skillspeed

for the serious learner

Apache Spark and Scala

Module 3: Scala – Essentials and Deep Dive

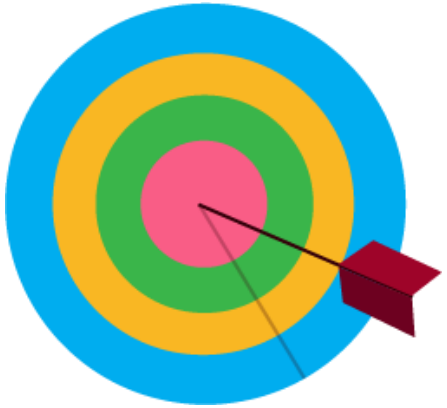
Course Topics

Module 1	Module 2	Module 3	Module 4
Getting Started / Introduction to Scala	RDD and Spark Streaming	Scala Basics	SparkSQL – Real-time Analysis

Session Objectives

In this session, you will be able to understand

- Data Types in Scala
- Variable Types in Scala
- Lazy Values
- Control Structures in Scala
- Functions
- Procedures
- Collections
- Reserved Words
- Pattern Matching
- Enumeration
- Ternary Operators



Data Types in Scala

- A Data type tells the compiler about the type of the value to be stored in a location
- 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

Data Types in Scala (cont'd)

Few Examples:

```
scala> "Hello Skillspeed"
res11: String = Hello Skillspeed
```

String result

```
scala> 10
res12: Int = 10
```

Integer Result

```
scala> 3.5
res13: Double = 3.5
```

Double result

```
scala> "2"+"5"
res14: String = 25
```

Concatenation of two integers, results string(25) not int(5)

```
scala> "skill"+"speed"
res15: String = skillspeed
```

Concatenation of two strings , results string

```
scala> 2+3
res16: Int = 5
```

Adding two numbers, results integers

```
scala> "Hello"+9
res17: String = Hello9
```

Adding string and Integers results String

Variables Types in Scala

Variables are simply names used to refer to some location in memory – a location that holds a value with which we are working

Scala variables come in two shapes: Values and Variables

Values:

Immutable - "val" (Read only)

- Similar to Java Final Variables
- Once initialized, Vals can't be reassigned

```
scala> val myVal="Hello Skillspeed"
myVal: String = Hello Skillspeed

scala> myVal="Welcome"
<console>:8: error: reassignment to val
       myVal="Welcome"
            ^
```

Vals can't be reassigned

Variables Types in Scala (Cont'd)

Variables:

Mutable - "var" (Read-write) - Similar to non-final variables in Java

Here, myVar is declared using the keyword var. This means that it is a variable that can change value and this is called mutable variable

```
scala> var myVar="Hello Skillspeed"
myVar: String = Hello Skillspeed

scala> myVar="Welcome"
myVar: String = Welcome
```

Vars can be reassign

Lazy Values

- One nice feature built into Scala are "lazy val" values.
- Lazy value initialization is deferred till it's accessed for first time
- For example : If you want to read a file abc.txt, if the file is not existing , you will get FileNotFoundException exception

```
scala> val file=scala.io.Source.fromFile("file.txt").mkString
java.io.FileNotFoundException: file.txt (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:146)
        at scala.io.Source$.fromFile(Source.scala:90)
        at scala.io.Source$.fromFile(Source.scala:75)
        at scala.io.Source$.fromFile(Source.scala:53)
        at .<init>(<console>:7)
        at .<clinit>(<console>)
        at .<init>(<console>:7)
        at .<clinit>(<console>)
        at $print(<console>)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

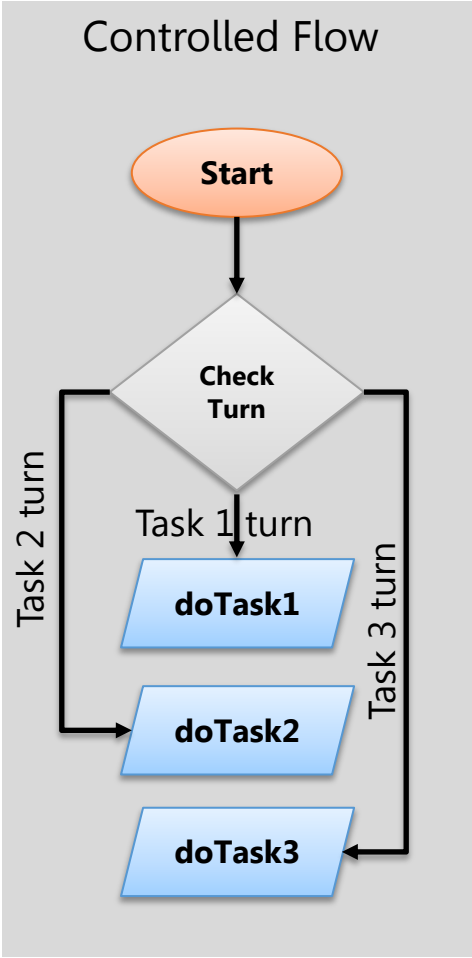
- But if you initialize the value as Lazy, you won't get this error, because it will delay the initialization till it accesses the file abc.txt

