example

```
object Bitwise {
def main(args: Array[String]) {
var x = 16;
var y = 12;
var z = 0;
z = x & y;
println("Bitwise And of x & y = " + z );

z = x | y;
println("Bitwise Or of x | y = " + z );

z = x ^ y;
println("Bitwise Xor of x ^ y = " + z );

z = ~x;
println("Bitwise Ones Complement of ~x = " + z );

z = x << 2;
println("Bitwise Left Shift of x << 2 = " + z );

z = x >> 2;
println("Bitwise Right Shift of x >> 2 = " + z );

z = x >>> 2;
println("Bitwise Shift Right x >>> 2 = " + z );
}
}
```

We are defining an object Bitwise with the main method which accepts argument of String type of array. x and y are variables with values 16 and 12 respectively. We are printing the result by performing all bitwise operations.

Run the program by typing Bitwise.main(null)

Output:

```
scala> Bitwise.main(null)
Bitwise And of x & y = 0
Bitwise Or of x | y = 28
Bitwise Xor of x ^ y = 28
Bitwise Ones Complement of ~x = -17
Bitwise Left Shift of x << 2 = 64
Bitwise Right Shift of x >> 2 = 4
Bitwise Shift Right x >>> 2 = 4
```

Assignment Operators

Assignment operators include =,+=, -=, *=, /=, %=, <<=, >>=, &=, |= and ^=.The following table shows the list of assignment operators.

Operator	Description
=	Assigns value from right side operand to left side operand
+=	Adds right operand to left operand and assigns the result to left operand
-=	Subtracts right operand from the left operand and assigns the result to left operand
*=	Multiplies right operand with the left operand and assigns the result to the left operand
/=	Divides the left operand with the right operand and assigns the result to the left operand
%=	Finds the modulus of two operands and assigns the result to left operand
<<=	Left shift assignment operator.It left shifts the operand and assigns result to the left operand
>>=	Right shift assignment operator.It rights shifts the operator and assigns the value to left operand
&=	Bitwise AND assignment operator performs bitwise AND operation and assigns the result to left operand
|=	Bitwise OR assignment operator performs bitwise OR operation and assigns result to left operand
^=	Performs bitwise exclusive OR operation and assigns result to the left operand

example for assignment operators

```
object Assignment {

def main(args: Array[String]) {
var x = 20;
var y = 30;
var z = 0;

z = x + y;
println("z= x+ y = " + z );

z+= x ;
println("Add and assignment of z += x = " + z );
z -= x ;
println("Subtract and assignment of z -= x = " + z );

z *= x ;
println("Multiplication and assignment of z *= x = " + z );

x = 20;
z = 15;
```

```
z /= x ;
println("Division and assignment of z /= x = " + z );

x = 30;
z = 15;
z %= x;
println("Modulus and assignment of z %= x = " + z );

z <<= 2;
println("Left shift and assignment of z <<= 2 = " + z );

z >>= 2;
println("Right shift and assignment of z >>= 2 = " + z );

z &= x;
println("Bitwise And assignment of z &= 2 = " +z );

z ^= x ;
println("Bitwise Xor and assignment of z ^= x = " + z );

z |= x ;
println("Bitwise Or and assignment of z |= x = " + z );
}
}
```

We are defining an object Assignment with the main method which accepts argument of String type of array. x and y are variables. We are printing the result by performing all assignment operations.

Run the above example by tying Assignment.main(null)

Output:

```
scala> Assignment.main(null)
z= x+ y = 50
Add and assignment of z += x = 70
Subtract and assignment of z -= x = 50
Multiplication and assignment of z *= x = 1000
Division and assignment of z /= x = 0
Modulus and assignment of z %= x = 15
Left shift and assignment of z <<= 2 = 60
Right shift and assignment of z >>= 2 = 15
Bitwise And assignment of z &= 2 = 14
Bitwise Xor and assignment of z ^= x = 16
Bitwise Or and assignment of z |= x = 30
```

9. If Else Statement :

The Scala if command executes a certain block of code, if a certain condition is true. Here is an example:

```
var myInt : Int = 0;

if(myInt == 0) {
  println("myInt == 0");
}
```

This example would print the text "myInt == 0" to the console.

The expression inside the paranthesis must result in a boolean value (true or false). For instance, if you call a method inside the paranthesis, that method must return a boolean value.

if - else

You can add an else to an if condition, like this:

```
var myInt : Int = 1;

if(myInt == 0) {
    println("myInt == 0");
} else {
    println("myInt != 0");
}
```

Omitting { } in if - statements

Like in Java it is possible to omit the {} in an if-statement around the code to execute, if the code consists of a single line. Here is an example:

```
var myInt : Int = 1;

if(myInt == 0)
    println("myInt == 0");
 else
    println("myInt != 0");
```

if - statements as Functions

In Scala if-statements can be used as functions. That is, they can return a value. Here is an example:

```
var myInt : Int = 1;

var myText : String =
   if(myInt == 0) "myInt == 0";
   else           "myInt != 0";

println(myText);
```

Notice how the myText variable is assigned to the result of the if-statement.

The if-statement returns the last value assigned inside it. Thus, in this case, since else-clause is executed, the last value assigned is the "myInt !=0". If the if or else clause had more than one statement in them, remember that only the last assignment would be returned.

Since if-statements behave like functions, you can use them in any place you could normally use a function.

10.Scala – Loop Statements

The Scala while loop executes a certain block of code, as long as a certain condition is true. Here is an example:

```
var myInt : Int = 0;

while(myInt < 10) {
   println("myInt = " + myInt);
   myInt += 1;
}
```

This while loop would execute 10 times. For each iteration in the loop it would print the value of myInt, and then add 1 to myInt.

do while

Scala also has a do while loop. The do while loop is similar to the while loop except the condition is executed after the loop body. This means that the loop body is always executed at least once. Here is an example:

```
var myInt : Int = 0;

do {
   println("myInt = " + myInt);
   myInt+=1;
} while(myInt < 10)
```

There are situations in a program where it makes sense to always execute the loop body at least once. Thus, the do while loop comes in handy

Omitting the { } in while Loops

Like in if-statements, you can omit the { } in the while loop if the loop body consists of only a single line. Here is an example:

```
while(someObject.hasNext())
   process(someObject.next());
```

Scala for Loop:

The Scala for loop also executes a certain block of code, as long as a certain condition is true.

However, it works a bit differently than the while loop. Here is an example:

```
for(i <- 1 to 10) {
   println("i is " + i);
}
```

This for loop iterates 10 times and for each iteration assigns the val i the next number in the range.

The i <- construct is called a generator. For each iteration it initializes the val i with a value.

The 1 to 10 is called a Range type. It returns a range containing the values from 1 to 10, including both numbers. Range types are described in more detail in a different text.

The 1 to 10 is actually a method call that returns the Range type. It is equivalent to

```
(1).to(10);
```

How this works is described in more detail in the text on Scala's support for domain specific languages (DSL).

Omitting the { } in for Loops

You can omit the { } in for loops, if the body of the loop consists of a single statement. Here is an example:

```
for(i <- 1 to 10)
   println("i is " + i);
```

to vs. until

You can use either the keyword to or until when creating a Range object. The difference is, that to includes the last value in the range, whereas until leaves it out. Here are two examples:

```
for(i <- 1 to 10) {
   println("i is " + i);
}

for(i <- 1 until 10) {
   println("i is " + i);
}
```

The first loop iterates 10 times, from 1 to 10 including 10.

The second loop iterates 9 times, from 1 to 9, excluding the upper boundary value 10.

Iterating Collections and Arrays

You can iterate a collection or array using the for loop, like this:

```
var myArray : Array[String] = new Array[String](10);

for(i <- 0 until myArray.length){
   myArray(i) = "value is: " + i;
}

for(value : String <- myArray ) {
   println(value);
}
```

First an array is created.

Second, each element in the array is initialized to the text "value is: " with the index of the element appended.

Third, the array is iterated using the generator notation (<-). For each iteration the next element in the array is assigned to the val value, and then printed to the console.

Filtering for Loops

In Scala for loops it is possible to apply filtering to the iteration of a collection or array. Here is how:

```
var myArray : Array[String] = new Array[String](10);

for(i <- 0 until myArray.length){
   myArray(i) = "value is: " + i;
}

for(value : String <- myArray if value.endsWith("5")) {
   println(value);
}
```

Notice the if value.endsWith("5") marked in bold. This condition, or filter, means that the for loop only executes its body if the string assigned value ends with the text "5". Only one of the array elements ends with the text "5", so the for loop body is only executed once.

The filtering for loop above is equivalent to the following code:

```
for(value : String <- myArray) {
   if value.endsWith("5"){
      println(value);
   }
}
```

Personally, I actually prefer the good old way of writing it (the second way). While the new way is a bit shorter, it is not really much easier to read. Especially not if the conditions are more complex, as you will see hereafter.

Multiple for Loop Filters

You can apply multiple for loop filters like this:

```
for(value : String <- myArray
   if value.endsWith("5");
   if value.indexOf("value") != -1 ) {

   println(value);
}
```

Notice the two if-statements inside the for loop declaration. They are separated by a semicolon. Both of these conditions now have to be true, for the for loop body to be executed.

The for loop above is equivalent to this old school for loop:

```
for(value : String <- myArray) {
   if( value.endsWith("5") && value.indexOf("value") != -1){
      println(value);
   }
}
```

Personally, I still find the second, old school version easier to read and understand.

Nested Iteration

It is possible to do nested looping in a single for loop. Imagine you had an array of arrays. You could then iterate them like this:

```
var myArray : Array[Array[String]] = new Array[Array[String]](10);

for(i <- 0 until myArray.length){
   myArray(i) = new Array[String](10);

   for(j <- 0 until myArray(i).length){
     myArray(i)(j) = "value is: " + i + ", " + j;
   }
}

for(anArray : Array[String] <- myArray;
   aString : String       <- anArray ) {

   println(aString);
}
```

First, an array of arrays is created, and initialized with arrays, and string values.