```
scala> lazy val file=scala.io.Source.fromFile("file.txt").mkString
file: String = <lazy>
```

Lazy Values (cont'd)

- Lazy values are very useful for delaying costly initialization instructions
- Lazy values don't give error on initialization, whereas no lazy value do give error

Control Structures in Scala



- Control Structures controls the flow of execution
- Scala provides various tools to control the flow of program's execution
- Some of them are:
  - if..else
  - while
  - do-while
  - foreach
  - for

Control Structures in Scala (cont'd): if-else

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

- if-else syntax in Scala is same as Java or C++
- In Scala, if-else has a value, of the expression following it
- Semicolons are optional in Scala

```
scala> val x= -10;
x: Int = -10

scala> val s= if(x > 0) "Positive" else "Negative"
s: String = Negative
```

Every expression in Scala has a type. First If statement has a type Int

Second statement has a type Any. Type of a mixed expression is supertype of both branches

```
scala> val s= if(x > 0) "Positive" else 0
s: Any = 0
```

Control Structures in Scala (cont'd) : While Loop

- A **while** loop statement repeatedly executes a target statement as long as a given condition is true
- In Scala while and do-while loops are same as Java

Syntax:

```
While(condition)
{
// Block of code ;
}
```

```
scala> var x=5
x: Int = 5

scala> while( x < 10 ){
     | println( " Value of x : " + x);
     | x = x + 1;
     | }
 Value of x : 5
 Value of x : 6
 Value of x : 7
 Value of x : 8
 Value of x : 9
```

Note: The ++i, or i++ operators don't work in Scala, use i+=1 or i=i+1 expressions instead

```
scala> var x=5
x: Int = 5

scala> while( x < 10 ){
     | println( " Value of x : " + x);
     | x = x++;
     | }
<console>:11: error: value ++ is not a member of Int
              x = x++;
                    ^
```

Control Structures in Scala (cont'd): do- While Loop

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time

Syntax:

```
do
{
//Block of code
} while(condition);
```

```
scala> var x=5
x: Int = 5

scala> do{
     | x = x + 1;
     | print("value of x : " +x)
     | }while(x<5)
value of x : 6
scala>
```

Control Structures in Scala (cont'd): foreach Loop

skillspeed
for the serious learner

Looping with foreach:

```
scala> var args="Welcome"
args: String = Welcome

scala> args.foreach(arg => println(arg))
W
e
l
c
o
m
e

scala>
```

```
scala> args.foreach(println)
W
e
l
c
o
m
e

scala>
```

Control Structures in Scala (cont'd): for loop

skillspeed
for the serious learner

for Loop:

for loop can execute a block of code for specific number of times.

Scala doesn't have for (initialize; test; update) syntax

```
for( var x <- n )  {                    here,  n -- > Range
   //Block of statements;        <- operator is called a generator
}
```

Scala: For Loop : 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:

```
scala> for(x <- 1 to 5) {
     |     println("x is " + x);
     | }
x is 1
x is 2
x is 3
x is 4
x is 5
```

Including 5

```
scala> for(x <- 1 until 5) {
     |     println("x is " + x);
     | }
x is 1
x is 2
x is 3
x is 4
```

Excluding 5

The first loop iterates 5 times, from 1 to 5 including 5

The second loop iterates 4 times, from 1 to 4, excluding the upper boundary value 5

Control Structures in Scala: for Loop (cont'd)

While traversing an array, following could be applied:

```
scala> val txt ="skillspeed"
txt: String = skillspeed

scala> var sum=0
sum: Int = 0

scala> for(i<-0 until txt.length) sum += i

scala> println(sum)
45
```

Advanced For Loop: can have multiple generators in for loop

```
scala> for(i<- 1 to 3; j<-1 to 3) println(5*i + j+1)
7
8
9
12
13
14
17
18
19
```

Control Structures in Scala: for Loop(cont'd)

We can put conditions in multi generators for loop

```
for(i<- 1 to 3; j<-1 to 3 if i ==j) println(5*i + j+1)
```

```
scala> for(i<- 1 to 3; j<-1 to 3 if i ==j) println(5*i + j+1)
7
13
19

scala>
```

We can introduce variables in loop!