Second, the array of arrays is iterated using a nested loop. First, each String array is in the array of arrays is assigned to the val anArray. Then each String value of each String array is assigned to the valu aString.

The above nested loop is equivalent to this old school nested for loop:

```
for(anArray : Array[String] <- myArray) {
   for(aString : String <- anArray) {
      println(aString);
   }
```

```
}
```

Again, I still prefer the old school for loop version.

Midstream Variable Binding

It is possible to bind values to a variable in the middle of a nested iteration, like this:

```
for(anArray : Array[String] <- myArray;
   aString : String       <- anArray;
   aStringUC = aString.toUpperCase()
   if aStringUC.indexOf("VALUE") != -1;
   if aStringUC.indexOf("5") != -1
   ) {

   println(aString);
}
```

The aString.toUpperCase() result is needed by two filter conditions. Rather than compute the .toUpperCase() value twice, by nesting them inside each if-statement, the uppercase version of aString is computed just once, and assigned to the variable aStringUC. The two filter-conditions (if-statements) can now refer to this "mid stream" computed value.

You could achieve the same effect using this old school for loop:

```
for(anArray : Array[String] <- myArray) {
  for(aString : String <- anArray) {

    var aStringUC = aString.toUpperCase();

    if(aStringUC.indexOf("5") != -1 && aStringUC.indexOf("VALUE") != -1){
      println(aString);
    }
  }
}
```

11.Scala – Functions :

A function is a set of statements combined together to perform a specific task. The code can be divided logically into separate functions where each function is assigned a specific task.

The function in Scala is a complete object which can be assigned to a variable whereas a method in Scala is a part of class having name, signature, bytecode and annotations. Function name can have characters like ++,+,-,– etc.

A function can be defined inside a function known as nested functions which is supported in Scala unlike in Java.

Scala Function Declarations

The syntax for declaring a function is;

def functionName

def functionName([arguments]) : return type

For example,

def multiply(a:Int ,b:Int): Int

Here def is the short form for define and multiply is the function name. a and b are parameters to the multiply function of Integer data type and the result returns an Integer data type.

Function Definition

The function definition takes the following form;

```
def functionName ([list of parameters]) : [return type] = {
function body
return [expr]
}
```

def is the keyword to define a function, functionName is the name of the function, list of parameters are the variables separated by comma and return type is the datatype of the return value. function body contains the logic and return keyword can be used along with an expression in case function returns value.

def multiply(a:Int,b:Int) : Int = {

```
var c : Int = 0
c = a*b
return c;
}
```

multiply is the name of the function with two variables a and b. We declare another variable c of integer data type, store the result of a*b in this variable and return the computed variable c.

Run the program by typing multiply(50,20) in Scala shell and you will get output like res0: Int = 1000.

A function which does not return anything can return Unit which is equivalent to void in Java. The functions which does not return anything are called procedures in scala.

```
def hello( ) : Unit = {
println("Hello, World!")
}
```

Here we are just printing hello world so the return data type is Unit.

Run the program by typing hello and you will get output as Hello, World!.

Let us see a complete example of a function by combining the declaration and definition and test the function by creating the object.

```
object mult {

def main(args:Array[String]) {
```

```
println("Multiplication result of two numbers is : "+multiply(20,21));
}

def multiply( a:Int, b:Int ) : Int = {
var c:Int = 0
c = a * b
return c
}
}
```

The line “def multiply(a:Int, b:Int) : Int” is the function declaration and “var c:Int = 0,c = a * b” constitute the function body. return c is the return value of the function. All the three declaration, definition and return type constitute a complete function.

Run the above function code by typing mult.main(null) and see the result as below.

Multiplication result of two numbers is : 420

Invoking a Function

The way of calling a function in Scala is;

functionName(list of parameters)

For example,

multiply(4,3)

We can also create an instance and invoke the function as;

[instance.]functionName(list of parameters)

test.multiply(5,6)

Consider an example of calling a function;

```
object multest {

def main(args: Array[String]) {
println( "Result is : " + multiply(12,14) );
}

def multiply( a:Int, b:Int ) : Int = {
var c:Int = 0
c = a * b
return c
}
}
```

We are creating an object multest and defining a method multiply which accepts two variables a and b of integer data type and then perform the multiplication of a and b variables storing the result of the operation in variable c. We are writing a method main and calling the function multiply(12,14) along with the parameters.

Run the above code by typing <code>multest.main(null)</code> in the scala shell and see the following output;

Result is : 168

Now let us see an example how to invoke a function by creating an instance.

```
class Multiplication {

def multiply(a : Int , b: Int ) :Int = {
var c : Int = 0
c = a * b
return c
}
}
```

Multiplication class is created and the multiply function is defined with variables a and b and return the variable c which stores the multiplication result of the two variables passed.

Create an object multest and call the method multiply creating instance of Multiplication class as;

```
object multest {

def main(args:Array[String]) {
var mul = new Multiplication()
println( "Result is : " +mul.multiply(15,16));
}
}
```

mul is the instance of the Multiplication class and the method is called as mul.multiply(15,16)

Run the code in the shell by typing multest.main(null) which produces the below output.

Result is : 240

```
scala> class Multiplication {
     |
     | def multiply(a : Int , b: Int ) :Int = {
     | var c : Int = 0
     | c = a * b
     | return c
     | }
     | }
defined class Multiplication

scala> object multest {
     |
     | def main(args:Array[String]) {
     | var mul = new Multiplication()
     | println( "Result is : " +mul.multiply(15,16));
     | }
     | }
defined object multest

scala> multest.main(null)
Result is : 240

scala>
```

Scala - Currying Functions

What this is saying is that the currying process transforms a function of two parameters into a function of one parameter which returns a function of one parameter which itself returns the result. In Scala, we can accomplish this like so:

examples:

```
def add(x:Int, y:Int) = x + y
add(1, 2)   // 3
add(7, 3)   // 10
```

And after currying:

```
def add(x:Int) = (y:Int) => x + y

add(1)(2)   // 3

add(7)(3)   // 10
```

In the first sample, the add method takes two parameters and returns the result of adding the two. The second sample redefines the add method so that it takes only a single Int as a parameter and returns a functional (closure) as a result. Our driver code then calls this functional, passing the second “parameter”. This functional computes the value and returns the final result.

We can shorten our curried definition to something more like this:

```
def add(x:Int)(y:Int) = x + y

add(1)(2)   // 3

add(7)(3)   // 10
```

Applied to Existing Methods

Currying isn’t just limited to methods which you define. Thanks to the power of a language with inner methods and closures, it is possible to define a method which takes a function of n parameters and converts it to a curried function of order n. In fact, this function is already defined within the Function singleton in the scala package. We can use it to curry the first version of our add method:

```
def add(x:Int, y:Int) = x + y
val addCurried = Function.curried(add _)
add(1, 2)   // 3
addCurried(1)(2)   // 3
```

12. Scala – Closures

A closure can be defined as the function whose return value depends on the value of one or more variable defined outside this function. A closure is an instance of a function, a value whose non-local variables have been bound either to values or to storage locations.

```
object Divide {

def main(args:Array[String]) {
println( "Divided value = " + divider(4) )
println( "Divided value = " + divider(6) )
println( "Divided value = " + divider(8) )
println( "Divided value = " + divider(10) )
}

var div = 2
val divider = (i:Int) =&gt; i/div
}
```

We are defining an object Divide with a main function that accepts arguments of String Array type and printing the value of the divider by calling a variable divider defined outside the main method where we divide the value passed to divider with the div value “2” initialized outside the main method.

i and div are the two free variables. i is the formal parameter to the function. div is not a formal parameter and has a reference to a variable outside the function but in the enclosing scope as

```
var div = 2
val divider = (i:Int) => i/div
```

Run the above example by typing Divide.main(null)

Output:

```
Divided value = 2
Divided value = 3
Divided value = 4
Divided value = 5
```

13. Scala – Strings

A String can be defined as a sequence of characters. Consider an example of defining a string variable in Scala programming.
Ex:

```
object Str {
val st: String = "Scala is a functional programming language"
def main(args:Array[String]) {
println(st)
}
}
```

Output: Scala is a functional programming language

The strings in scala are same as java string and hence the value is of type java.lang.String. Java classes are available in scala and hence the scala makes use of java strings without creating a separate string class.

Similar to Java, String is immutable in Scala i.e. the object cannot be modified.

here are some basic uses of the Scala String class to help get us warmed up:

```
val name = "Alvin"
val fullName = name + " Alexander"  // Alvin Alexander
val fullName = s"$name Alexander"   // Alvin Alexander

name.length              // 5
name.foreach(print)      // Alvin
name.take(2)             // Al
name.take(2).toLowerCase  // al
```

String equality

Test string equality with the == symbol (a method, actually):

```
val a = "foo"
val b = "foo"
val c = "bar"
a == b      // true
a == c      // false
```

Multiline strings

It's easy to create multiline strings in Scala:

```
val foo = """This is
   a multiline
   String"""

val speech = """Four score and
           |seven years ago""".stripMargin

val speech = """Four score and
           #seven years ago""".stripMargin('#')
```

```
val speech = """Four score and
             |seven years ago
             |our fathers""".stripMargin.replaceAll("\n", " ")

// from @SomSnytt an twitter, instead of replaceAll("\n", " ") you can use these:
yourStringHere.lines.mkString(" ")
yourStringHere.lines mkString " "

// use single- and double-quotes in a multiline string
val s = """This is known as a
"multiline" string
or 'heredoc' syntax"""
```

String interpolation/substitution

Regarding Scala String substitution/interpolation, there are three interpolators built in:

- s
- f
- raw

Examples of String substitution/interpolation:

```
val name = "Joe"
val age = 42
val weight = 180.5

// prints "Hello, Joe"
println(s"Hello, $name")

// prints "Joe is 42 years old, and weighs 180.5 pounds."
println(f"$name is $age years old, and weighs $weight%.1f pounds.")

// 'raw' interpolator
println(raw"foo\nbar")
```

here are some printf formatting characters:

```
%c  character
%d  decimal (integer) number (base 10)
%e  exponential floating-point number
%f  floating-point number
%i  integer (base 10)
%o  octal number (base 8)
%s  a string of characters
%u  unsigned decimal (integer) number
%x  number in hexadecimal (base 16)
%%  print a percent sign
\%  print a percent sign
```

Substrings

```
val s = "four score"
s.substring(0,3)          // "fou"
s.substring(0,4)          // "four"
s.substring(1,5)          // "our "
s.substring(1,6)          // "our s"

s.substring(0, s.length-1)   // "four scor"
s.substring(0, s.length)     // "four score"
```

Regular expressions

Can use .r on a string to create a regular expression. Can also use the Regex class:

```
// create a regex with '.r'
val numPattern = "[0-9]+".r
val address = "123 Main Street"              // "123 Main Street"
val match1 = numPattern.findFirstIn(address)    // Some(123)

// create a regex with Regex class
import scala.util.matching.Regex
val numPattern = new Regex("[0-9]+")
val address = "123 Main Street Unit 639"
val matches = numPattern.findAllIn(address)         // non-empty iterator
val matches = numPattern.findAllIn(address).toArray   // Array(123, 639)
```

“Replacing” string content:

```
val regex = "[0-9]".r
regex.replaceAllIn("123 Main Street", "x")

"123 Main Street".replaceAll("[0-9]", "x")   // "xxx Main Street"
"Hello world".replaceFirst("H", "J")        // "Jello world"
"99 Luft Balloons".replaceAll("9", "1")     // "11 Luft Balloons"
"99 Luft Balloons".replaceFirst("9", "1")    // "19 Luft Balloons"
```

```
scala> val pattern(month, day, year) = "June 22, 2012"
month: String = June
day: String = 22
year: String = 2012
```

Transforming arrays to a String

```
val a = Array(1,2,3)
a.mkString               // "123"
a.mkString(",")          // "1,2,3"
a.mkString(" ")          // "1 2 3"
a.mkString("(", ",", ")")   // "(1,2,3)"
```

How to get a character at a String position:

```
// the java way (don't do this)
```

```
"hello".charAt(0)   // "h"

// the scala way
"hello"(0)        // "h"
"hello"(1)        // "e"
```

A lot of String method examples

Uppercase, lowercase, and capitalize method examples:

```
val s = "hello world"
s.toUpperCase    // "HELLO WORLD"
s.toLowerCase    // "hello world"
s.capitalize     // "Hello world"
```

Distinct, intersect, and diff method examples:

```
// distinct
"hello world".distinct   // yields "helo wrd"

// intersect
val a = "Alvin"
val b = "Alexander"
a intersect b          // yields "Aln"
b intersect a          // yields "Aln"

// diff
val a = "Four score and six years ago"
val b = "Four score and seven years ago"
a diff b               // "ix"
b diff a               // "vene"
```

A String can be treated as a collection of Char, and that's what is shown in many of the following examples. I've skipped over a few of the collection-related methods because (a) they're a little harder to show, and (b) I'm running out of time. Here are the String examples:

```
foo * 3                  // foofoofoo

a.capitalize               // Foo bar baz
a.collect
a.compare
a.compareTo
a.count(_ == 'a')           // 2
a.diff("foo")              // " bar baz"
a.distinct                // fo barz
a.drop(4)                  // bar baz
a.dropRight(2)              // foo bar b
a.dropWhile(_ != ' ')        // " bar baz"
a.endsWith("baz")            // true
a.filter(_ != 'a')          // foo br bz
a.fold                    //
a.foreach(println(_))        // prints one character per line
a.foreach(println)           // prints one character per line
a.format                  //
```

```
a.getBytes.foreach(println)    // prints the byte value of each character, one value per line
a.head                  // f
a.headOption                // Some(f)
a.indexOf('a')              // 5
a.intersect                 // shown above
a.isEmpty                   // false
a.lastIndexOf('o')          // 2
a.length                    // 11
a.map(_.toUpper)            // FOO BAR BAZ
a.map(_.byteValue)          // Vector(102, 111, 111, 32, 98, 97, 114, 32, 98, 97, 122)
a.min                   // " "
a.mkString(",")             // f,o,o, ,b,a,r, ,b,a,z
a.mkString("->", ",", "<-")    // ->f,o,o, ,b,a,r, ,b,a,z<-
a.nonEmpty                  // true
a.par                   // a parallel array, ParArray(f, o, o,  , b, a, r,  , b, a, z)
a.partition(_ > 'e')        // (foorz, " ba ba")  // a Tuple2
a.reduce                    //
a.replace('o', 'x')         // fxx bar baz
a.replace("o", "x")         // fxx bar baz
a.replaceAll("o", "x")      // fxx bar baz
a.replaceFirst("o", "x")    // fxo bar baz
a.reverse                   // zab rab oof
a.size                  // 11
a.slice(0,5)                // foo b
a.slice(2,9)                // o bar b
a.sortBy                    //
a.sortWith(_ < _)           // "  aabbfoorz"
a.sortWith(_ > _)           // "zroofbbaa  "
a.sorted                    // "  aabbfoorz"
a.span                  //
a.split(" ")                // Array(foo, bar, baz)
a.splitAt(3)                // (foo," bar baz")
a.substring(4,9)            // bar b
a.tail                  // oo bar baz
a.take(3)                   // foo
a.takeRight(3)              // baz
a.takeWhile(_ != 'r')       // foo ba
a.toArray                   // Array(f, o, o,  , b, a, r,  , b, a, z)
a.toBuffer                  // ArrayBuffer(f, o, o,  , b, a, r,  , b, a, z)
a.toList                    // List(f, o, o,  , b, a, r,  , b, a, z)
a.toSet                     // Set(f, a,  , b, r, o, z)
a.toStream                  //
a.toLowerCase               // foo bar baz
a.toUpperCase               // FOO BAR BAZ
a.toVector                  // Vector(f, o, o,  , b, a, r,  , b, a, z)
a.trim                  // "foo bar baz"
a.view                  //
a.zip(0 to 10)              // Vector((f,10), (o,11), (o,12), ( ,13), (b,14), (a,15), (r,16), ( ,17), (b,18), (a,19),
(z,20))
a.zipWithIndex              // Vector((f,0), (o,1), (o,2), ( ,3), (b,4), (a,5), (r,6), ( ,7), (b,8), (a,9), (z,10))
```

14.Arrays

In Scala arrays are immutable objects. You create an array like this:

```
var myArray : Array[String] = new Array[String](10);
```

First you declare variable var myArray to be of type Array[String]. That is a String array. If you had needed an array of e.g. Int, replace String with Int.

Second you create a new array of Strings with space for 10 elements (10 Strings). This is done using the code new Array[String](10). The number passed in the parentheses is the length of the array. In other words, how many elements the array can contain.

Once created, you cannot change the length of an array.

Accessing Array Elements

You access the elements of an array by using the elements index in the array. Element indexes go from 0 to the length of the array minus 1. So, if an array has 10 elements, you can access these 10 elements using index 0 to 9.

You can access an element in an array like this:

```
var aString : String = myArray(0);
```

Notice how you use normal parentheses to enclose the element index, rather than the square brackets [] used in Java.

To assign a value to an array element, you write this:

```
myArray(0) = "some value";
```

Iterating Array Elements

You can iterate the elements of an array in two ways. You can iterate the element indexes, or iterate the elements themselves.

Iterate Indexes

The first way is to use a for loop, and iterate through the index numbers from 0 until the length of the array. Here is how:

```
for(i <- 0 until myArray.length){
   println("i is: " + i);
   println("i'th element is: " + myArray(i));
}
```

The until keyword makes sure to only iterate until myArray.length - 1. Since array element indexes go from 0 to the array length - 1, this is the appropriate way to iterate the array.

If you had needed i to also take the value of myArray.length in the final iteration, you could have used the to keyword instead of the until keyword. For more details on for loops, see the text Scala for.

Iterate Elements

The second way to iterate an array is to ignore the indexes, and just iterate the elements themselves. Here is how:

```
for(myString <- myArray) {
   println(myString);
}
```

For each iteration of the loop the myString takes the value of the next element in the array. Note that myString is a val type, so you cannot reassign it during the for loop body execution.

Scala Arrays Example

Scala supports the array data structure. An array is a fixed size data structure that stores elements of the same data type. The index of first element of an array is zero and the last element is the total number of elements minus one.

Scala Array Declaration

The syntax for declaring an array variable is

var arrayname = new Array[datatype](size)

var indicates variable and arrayname is the name of the array, new is the keyword, datatype indicates the type of data such as integer, string and size is the number of elements in an array.

Ex1:

var student = new Array[String](5)

Or

var student:Array[String] = new Array[String](5)

Here student is a string array that holds five elements. Values can be assigned to an array as below.

var student = Array("John","Adam","Rob","Reena","Harry")
or
student(0) = "Martin" ; student(1) = "Jack" ; student(2) = "Jill" ; student(3) = "Paul" ; student(4) = "Kevin"

Scala Arrays Processing

While processing arrays we use for loop as the elements are of same data type.

```
object Student {
def main(args: Array[String]) {
var marks = Array(75,80,92,76,54)

println("Array elements are : ")
for ( m1 <- marks ) {
println(m1 )
}

var gtot = 0.0
for ( a <- 0 to (marks.length - 1)) {
gtot += marks(a);
          }
          println("Grand Total : " + gtot);

var average = 0.0
average = gtot/5;
println("Average : " + average);
}
}
```

Here we are creating marks array of Integer data type. We are printing the elements of an array using for loop. We are calculating the total marks by adding all the elements and calculate average by dividing the total by the number of subjects.