```
for(i<- 1 to 3;  x = 4-i; j<- x to 3) println(5*i + j+1)
```

```
scala> for(i<- 1 to 3;  x = 4-i; j<- x to 3) println(5*i + j+1)
9
13
14
17
18
19
```

Control Structures in Scala: The for Loop with Yield

If the body of for loop starts with yield, it returns a collection of values

```
val x = for(i<- 1 to 10) yield i*5
```

```
for (i<- x) println(i)
```

```
scala> val x = for(i<- 1 to 10) yield i*5
x: scala.collection.immutable.IndexedSeq[Int] = Vector(5, 10, 15, 20, 25, 30, 35, 40, 45, 50)

scala> for (i<- x) println(i)
5
10
15
20
25
30
35
40
45
50
```

Functions

A function is a group of statements that together perform a task

Scala function is a complete object which can be assigned to a variable

The last statement in the function is the return value.

You can create functions with “def” keyword

Syntax:

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

Note: In Java, this concept is very close to a method

```
scala> def myFun( Eno:Int, Ename:String )={
     | println("Employee number : "+ Eno);
     | println("Employee Name : " +Ename);
     | }
myFun: (Eno: Int, Ename: String)Unit
```

```
scala> myFun(1033,"Robert")
Employee number : 1033
Employee Name : Robert
```

Functions (cont'd)

Named and Default Arguments

We can provide defaults to function arguments, which will be used in case no value is provided in function calls

```
scala> def DefaultFun(Ename:String,Cname:String="Skillspeed"){
     | println("Employee Name: " + Ename);
     | println("Company Name : " + Cname);
     | }
DefaultFun: (Ename: String, Cname: String)Unit

scala> DefaultFun("Robert")
Employee Name: Robert
Company Name : Skillspeed
```

- We can specify argument names in function calls
- In named invocations the order of arguments is not necessary
- We can mix unnamed and named arguments, if the unnamed argument is the first one. We can specify argument names in function calls
- In named invocations the order of arguments is not necessary
- We can mix unnamed and named arguments, if the unnamed argument is the first one

Variable Arguments

- Scala supports variable number of arguments to a function

Check your Understanding – 3

What is the output of the following?

```
def concatStr(a:String, b:Int=2 , c:String) = {a + b + c}
println(concatStr( "Hi",200, "Welcome"))
```

a) Hi2Welocme

b) Hi200Welcome

c) Error

d) Hi2200Welcome

Check your Understanding – Solution

What is the output of the following?