Scala Multi Dimensional Arrays

Multidimensional arrays can be defined as an Array whose elements are arrays.

Consider an example below.

```
object Multidim {
def main(args:Array[String]) {
val rows = 2
val cols = 3
val multidim = Array.ofDim[String](rows,cols)
multidim(0)(0) = "Reena"
multidim(0)(1) = "John"
multidim(0)(2) = "Adam"
multidim(1)(0) = "Michael"

multidim(1)(1) = "Smith"
multidim(1)(2) = "Steve"
for {
i <- 0 until rows
j <- 0 until cols
}println(s"($i)($j) = ${multidim(i)(j)}")
}
}
```

We are creating a two dimensional array with 2 rows and 3 columns using the ofDim method which accepts rows and columns as arguments. Add the elements to the array as rows and columns accepting string arguments. We are using for loop retrieve the inserted elements.

Scala Concatenate Arrays

Two arrays can be concatenated using the concat method. Array names can be passed as an argument to the concat method.

Below is an example showing how to concatenate two arrays.

```
import Array._

object Student {
def main(args: Array[String]) {
var sname = Array("John","Adam","Rob","Reena","Harry")
var sname1 = Array("Jack","Jill","Henry","Mary","Rohan")

var names = concat( sname, sname1)
println("Student name array elements are : ");
for ( n1 <- names ) {
println( n1 )
}
}
}
```

We have to use import Array._ since the array methods concat is defined in the package. We have declared two arrays sname and sname1 having student names as the elements. We are concatenating these two arrays using concat method and passing sname and sname1 as arguments and storing the resultant elements in names array.

Scala Array with range

The range() method generates an array containing sequence of integers. The final argument is used as a step to generate the sequence of integers. If the argument is not specified then the default value assumed is

Let us understand this range method through an example.

```
import Array._

object Student {

def main(args: Array[String]) {
var id = range(7, 23, 3)
var age = range(15,20)
for ( s <- id ) {
print( " " + s)
}
println()
for ( a <- age ) {
print( " " + a )
}
}
}
```

We are declaring arrays id and age and generating the elements using range method. The elements start from 7 to 23 incrementing by 3. For the age the increment value by 1 starting from 15 until 20.

OutPut:

7 10 13 16 19 22
15 16 17 18 19

some useful Array methods:

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

1. def empty[T]: Array[T]: Returns an array of length 0
2. def ofDim[T](n1: Int, n2: Int): Array[Array[T]]: Creates a 2-dimensional array
3. def range(start: Int, end: Int, step: Int): Array[Int]: Returns an array containing equally spaced values in some integer interval
4. def copy(src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int): Unit: Copy one array to another
5. def ofDim[T](n1: Int, n2: Int, n3: Int): Array[Array[Array[T]]]: Creates a 3-dimensional array

15.Scala – Collections:

Collections are the container of things which contains random number of elements. All collection classes are found in the package scala.collection. Collections are of two types –

- Mutable Collections
- Immutable Collections

Mutable Collection – This type of collection is changed after it is created. All Mutable collection classes are found in the package scala.collection.mutable.

Immutable Collection – This type of collection will never change after it is created. All Immutable collection classes are found in the packagescala.collection.immutable.

Most Common Collection types are –

1. List
2. Map
3. Set
4. Tuple
5. Iterators

1. List –

List is a collection of similar types of elements which are immutable. It represents the Linked list.

If list contains t tpes of elements then it can be represented as –

Lists preserve order, can contain duplicates, and are immutable.

List is represented as List[T] where T is the data-type of the elements.

Ex:

val StudentNames[String] = List("Rohan", "Andreas", "Rob", "John")

Two dimensional list can be created as

val twodim: List[List[Int]] =

List(
List(1, 0, 0),
List(0, 1, 0),
List(0, 0, 1)
)

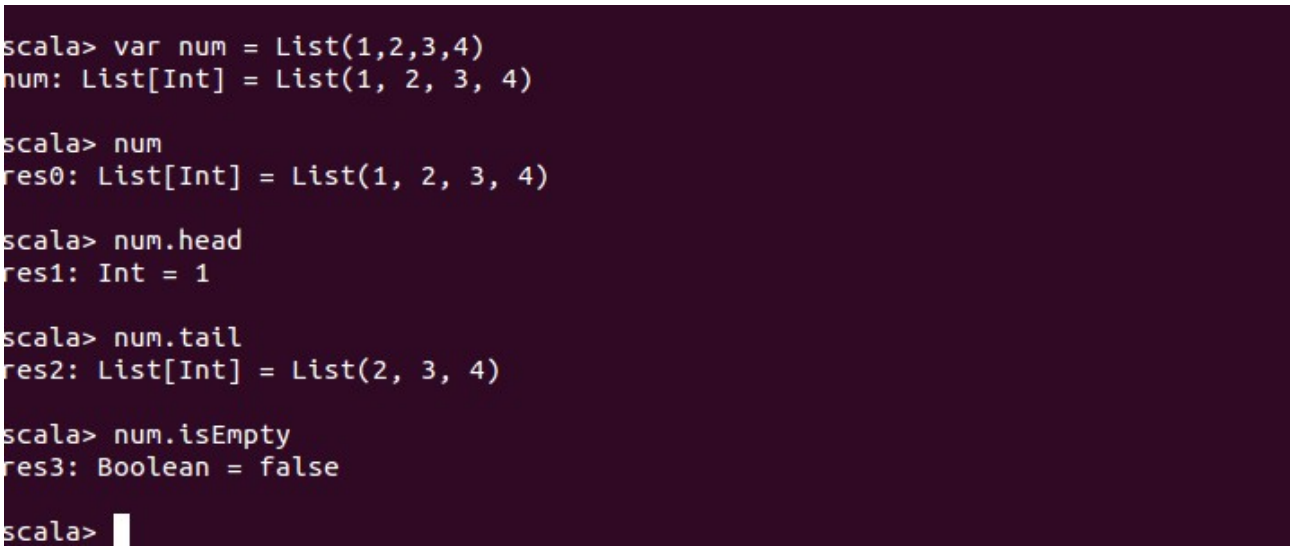
example :

scala> val nums = List(5, 1, 4, 3, 2)
nums: List[Int] = List(5, 1, 4, 3, 2)

Basic operations on lists –

Methods	Description
head	It returns the first element of a list.
tail	It returns a list consisting of all elements except the first.
isEmpty	It returns true if the list is empty.

Example for Basic operations :



Scala List class examples

The Scala List class may be the most commonly used data structure in Scala applications. Therefore, it's very helpful to know how create lists, merge lists, select items from lists, operate on each element in a list, and so on.

The List range method

You can also create a List with the List's range method:

```
scala> val x = List.range(1,10)
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

The range function can also take a third argument which serves as a "step" value when creating the List:

```
scala> val x = List.range(0,10,2)
x: List[Int] = List(0, 2, 4, 6, 8)
```

The List fill method

You can also create a new List with the fill method:

```
scala> val x = List.fill(3)("foo")
x: List[java.lang.String] = List(foo, foo, foo)
```

The List tabulate method

Finally, you can create a new List with the tabulate method of the List class. The tabulate method creates a new List whose elements are created according to the function you supply. The book Programming in Scala shows how to create a List using a simple "squares" function:

```
scala> val x = List.tabulate(5)(n => n * n)
x: List[Int] = List(0, 1, 4, 9, 16)
```

Prepend items to a List

You can prepend items to a Scala List using the :: method:

```
// create a List
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

// prepend an element to the list
scala> val y = 0 :: x
y: List[Int] = List(0, 1, 2, 3)
```

Appending and merging Lists

There are at least two ways to merge Scala Lists. First, you can merge two Scala lists using the ::: method of the List class:

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

scala> val b = List(4,5,6)
b: List[Int] = List(4, 5, 6)

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

You can also merge two Scala lists using the List's concat method:

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

```
scala> val b = List(4,5,6)
b: List[Int] = List(4, 5, 6)

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

Iterating lists with foreach

A very common way to iterate over Scala lists is with the foreach method.

Here's a simple example showing how to use the foreach function to print every item in a List:

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

scala> x.foreach { println }
1
2
3
```

This next example shows a way to sum all the elements in a list using the foreach method:

```
scala> var sum = 0
sum: Int = 0

scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x.foreach(sum += _)

scala> println(sum)
6
```

Scala Lists and the for comprehension

The Scala for comprehension is not specific to lists, but is an extremely powerful way to operate on lists. Here's a simple example of how to iterate over a list using the for comprehension:

```
scala> val names = List("Bob", "Fred", "Joe", "Julia", "Kim")
names: List[java.lang.String] = List(Bob, Fred, Joe, Julia, Kim)

scala> for (name <- names) println(name)
Bob
Fred
Joe
Julia
Kim
```

So far, so good. Now let's add a simple "if" clause to the for comprehension to print only the elements we want to print:

```
scala> val names = List("Bob", "Fred", "Joe", "Julia", "Kim")
names: List[java.lang.String] = List(Bob, Fred, Joe, Julia, Kim)

scala> for (name <- names if name.startsWith("J"))
     | println(name)
```

Joe
Julia

Filtering Scala lists

A great thing about Scala is that it is a functional programming language. In the next examples we'll show some of the power of functional programming. In this section we'll focus on the filter method of the List class.

```
scala> val x = List(1,2,3,4,5,6,7,8,9,10)
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// create a list of all the even numbers in the list
scala> val evens = x.filter(a => a % 2 == 0)
evens: List[Int] = List(2, 4, 6, 8, 10)
```

takeWhile

```
scala> val x = List(1,2,3,4,5,6,7,8,9,10)
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val y = x.takeWhile(a => a < 6)
y: List[Int] = List(1, 2, 3, 4, 5)
```

The List map function

The Scala List map function "transforms each element of a collection based on a function."

Here are a few map examples. First, doubling every item in a List:

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

scala> val y = x.map(a => a * 2)
y: List[Int] = List(2, 4, 6)
```

Here's a slightly simpler version of that map example, using the Scala wildcard character (_) instead of a fake variable name:

```
scala> val y = x.map(_ * 2)
y: List[Int] = List(2, 4, 6)
```

Here's an example using a list of strings:

```
scala> val names = List("Fred", "Joe", "Bob")
names: List[java.lang.String] = List(Fred, Joe, Bob)

scala> val lower = names.map(_.toLowerCase)
lower: List[java.lang.String] = List(fred, joe, bob)

scala> val upper = names.map(_.toUpperCase)
upper: List[java.lang.String] = List(FRED, JOE, BOB)
```

A very nice example in the book Beginning Scala demonstrates how to convert a List into something useful in the HTML world, a list of <li> elements using the map function:

```
scala> val names = List("Fred", "Joe", "Bob")
names: List[java.lang.String] = List(Fred, Joe, Bob)
```

```
scala> val li = names.map(name => <li>{name}</li>)
li: List[scala.xml.Elem] = List(<li>Fred</li>, <li>Joe</li>, <li>Bob</li>)
```

As you can see, you can rapidly build a lot of functionality in just a little bit of code.

Sorting Scala Lists

I need to research why this code is deprecated, but for the time being, here's an example of how to sort a Scala List:

```
scala> val x = List(10, 2, 5)
x: List[Int] = List(10, 2, 5)

scala> val y = x.sort(_ < _)
warning: there were 1 deprecation warnings; re-run with -deprecation for details
y: List[Int] = List(2, 5, 10)
```

Scala List class and pattern matching

You can use the List class with the Scala pattern matching and case/match syntax. I'll add examples here as I create them in my own code.

Other Scala List functions

The Scala List class has an incredible number of functions/methods, and over time I'll attempt to document them all. In the meantime, here's a short list of the many other Scala List methods I don't have examples for at this time:

```
length  - returns the length of a List
head    - returns the first element of a List
last    - returns the last element of a List
init    - returns a List consisting of all elements except the last one
tail    - returns every elements of a List except the first element
isEmpty - returns a Boolean indicating if the List is empty
```

reverse - returns a reversed version of the List

flatten - takes a list of lists and flattens it out to a single list

mkString - converts a List to a String

Scala Collection maps:

Scala map is a collection 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.

The immutable Map class is in scope by default, so you can create an immutable map without an import, like this:

```
val states = Map("AL" -> "Alabama", "AK" -> "Alaska")
```

To create a mutable Map, import it first:

```
var states = scala.collection.mutable.Map("AL" -> "Alabama")
```

Basic Operations on Map:

All operations on maps can be expressed in terms of the following three methods:

Methods	Description
keys	It returns an iterable containing each key in the map.
values	It returns an iterable containing each value in the map.
isEmpty	It returns true if the map is empty otherwise false.

Examples:

Adding, removing, and updating mutable Map elements

The following examples show how to add, remove, and update elements in a mutable Scala Map:

```
// create an empty map

var states = scala.collection.mutable.Map[String, String]()

// create a map with initial elements
var states = scala.collection.mutable.Map("AL" -> "Alabama", "AK" -> "Alaska")

// add elements with +=
states += ("AZ" -> "Arizona")
states += ("CO" -> "Colorado", "KY" -> "Kentucky")

// remove elements with -=
states -= "KY"
states -= ("AZ", "CO")

// update elements by reassigning them
states("AK") = "Alaska, The Big State"
```

Iterating over Scala maps

Once you have a Map, you can iterate over it using several different techniques. I prefer using the for loop (or for comprehension):

```
scala> val m1 = Map("fname" -> "Al", "lname" -> "Alexander")

scala> for ((k,v) <- m1) printf("key: %s, value: %s\n", k, v)
key: fname, value: Al
key: lname, value: Alexander
```

Problem

You want to add, update, or delete elements when working with an immutable map.

Solution

Use the correct operator for each purpose, remembering to assign the results to a new map.

To be clear about the approach, the following examples use an immutable map with a series of val variables. First, create an immutable map as a val:

```
scala> val a = Map("AL" -> "Alabama")
a: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama)
```

Add one or more elements with the + method, assigning the result to a new Map variable during the process:

```
// add one element
scala> val b = a + ("AK" -> "Alaska")
b: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama, AK -> Alaska)

// add multiple elements
scala> val c = b + ("AR" -> "Arkansas", "AZ" -> "Arizona")
c: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama, AK -> Alaska, AR ->
Arkansas, AZ -> Arizona)
```

To update a key/value pair with an immutable map, reassign the key and value while using the + method, and the new values replace the old:

```
scala> val d = c + ("AR" -> "banana")
d: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama, AK -> Alaska, AR ->
banana, AZ -> Arizona)
```

To remove one element, use the - method:

```
scala> val e = d – "AR"

e: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama, AK →

 Alaska, AZ -> Arizona)
```

To remove multiple elements, use the - or -- methods:

```
scala> val f = e - "AZ" – "AL"

f: scala.collection.immutable.Map[String,String] = Map(AK -> Alaska)
```

Problem

You want to add, remove, or update elements in a mutable map.

Solution

Add elements to a mutable map by simply assigning them, or with the += method. Remove elements with -= or --=. Update elements by reassigning them.

Given a new, mutable Map:

```
scala> var states = scala.collection.mutable.Map[String, String]()
states: scala.collection.mutable.Map[String,String] = Map()
```

You can add an element to a map by assigning a key to a value:

```
scala> states("AK") = "Alaska"
```
You can also add elements with the += method:

```
scala> states += ("AL" -> "Alabama")
res0: scala.collection.mutable.Map[String,String] = Map(AL -> Alabama, AK -> Alaska)
```

Add multiple elements at one time with +=:

```
scala> states += ("AR" -> "Arkansas", "AZ" -> "Arizona")
res1: scala.collection.mutable.Map[String,String] = Map(AL -> Alabama, AR -> Arkansas, AK ->
Alaska, AZ -> Arizona)
```

Add multiple elements from another collection using ++=:

```
scala> states ++= List("CA" -> "California", "CO" -> "Colorado")
res2: scala.collection.mutable.Map[String,String] = Map(CO -> Colorado,
  AZ -> Arizona, AL -> Alabama, CA -> California, AR -> Arkansas,
  AK -> Alaska)
```

Remove a single element from a map by specifying its key with the -= method:

```
scala> states -= "AR"
res3: scala.collection.mutable.Map[String,String] = Map(AL -> Alabama, AK -> Alaska, AZ ->
Arizona)
```

Remove multiple elements by key with the -= or --= methods:

```
scala> states -= ("AL", "AZ")
res4: scala.collection.mutable.Map[String,String] = Map(AK -> Alaska)

// remove multiple with a List of keys
scala> states --= List("AL", "AZ")
res5: scala.collection.mutable.Map[String,String] = Map(AK -> Alaska)
```

Update elements by reassigning their key to a new value:

```
scala> states("AK") = "Alaska, A Really Big State"

scala> states
res6: scala.collection.mutable.Map[String,String] = Map(AK -> Alaska, A Really Big State)
```

There are other ways to add elements to maps, but these examples show the most common uses.

```
scala> val states = collection.mutable.Map(
    |   "AK" -> "Alaska",
    |   "IL" -> "Illinois",
    |   "KY" -> "Kentucky"
    | )
states: collection.mutable.Map[String,String] = Map(KY -> Kentucky, IL -> Illinois, AK -> Alaska)

scala> states.put("CO", "Colorado")
res0: Option[String] = None
```

```
scala> states.retain((k,v) => k == "AK")
res1: states.type = Map(AK -> Alaska)

scala> states.remove("AK")
res2: Option[String] = Some(Alaska)

scala> states
res3: scala.collection.mutable.Map[String,String] = Map()

scala> states.clear

scala> states
res4: scala.collection.mutable.Map[String,String] = Map()
```

As shown, the remove method returns an Option that contains the value that was removed.

How to process a Scala String one character at a time (with map, for, and foreach)

How can I iterate through each character in a Scala String, performing an operation on each character as I traverse the string?

Solution

Depending on your needs and preferences, you can use the Scala map or foreach methods, a for loop, or other approaches.

The map method

Here's a simple example of how to create an uppercase string from an input string, using the map method that's available on all Scala sequential collections:

```
scala> val upper = "hello, world".map(c => c.toUpper)
upper: String = HELLO, WORLD
```

As you see in many examples in the Scala Cookbook, you can shorten that code using the magic of Scala's underscore character:

```
scala> val upper = "hello, world".map(_.toUpper)
upper: String = HELLO, WORLD
```

With any Scala collection — such as a sequence of characters in a string — you can also chain collection methods together to achieve a desired result. In the following example, the filter method is called on the original String to create a new String with all occurrences of the lowercase letter "L" removed. That String is then used as input to the map method to convert the remaining characters to uppercase:

```
scala> val upper = "hello, world".filter(_ != 'l').map(_.toUpper)
upper: String = HEO, WORD
```

The for loop

When you first start with Scala, you may not be comfortable with the map method, in which case you can use Scala's for loop to achieve the same result. This example shows another way to print each character:

```
scala> for (c <- "hello") println(c)
h
e
l
l
o
```

To write a for loop to work like a map method, add a yield statement to the end of the loop. This for/yield loop is equivalent to the first two map examples:

```
scala> val upper = for (c <- "hello, world") yield c.toUpper
upper: String = HELLO, WORLD
```

Adding yield to a for loop essentially places the result from each loop iteration into a temporary holding area. When the loop completes, all of the elements in the holding area are returned as a single collection.