```
def concatStr(a:String, b:Int=5 , c:String) = {a + b + c}
println(concatStr( "Hi",200, "Welcome"))
```

a) Hi5Welocme

b) ✓ Hi200Welcome

c) Error

d) Hi5200Welcome

Error

Procedures

Scala has a special notation for a function that returns no value
If the function body is enclosed in braces without a preceding = symbol, then the return type is Unit

Such functions are called Procedures. Procedures do not return any value in Scala

Example:

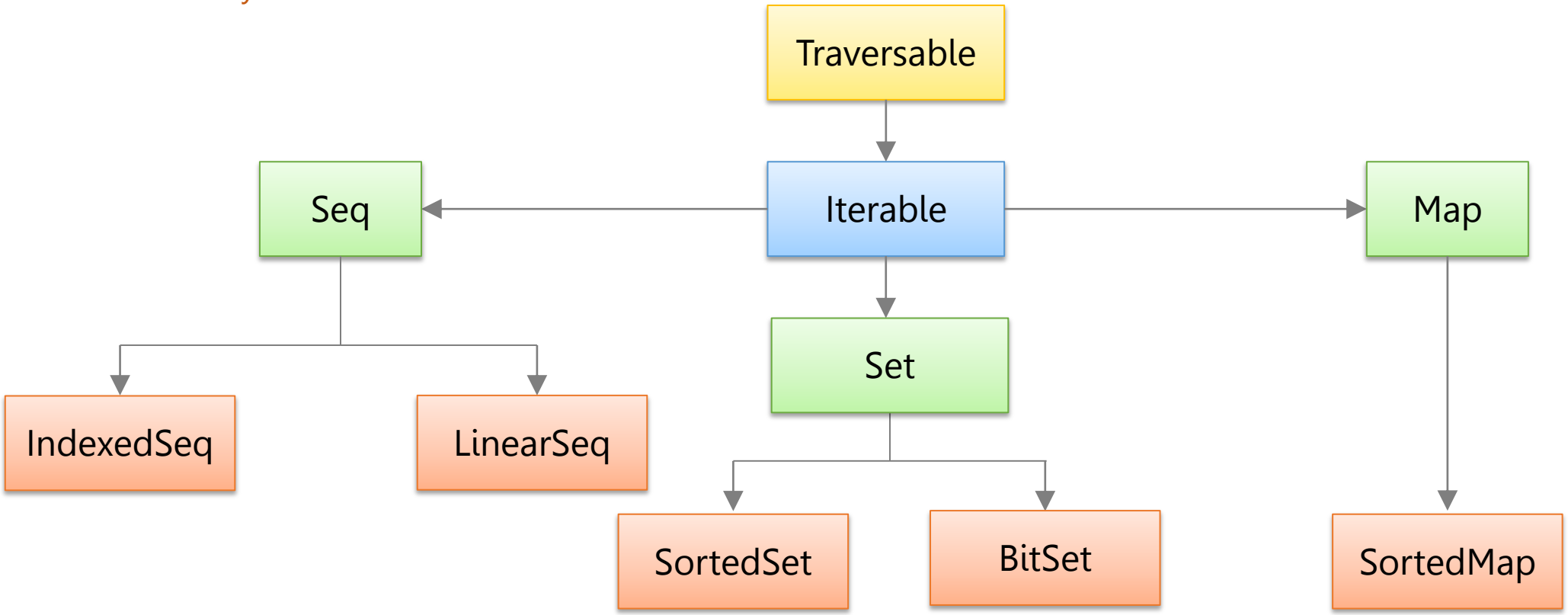
```
scala> def Display(s : String) {
     |   val border = "-" * s.length + "--\n"
     |   println(border + "|" + s + "|\n" + border)
     | }
Display: (s: String)Unit

scala> Display("Skillspeed")
------------
|Skillspeed|
------------
```

Same rules of default and named arguments apply on Procedures as well

Scala: Collections

- Scala has a rich set of collection library
- Collections are containers that hold objects
- Those containers can be sequenced, linear sets of items like Arrays, List, Tuple, Option, Map, etc.

Collections hierarchy



Collections

Collections can be mutable and immutable

Scala collections systematically distinguish between mutable and immutable collections

Mutable collection:

- A mutablecollection can be updated or extended in place
- This means you can change, add, or remove elements of a collection as a side effect

Immutable collections:

- By contrast, never change
- You have still operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged

Scala Collections: Array

Arrays are mutable, indexed collections of values

Array[T] is Scala's representation for Java's T[]

Declaring Arrays:

```
scala> val numbers= Array(1, 2, 3, 4)
numbers: Array[Int] = Array(1, 2, 3, 4)

scala> val Courses= Array("scala","Python","Java")
Courses: Array[String] = Array(scala, Python, Java)
```

Integer Array

String Array

Accessing arrays :

```
scala> numbers(3)
res18: Int = 4

scala> Courses(1)
res19: String = Python
```

Scala Collections: Array (cont'd)

skillspeed
for the serious learner

Fixed Length Arrays:

Examples:

```
scala> var myIntArray= new Array[Int](10)
myIntArray: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> var myStrArray= new Array[String](5)
myStrArray: Array[String] = Array(null, null, null, null, null)

scala> val myArray=Array("Hello","Welcome")
myArray: Array[String] = Array(Hello, Welcome)
```

Accessing Arrays:

```
scala> myStrArray(0)="Scala"

scala> myStrArray(1)="Python"

scala> myStrArray
res63: Array[String] = Array(Scala, Python, null, null, null)
```

Scala Collections: ArrayBuffers

An ArrayBuffer buffer holds an array and a size. Most operations on an array buffer have the same speed as for an array, because the operations simply access and modify the underlying array