This for/yield loop achieves the same result as the third map example:

```
val result = for {
   c <- "hello, world"
   if c != 'l'
} yield c.toUpper
```

The foreach method

Whereas the map or for/yield approaches are used to transform one collection into another, the foreach method is typically used to operate on each element without returning a result. This is useful for situations like printing:

```
scala> "hello".foreach(println)
h
e
l
l
o
```

Scala Collection Sets:

Set – It is a collection of elements which are of same type but does not contain same elements. By default scala uses immutable set. Scala provides mutable and immutable versions of Set. If you want to use immutable set then use Set and if you want to use mutable set the use mutable.Set.

Sets example:

Sets do not preserve order and have no duplicates

```
scala> val numbers = Set(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

Set. Two birds sit on a branch. But in a set, only one "bird" string will exist. A set allows only distinct elements. It eliminates duplicates.
With special operators, like "++," we combine sets. In the two sets, any duplicate elements will be lost. Scala provides the Set type with simple syntax.
First example. This example uses a Set. It adds two elements to the set in the first line. With val we specify that our "animals" variable cannot be reassigned to something else.

Immutable:The set is immutable. To modify a set in Scala we can only create a new set, such as with a special operator.

Contains:This method returns true or false. It tells us whether the argument exists within the set.

Based on: Scala 2.11

Scala program that uses set, contains

```
// Create a Set of two strings.
val animals = Set("bird", "fish")
println(animals)

// See if this string is in the set.
if (animals.contains("fish"))
   println(true)

// This string is not contained in the set.
println(animals.contains("apple"))
```

Output

```
Set(bird, fish)
true
false
```

Combine set. Sets are immutable, so we cannot add or remove single elements of the existing collection. Instead we must create new sets with operators or methods.

Here:We use the "++" operator to combine two sets. Both sets have 15 as an element, and the resulting set has one instance of this value.

Scala program that combines two sets

```
// Create two sets.
val results1 = Set(10, 11, 15)
val results2 = Set(2, 3, 15)

// Combine the sets.
// ... This eliminates duplicate elements.
//     Ordering of elements is not retained.
val results3 = results1 ++ results2

// Display all sets.
println(results1)
println(results2)
println(results3)
```

Output

```
Set(10, 11, 15)
Set(2, 3, 15)
Set(10, 2, 3, 11, 15)
```

Intersect. The intersection of two sets is the common elements of both sets. We compute an intersection with intersect() or an ampersand. Both map to the same function.

Logic:Set theory is an important part of mathematics. But for programs, it usually just makes some operations easier.

Scala program that finds intersection

```
val codes1 = Set(20, 21, 30)
val codes2 = Set(40, 20, 30)

// Use intersect to find common elements of two sets.
val result1 = codes1.intersect(codes2)
println(result1)

// Short syntax for intersection.
val result2 = codes1 & codes2
println(result2)
```

Output

```
Set(20, 30)
Set(20, 30)
```

How to add elements to a Set in Scala (operators, methods)

Problem

You want to add elements to a mutable set, or create a new set by adding elements to an immutable set.

Solution

Mutable and immutable sets are handled differently, as demonstrated in the following examples.

Mutable set

Add elements to a mutable Set with the +=, ++=, and add methods:

```
// use var with mutable
scala> var set = scala.collection.mutable.Set[Int]()
set: scala.collection.mutable.Set[Int] = Set()

// add one element
scala> set += 1
res0: scala.collection.mutable.Set[Int] = Set(1)

// add multiple elements
scala> set += (2, 3)
res1: scala.collection.mutable.Set[Int] = Set(2, 1, 3)

// notice that there is no error when you add a duplicate element
scala> set += 2
res2: scala.collection.mutable.Set[Int] = Set(2, 6, 1, 4, 3, 5)

// add elements from any sequence (any TraversableOnce)
scala> set ++= Vector(4, 5)
res3: scala.collection.mutable.Set[Int] = Set(2, 1, 4, 3, 5)

scala> set.add(6)
res4: Boolean = true

scala> set.add(5)
res5: Boolean = false
```

The last two examples demonstrate a unique characteristic of the add method on a set: It returns true or false depending on whether or not the element was added to the set. The other methods silently fail if you attempt to add an element that's already in the set.

You can test to see whether a set contains an element before adding it:

```
set.contains(5)
```

But as a practical matter, I use += and ++=, and ignore whether the element was already in the set.

Whereas the first example demonstrated how to create an empty set, you can also add elements to a mutable set when you declare it, just like other collections:

```
scala> var set = scala.collection.mutable.Set(1, 2, 3)
set: scala.collection.mutable.Set[Int] = Set(2, 1, 3)
```

Immutable set

The following examples show how to create a new immutable set by adding elements to an existing immutable set.

First, create an immutable set:

```
scala> val s1 = Set(1, 2)
s1: scala.collection.immutable.Set[Int] = Set(1, 2)
```

Create a new set by adding elements to a previous set with the + and ++ methods:

```
// add one element
scala> val s2 = s1 + 3
s2: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

// add multiple elements (+ method has a varargs field)
scala> val s3 = s2 + (4, 5)
s3: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)

// add elements from another sequence
scala> val s4 = s3 ++ List(6, 7)
s4: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 7, 3, 4)
```

I showed these examples with immutable variables just to be clear about how the approach works. You can also declare your variable as a var, and reassign the resulting set back to the same variable:

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

scala> set += 4

scala> set
res0: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
```

Scala Tuples:

A Scala tuple is a class that can contain a miscellaneous collection of elements. I like to think of them as a little bag or container you can use to hold things and pass them around.
You create a tuple with the following syntax, enclosing its elements in parentheses. Here's a tuple that contains an Int and a String:

```
val stuff = (42, "fish")
```

This creates a specific instance of a tuple called a Tuple2, which we can demonstrate in the REPL:

```
scala> val stuff = (42, "fish")
stuff: (Int, java.lang.String) = (42,fish)

scala> stuff.getClass
res0: java.lang.Class[_ <: (Int, java.lang.String)] = class scala.Tuple2
```

A tuple isn't actually a collection; it's a series of classes named Tuple2, Tuple3, etc., through Tuple22. You don't have to worry about that detail, other than knowing that you can have anywhere from two to twenty-two items in a tuple. (And in my opinion, if you have twenty-two

miscellaneous items in a bag, you should probably re-think your design.)

Accessing tuple elements

You can access tuple elements using an underscore syntax. The first element is accessed with _1, the second element with _2, and so on, like this:

```
scala> val things = ("a", 1, 3.5)
things: (java.lang.String, Int, Double) = (a,1,3.5)

scala> println(things._1)
a

scala> println(things._2)
1

scala> println(things._3)
3.5
```

Use variable names to access tuple elements

When referring to a Scala tuple you can also assign names to the elements in the tuple. I like to do this when returning miscellaneous elements from a method. To demonstrate the syntax, let's create a very simple method that returns a tuple:

```
def getUserInfo = ("Al", 42, 200.0)
```

Now we can call that method, and assign the tuple results directly to variables, like this:

```
val(name, age, weight) = getUserInfo
```

Here's what this looks like in the REPL:

```
scala> def getUserInfo = ("Al", 42, 200.0)
getUserInfo: (java.lang.String, Int, Double)

scala> val(name, age, weight) = getUserInfo
name: java.lang.String = Al
age: Int = 42
weight: Double = 200.0
```

It's shown in the REPL results, but we'll further confirm that we can indeed access the values by variable name:

```
scala> name
res4: java.lang.String = Al

scala> age
res5: Int = 42

scala> weight
res6: Double = 200.0
```

That's pretty nice.

In a cool, related feature, if you only want to access some of the elements, you can ignore the others by using an underscore placeholder for the elements you want to ignore. Imagine you want to ignore the weight in our example:

```
scala> val(name, age, _) = getUserInfo
name: java.lang.String = Al
age: Int = 42
```

Or suppose you want to ignore the age and weight:

```
scala> val(name, _, _) = getUserInfo
name: java.lang.String = Al
```

Again, that's good stuff.

Iterating over a Scala tuple

As mentioned, a tuple is not a collection; it doesn't descend from any of the collection traits or classes. However, you can treat it a little bit like a collection by using its productIterator method.

Here's how you can iterate over the elements in a tuple:

```
scala> val t = ("Al", 42, 200.0)
t: (java.lang.String, Int, Double) = (Al,42,200.0)

scala> t.productIterator.foreach(println)
Al
42
200.0
```

The tuple toString method

The tuple toString method gives you a nice representation of a tuple:

```
scala> t.toString
res9: java.lang.String = (Al,42,200.0)

scala> println(t.toString)
(Al,42,200.0)
```

Creating a tuple with ->

In another cool feature, you can create a tuple using this syntax:

```
1 -> "a"
```

This creates a Tuple2, which we can demonstrate in the REPL:

```
scala> 1 -> "a"
res1: (Int, java.lang.String) = (1,a)

scala> res11.getClass
res2: java.lang.Class[_ <: (Int, java.lang.String)] = class scala.Tuple2
```

You'll see this syntax a lot when creating maps:

```
scala> val map = Map(1->"a", 2->"b")
map: scala.collection.immutable.Map[Int,java.lang.String] = Map(1 -> a, 2 -> b)
```

Scala Iterators :

It is used to access the elements of the collection one by one.Two important operations of iterator is

- next

- hasNext

next – It is used to return the next element of the iterator and move ahead the state of the iterator.

hasNext – It is used to find out whether there are more elements to return.

Iterator.scala

```
object Car {
def main(args: Array[String]) {
val car = Iterator("Santro", "Punto", "WagonR", "Polo", "Audi")
while (car.hasNext) {
println(car.next())
}
}
}
```

Output:

```
Santro
Punto
WagonR
Polo
Audi
```

Length of Iterator

The size or length methods can be used to find the number of the elements in the iterator.

```
object CarLength {
def main(args: Array[String]) {
val c1 = Iterator("Santro", "Punto", "WagonR", "Polo", "Audi")
val c2 = Iterator("Volkswagen", "Alto", "Xylo", "Innova")
println("Iterator c1 : " + c1.size)
println("Length of c2 : " + c2.length)
}
}
```

Output:

```
Iterator c1 : 5
Length of c2 : 4
```

Finding Minimum and Maximum Elements

The it.min and it.max methods are used to find the minimum and maximum elements in the iterator.

Create scala object MinMax.scala as;

```
object MinMax {
def main(args: Array[String]) {
val m1 = Iterator(12,45,67,89)
val m2 = Iterator(44,66,77,88)
println("Smallest element " + m1.min )
```

```
println("Largest element " + m2.max )
}
}
```

Output:

```
Smallest element 12
Largest element 88
```

16.Scala – Traits

Traits can be thought of as interfaces in which it's possible, but not necessary, to provide implementations for all or some of the methods that the interface defines. Or, by looking at traits from a different perspective we can also think of them as classes that can't have constructor parameters.

Using a trait like a Java interface

A trait can be used just like a Java interface. As with interfaces, just define the methods in your trait that you want extending classes to implement:

```
rait BaseSoundPlayer {
  def play
  def close
  def pause
  def stop
  def resume
}
```

Extending traits

If a class implements one trait it will use the extends keyword:

```
class Mp3SoundPlayer extends BaseSoundPlayer {
  def play   {}
  def close  {}
  def pause  {}
  def stop   {}
  def resume {}
}
```

One trait can extend another trait:

```
trait Mp3BaseSoundFilePlayer extends BaseSoundFilePlayer {
  def getBasicPlayer:BasicPlayer
  def getBasicController:BasicController
  def setGain(volume: Double)
}
```

If a class extends a trait but does not implement the methods defined in that trait, it must be declared abstract:

```
// must be declared abstract because it does not implement BaseSoundPlayer methods
abstract class JavaSoundPlayer extends BaseSoundPlayer {
  def play   {}
  def close  {}
}
```

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:

```
abstract class Animal {
  def speak
}

trait WaggingTail {
  def startTail
  def stopTail
}

trait FourLeggedAnimal {
  def walk
  def run
}

class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  // implementation code here ...
}
```

Abstract and concrete fields in traits

In a trait, define a field with an initial value to make it concrete, otherwise give it no initial value to make it abstract:

```
trait PizzaTrait {
  var numToppings: Int     // abstract
  val maxNumToppings = 10  // concrete
}
```

In the class that extends the trait, you'll need to define the values for the abstract fields, or make the class abstract. The Pizza class demonstrates how to override the numToppings field:

```
class Pizza extends PizzaTrait {
  var numToppings = 0      // override not needed
}

trait PizzaTrait {
  var numToppings: Int     // abstract
  val maxNumToppings = 10  // concrete
}
```

Using a trait like a Java abstract class

You can use traits like abstract classes in Java. In the following example, an implementation is provided for the speak method in the Pet trait, so implementing classes don't have to override it. The Dog class chooses not to override it, while the Cat class does:

```
trait Pet {
  def speak { println("Yo") }   // concrete implementation of a speak method
  def comeToMaster              // abstract
```

```
}

class Dog extends Pet {
  // don't need to implement 'speak' if you don't want to
  def comeToMaster { ("I'm coming!") }
}

class Cat extends Pet {
  // override 'speak'
  override def speak { ("meow") }
  def comeToMaster { ("That's not gonna happen.") }
}
```

If a class extends a trait without implementing its abstract methods, it must be defined abstract. Because FlyingPet does not implement comeToMaster, it is defined abstract:

```
abstract class FlyingPet extends Pet {
  def fly { ("I'm flying!") }
}
```

Using traits as simple mixins

To implement a simple mixin, define the methods you want in your trait, then add the trait to your class using extends or with. In the following example we define a Tail trait:

```
trait Tail {
  def wagTail { println("tail is wagging") }
}
```

This trait can be used with an abstract Pet class to create a Dog:

```
abstract class Pet (var name: String) {
  def speak  // abstract
  def ownerIsHome { println("excited") }
}

class Dog (name: String) extends Pet (name) with Tail {
  def speak { println("woof") }
}
```

A Dog can now use the methods defined by both the abstract Pet class, as well as the Tail trait:

```
object Test extends App {
  val zeus = new Dog("Zeus")
  zeus.ownerIsHome
  zeus.wagTail
  zeus.speak
}
```

Limiting which classes can use a trait

You can limit a trait so it can only be added to classes which extend a specific subclass. To do this, use the “trait [TraitName] extends [SubclassName]” syntax. For instance, in the following example the Starship and WarpCore both extend the common superclass StarfleetComponent, so the WarpCore trait can be mixed into the Starship class:

```
class StarfleetComponent
trait WarpCore extends StarfleetComponent
```

```
class Starship extends StarfleetComponent with WarpCore
```

However, in the following example, the Warbird can't extend the WarpCore trait because Warbird and WarpCore don't share the same superclass:

```
class StarfleetComponent
trait WarpCore extends StarfleetComponent
class RomulanStuff
class Warbird extends RomulanStuff with WarpCore   // won't compile
```

A trait inheriting a class is not a common occurrence, and in general the following approach is preferred.

You can mark your traits so they can only be used by subclasses of a certain type. To do this, begin your trait with the "this: LimitingType =>" statement, as shown here:

```
trait MyTrait {
  this: LimitingType =>
  // more code here ...
```

For instance, to make sure a WarpCore can only be used in a Starship, mark the WarpCore trait like this:

```
trait WarpCore {
  this: Starship =>
}
```

Now this code will work:

```
class Starship
class Enterprise extends Starship with WarpCore
```

But other attempts like this will fail:

```
class RomulanShip
class Warbird extends RomulanShip with WarpCore   // this won't compile
```

Adding a trait to an object instance

This next example shows that you can add a trait to an instance of a class (an object):

```
class DavidBanner

trait Angry {
  println("You won't like me ...")
}

object Test extends App {
  val hulk = new DavidBanner with Angry
}
```

Extending Java interfaces like Scala traits

You can extend Java interfaces like Scala traits. In your Scala application, just use the extends and with keywords to implement your Java interfaces:

```
// java
public interface Animal {
```

```
  public void speak();
}

public interface Wagging {
  public void wag();
}

public interface Running {
  public void run();
}

// scala
class Dog extends Animal with Wagging with Running {
  def speak { println("Woof") }
  def wag { println("Tail is wagging!") }
  def run { println("I'm running!") }
}
```

17.Scala – Pattern Matching

In this match method is used instead switch statement. The match method takes a number of cases as an argument. Each alternative takes a pattern and one or more expressions which will be performed if the pattern matches. A symbol => is used to separate the pattern from the expressions.

Scala - Using a Match Expression Like a switch Statement

Problem: You have a situation where you want to create something like a simple Java integer-based switch statement, such as matching the days in a week, months in a year, and other situations where an integer maps to a result.

Solution

To use a Scala match expression like a Java switch statement, use this approach:

```
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 whoa  => println("Unexpected case: " + whoa.toString)
}
```

That example shows how to take an action based on a match. A more functional approach returns a value from a match expression:

```
val month = 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"  // the default, catch-all
}
```

Pattern matching is a feature that is not unfamiliar in a lot of functional languages and Scala is no exception.It matches a value against several patterns. Each pattern points to an expression. The expression that is associated with the the first matching pattern, will be executed.

The syntax of pattern matching in Scala is defined as follows:

e match { case p1 => e1 ... case pn => en }

18.Scala – Regular Expressionns

Scala supports regular expressions through Regex class available in the scala.util.matching package.

Creating regular expressions

Scala provides a very simple way to create regexes: just define a regex as a string and then call the r method on it.

```
scala> val AmBn = "a+b+".r
AmBn: scala.util.matching.Regex = a+b+
```

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.

To use meta-characters, like \s, \w, and \d, you must either escape the slashes or use multiquoted strings, which are referred to as raw strings. The following are two equivalent ways to write a regex that covers strings of a sequence of word characters followed by a sequence of digits.

```
scala> val WordDigit1 = "\\w+\\d+".r
ordDigit1: scala.util.matching.Regex = \w+\d+

scala> val WordDigit2 = """\w+\d+""".r
WordDigit2: scala.util.matching.Regex = \w+\d+
```

Matching with regexes

We saw above that using the r method on a String returns a value that is a Regex object (more on the scala.util.matching part below). How do you actually do useful things with these Regex objects? There are a number of ways. The prettiest, and perhaps most common for the non-computational linguist, is to use them in tandem with Scala's standard pattern matching capabilities. Let's consider the task of parsing names and turning them into useful data structures that we can do various useful things with.

```
scala> val Name = """(Mr|Mrs|Ms)\. ([A-Z][a-z]+) ([A-Z][a-z]+)""".r
Name: scala.util.matching.Regex = (Mr|Mrs|Ms)\. ([A-Z][a-z]+) ([A-Z][a-z]+)

scala> val Name(title, first, last) = "Mr. James Stevens"
title: String = Mr
first: String = James
last: String = Stevens

scala> val Name(title, first, last) = "Ms. Sally Kenton"
title: String = Ms
first: String = Sally
last: String = Kenton
```

Notice the similarity with pattern matching on types like Array and List.

```
scala> val Array(title, first, last) = "Mr. James Stevens".split(" ")
title: java.lang.String = Mr.
first: java.lang.String = James
last: java.lang.String = Stevens

scala> val List(title, first, last) = "Mr. James Stevens".split(" ").toList
title: java.lang.String = Mr.
first: java.lang.String = James
last: java.lang.String = Stevens
```

Exception Handling in Scala :

Error Handling in Scala

First, let's define what we mean by failure.

- Unexpected internal failure: the operation fails as the result of an unfulfilled expectation, such as a null pointer reference, violated assertions, or simply bad state.
- Expected internal failure: the operation fails deliberately as a result of internal state, i.e. a blacklist or circuit breaker.
- Expected external failure: the operation fails because it is told to process some raw input, and will fail if the raw input cannot be processed.
- Unexpected external failure: the operation fails because a resource that the system depends on is not there: there's a loose file handle, the database connection fails, or the network is down.

Scala Exception handling is similar to exception handling in Java. However Scala does not support checked exceptions.

Exceptions are the events which can change the flow of control through a program. When you want to handle exceptions, you use a try{…}catch{…} block .

Scala does not have checked exceptions like Java, so you can't do soemthing like this to force a programmer to deal with an exception:

Scala has an exception mechanism similar to Java's.

Exception handling is a rather large topic to cover in full detail. Therefore, I will be writing more about Scala exception handling in a different, more specialized trail. In this text I will just explain the exception handling mechanisms briefly.

Throw Exception

A Scala method can throw an exception instead of returning normally, in case an error occurs. Here is an example which should look familiar to Java developers:

```
//do something
//do something more

throw new IllegalArgumentException("arg 1 was wrong...");
```

```
//nothing is executed after the throw.
```

When an exception is thrown the normal thread of execution is interrupted, and the exception is propagated up the call stack until a catch clause catches it.

Try Catch

If you execute a sequence of code that might throw an exception, and you would like to handle that exception, you use the try-catch block. Here is an example:

```
try{
   throwsException();
   println("this line is never executed");
} catch {
  case e: Exception => println("exception caught: " + e);
}


//A METHOD THAT THROWS EXCEPTION
def throwsException() {
   throw new IllegalStateException("Exception thrown");
}
```

When an exception is thrown from inside the throwsException() method, the execution is interrupted and the execution jumps to the catch clause surrounding the code that threw the exception.

In the example above, when the throwsException() method is called, and it throws an exception, the statement below the throwsException() method call is never executed. Instead the execution jumps down to the catch clause.

The catch clause looks a bit different from a Java catch clause. Rather than declare the exception to catch in parantheses after the catch keyword, Scala requires that you write a list of case statements.

In the example above, only one exception is caught, Exception which is the superclass of most exceptions. Hence, most exceptions are caught using that catch clause. If you wanted to be more specific in what exceptions to catch, provide multiple case statements. For instance:

```
catch {
  case e: IllegalArgumentException => println("illegal arg. exception");
  case e: IllegalStateException    => println("illegal state exception");
  case e: IOException              => println("IO exception");
}
```

Finally

The finally clause can contain code that you need to be executed, no matter if an exception is thrown or not. Here is an example:

```
try {
   throwsException();
} finally {
   println("this code is always executed");
}
```

A finally clause is appended to a try block. You can have a catch clause inthere too, if you like. The

finally clause is still always executed.

The finally clause normally contains code that must be executed no matter whether an exception is thrown or not. For instance, say you were processing a stream of data. You would want to close that stream of data, regardless of whether your processing succeeded, or resulted in an exception being thrown.

Here is a full example, with both a try, catch and finally clause:

```
try {

  throwsException();

} catch {

  case e: IllegalArgumentException => println("illegal arg. exception");
  case e: IllegalStateException    => println("illegal state exception");
  case e: IOException              => println("IO exception");

} finally {

  println("this code is always executed");

}
```

Scala Exception Handling Example

Throwing Exceptions in Scala

Simply create an exception object and then you throw it with the throw keyword, for example;

```
throw new ArithmeticException
```

Scala try catch blocks

Scala allows us to try/catch an exception and perform pattern matching using case blocks.

Consider an example below.

```
object Arithmetic {

def main(args: Array[String]) {
  Try {
     val z = 4/0
     } catch {
           case ex: ArithmeticException => {
           println("Cannot divide a number by zero")
        }
                     }
        }
        }
```

Here we are trying to divide a number by zero and catch the arithmetic exception in the catch block. The case Arithmetic exception is matched and the statement “Cannot divide a number by zero” is
printed.

```
scala> object Arithmetic {
     |    def main(args: Array[String]) {
     |       try {
     |          val z = 4/0
     |       } catch {
     |                    case ex: ArithmeticException => {
     |              println("Cannot divide a number by zero")
     |          }
     |       }
     |    }
     | }
defined object Arithmetic

scala> Arithmetic.main(null);
Cannot divide a number by zero

scala>
```

Scala finally clause
finally clause executes the code irrespective of whether the expression/program prematurely terminates or successfully executes.

Consider an example below.

```
object Arithmetic {
   def main(args: Array[String]) {
      try {
      val z = 4/0
      } catch {
         case ex: ArithmeticException => {
       println("Cannot divide a number by zero")
          }
      }
    finally {
        println("This is final block")
           }
   }
   }
```

Scala Extractors apply, unapply and pattern matching

Scala extractor is an object which has a method called unapply as one of its members. The unapply method matches a value and take it apart. The extractor also defines apply method for building values.

Consider an example of extracting first name and last name that uses space as separator.

firstName.scala

```
object firstName {
          def main(args: Array[String]) {
                    println("Apply method : " + apply("Steve", "Smith"));
        println("Unapply method : " + unapply("Steve Smith"));
        println("Unapply method : " + unapply("Rob"));
            }
   def apply(fname: String, lname: String) = {
         fname + " " + lname
         }
   def unapply(s: String): Option[(String, String)] = {
         val pts = s split " "
         if (pts.length == 2) {
            Some(pts(0), pts(1))
            } else {
            None
              }
          }
      }
```

The object firstName defines two methods apply and unapply. The apply method turns into an object that accepts the arguments specified within the parenthesis and builds the value as specified in the method. The first and last name combined together with a space in between is returned.

The unapply method returns firstname object into an extractor and breaks the arguments as specified in the method. The unapply methods accepts a String argument and splits the String with space as separator and returns the option type over pair of strings. The result is a pair of strings if the first name and last name is passed as an argument else returns none.

Scala Extractor Pattern Matching

If the instance of the class contains a list of zero or more parameters, compiler calls apply method on that instance. The apply method can be defined for both objects and classes.

```
case class Student(name: String, address: Seq[Address])
case class Address(city: String, state: String)

object City {
def unapply(s: Student): Option[Seq[String]] =
Some(
for (c <- s.address)
yield c.state)
}

class StringSeqContains(value: String) {
def unapply(in: Seq[String]): Boolean =
in contains value
}

object PatternMatch {
def main(args: Array[String]) {

val stud = List(Student("Harris", List(Address("LosAngeles", "California"))),
Student("Reena", List(Address("Houston", "Texas"))),
Student("Rob", List(Address("Dallas", "Texas"))),
```

```
Student("Chris", List(Address("Jacksonville", "Florida"))))

val Texas = new StringSeqContains("Texas")
val students = stud collect {
case student @ City(Texas()) => student.name
}
println(students)
}
}
```

Scala File IO – Write File, Read File

Scala is open to make use of any Java objects and java.io.File is one of the objects which can be used in Scala programming to read and write files.

Scala Read File

We can use scala.io.Source to read data from a file. For reading a file, we have created a test file with below content.

data.tx

Scala vs java

Which is best programming

Why I should choose scala rather then Java

Why scala choosed for Spark programing

Here is a simple program where we are using Scala Source class to read file data to a String and then split it using regular expression to a Map. Finally we are printing the count of JournalDev in the file content.

Wordcount.scala

```
import scala.io.Source

object Wordcount {

  def main(args:Array[String]) {

  println(Source.fromFile("data.txt")) // returns scala.io.BufferedSource non-empty iterator instance

  val s1 = Source.fromFile("data.txt").mkString; //returns the file data as String
  println(s1)

  //splitting String data with white space and calculating the number of occurrence of each word in the file
  val counts = s1.split("\\s+").groupBy(x=>x).mapValues(x=>x.length)

  println(counts)

  println("Count of Scala:"+counts("Scala”)
```

```
  }

}
```

Word count line by line: Sometimes there arises a need to process each line rather than the whole contents of the file. This can be achieved through the getLines method. For example below code;

```
println(Source.fromFile("data.txt").getLines())

Source.fromFile("data.txt").getLines.foreach { x => println(x) };
```

output:

Scala vs java

Which is best programming

Why I should choose scala rather then Java

Why scala choosed for Spark programing

Now let us see how to extract specific set of lines.

```
Source.fromFile("data.txt").getLines.take(1).foreach { x => println(x) };

Source.fromFile("data.txt").getLines.slice(0, 1).foreach { x => println(x) };
```

take(n) method is used to Select first n values of the iterator where slice(from, until) returns part of the iterator where “from” index is the part of slice and “until” index is not. So both the lines in above code snippet are doing the same thing.

Scala Write to File

Scala standard library doesn’t contain any classes to write files, so we will use Java IO classes for scala write to file. Below is a simple program showing how to write files in scala.

```
import java.io.File

import java.io.PrintWriter

import scala.io.Source

object Write {
  def main(args: Array[String]) {
    val writer = new PrintWriter(new File("Write.txt"))

    writer.write("Hello Developer, Welcome to Scala Programming.")
    writer.close()

    Source.fromFile("Write.txt").foreach { x => print(x) }
  }

}
```

This will produce a file Write.txt with given content and then Source will read and print the same

file data.

That’s all for Scala File IO example. You can see how simple it is to read file in Scala and write file in Scala using Java IO classes.