```
scala> var cars = ArrayBuffer[String]()
cars: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer()
```

Array buffers can have data efficiently added to the end

```
scala> cars += "BMW"
res65: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW)

scala> cars += "Jaguar"
res66: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW, Jaguar)
```

```
scala> cars
res69: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW, Jaguar, Audi, Rolls Royce)
```

```
scala> println(cars.length)
4
```

Scala Collections: ArrayBuffers (cont'd)

cars.trimEnd(1) : Removes the last Element

```
scala> cars.trimEnd(1)

scala> cars
res74: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW, Jaguar, Audi)
```

// Adds element at 2nd index

```
scala> cars.insert(2,"Bentley")

scala> cars
res88: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW, Jaguar, Bentley, Audi)
```

Adds a list

```
scala> cars.insert(1,"fiat","Volvo","Renault")

scala> cars
res98: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW, fiat, Volv
o, Renault, Jaguar, Bentley, Audi)
```

Scala Collections: ArrayBuffers (cont'd)

skillspeed
for the serious learner

//Removes an element

```
scala> cars.remove(3)
res99: String = Renault

scala> cars
res100: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW, fiat, Vol
vo, Jaguar, Bentley, Audi)
```

//Removes three elements from index 1

```
scala> cars.remove(1,3)

scala> cars
res102: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(BMW, Bentley,
Audi)
```

Scala Collections: Maps

- A 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

```
scala> var mapping=Map("NY"->"New York","NJ"->"New Jersy")
mapping: scala.collection.immutable.Map[String,String] = Map(NY -> New York, NJ -> New Jersy)
```

Accessing immutable Maps:

```
scala> var x=mapping("NY")
x: String = New York
```

Accessing with keys

```
scala> var x= mapping("New Jersy")
java.util.NoSuchElementException: key not found: New Jersy
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  at scala.collection.MapLike$class.apply(MapLike.scala:141)
  at scala.collection.AbstractMap.apply(Map.scala:59)
  ... 33 elided

scala>
```

Can't access with values

Scala Collections: Maps (cont'd)

If there is a sensible default value for any key that might try with map, it can use the **getOrElse** method

it provides the key as the first argument, and then the default value as the second

```
scala> mapping.getOrElse("NY","???")
res22: String = New York

scala> mapping.getOrElse("New Jersy","???")
res23: String = ???
```

It is quite common to use getOrElse with a default of 0

```
scala> val x = mapping.getOrElse("NY",0)
x: Any = New York

scala> val x = mapping.getOrElse("NJ",0)
x: Any = New Jersy
```

Scala Collections: Mutable Maps

To create a mutable Map, import it first:

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

Create a map with initial elements

```
scala> var states = scala.collection.mutable.Map("NY"->"New York","WY"->"Wyomming
")
states: scala.collection.mutable.Map[String,String] = Map(WY -> Wyomming, NY -> N
ew York)
```

add elements with +=

```
scala> states += ("CA"->"California","NJ"->"New Jersy")
res105: scala.collection.mutable.Map[String,String] = Map(WY -> Wyomming, NJ -> N
ew Jersy, NY -> New York, CA -> California)
```

remove elements with -=

```
scala> states -= ("WY")
res106: scala.collection.mutable.Map[String,String] = Map(NJ -> New Jersy, NY ->
New York, CA -> California)
```

Scala Collections: Mutable Maps (cont'd)

Update elements by reassigning them

```
scala> states("NY") = "New York, The Big State"
```

```
scala> states
res108: scala.collection.mutable.Map[String,String] = Map(NJ -> New Jersy, NY ->
New York, The Big State, CA -> California)
```

Scala Collections: Tuples

A tuple is an ordered container of two or more values of same or different types

Unlike lists and arrays, however, there is no way to iterate through elements in a tuple

Its purpose is only as a container for more than one value

You create a tuple with the following syntax, enclosing its elements in parentheses

Here's a tuple that contains an Int and a String and Double

```
scala> var stuff=(101,"Robert",25000.00)
stuff: (Int, String, Double) = (101,Robert,25000.0)
```

Accessing the tuple elements:

```
scala> println(stuff._1)
101

scala> println(stuff._2)
Robert

scala> println(stuff._3)
25000.0
```

In tuples the offset starts with 1 and NOT from 0

Scala Collections: Tuples (cont'd)

Tuples are typically used for the functions which return more than one value:

```
scala> "skiLlspEed".partition(_.isUpper)
res13: (String, String) = (LE,skilsped)

scala> "skiLlspEed".partition(_.isLower)
res14: (String, String) = (skilsped,LE)
```

Scala Collections: Lists

Lists are quite similar to arrays, but there are two important differences.

First, lists are immutable. i.e., elements of a list cannot be changed .

Second , lists have a recursive structure whereas arrays are flat.

Class for immutable linked lists representing ordered collections of elements of type

This class comes with two implementing case classes scala.Nil and scala.:: that implement the abstract members is Empty, head and tail

Example:

```
scala> var num=List(1,2,3,4)
num: List[Int] = List(1, 2, 3, 4)

scala> var fruits=("Apple","Orange","Banana")
fruits: (String, String, String) = (Apple,Orange,Banana)

scala> val empty=List()
empty: List[Nothing] = List()
```

```
scala> num.head
res16: Int = 1

scala> num.tail
res17: List[Int] = List(2, 3, 4)
```

Scala Collections: Lists (cont'd)

:: operator adds a new List from given head and tail

```
scala> 1::List(2,3)
res18: List[Int] = List(1, 2, 3)
```

We can use iterator to iterate over a list, but recursion is a preferred practice in Scala

Example:

```
scala> def sum(l :List[Int]):Int = {if (l == Nil) 0 else l.head + sum(l.tail)}
sum: (l: List[Int])Int

scala> val y = sum(lst)
y: Int = 10
```

Reserved Words

- Reserved Keyword (also known as a Reserved Identifier) is a word that cannot be used as an identifier, such as the name of a variable, function, or label – it is reserved from use

- Few are listed here:

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

```
scala> val say= "Hello"
say: String = Hello

scala> var finally = "Hello"
<console>:1: error: illegal start of simple pattern
       var finally = "Hello"
           ^
```

finally is reserved keyword

Pattern Matching

Scala has a built-in general pattern matching mechanism

It allows to match on any sort of data with a first-match policy

Here is a small example which shows how to match against an integer value:

```
scala> object MatchTest1 extends App{
     | def matchTest(x:Int): String = x match {
     | case 1 => "one"
     | case 2 => "two"
     | case _ => "many"
     | }
     | println(matchTest(3))
     | }
defined object MatchTest1

scala>
```

Here is a second example which matches a value against patterns of different types:

```
scala> object MatchTest2 extends App {
     |   def matchTest(x: Any): Any = x match {
     |     case 1 => "one"
     |     case "two" => 2
     |     case y: Int => "scala.Int"
     |   }
     |   println(matchTest("two"))
     | }
defined object MatchTest2
```

Enumeration

Enumeration allows programmer to define their own data type

Often we have a variable that can take one of several values. For instance, a WeekDays field of an object could be either Mon, Tue, Wed, or Thu

In other languages such as C, Java, or Python, it is common to use a small integer to distinguish the possibilities

In Scala, we let the compiler create one object for each possibility, and we use a reference to that object

Here is the somewhat strange syntax to do this:

```
scala> object WeekDays extends Enumeration {
     |   val Mon,Tue,Wed,Thu,Fri = Value
     | }
defined object WeekDays

scala>
```

```
scala> var Days=WeekDays.Mon
Days: WeekDays.Value = Mon

scala> Days=WeekDays.Tue
Days: WeekDays.Value = Tue
```

Enumeration(Cont'd)

Another Way:

```
scala> import WeekDays._
import WeekDays._

scala> Days = Mon
Days: WeekDays.Value = Mon

scala> Days = Tue
Days: WeekDays.Value = Tue
```

Gives the error if value is not found

```
scala> Days = Sat
<console>:27: error: not found: value Sat
       Days = Sat
              ^
```

Ternary Operators

In other programming languages there is a definite, unique ternary operator syntax, but in Scala, the ternary operator is just the normal Scala if/else syntax

Example:

```
scala> val x=10
x: Int = 10

scala> val y=20
y: Int = 20

scala> if (i == 1) x else y
res27: Int = 20
```

Another Example, you can use the Scala ternary operator syntax on the right hand side of the equation, as you might be used to doing with Java:

```
scala> val loggedIn = false
loggedIn: Boolean = false

scala> val message = if (loggedIn) "Welcome" else "Please log in first"
message: String = Please log in first
```

Check your Understanding – 5

What is the output of the following?

```
val new = List(1,2,3,4) the new._2
```

a) 3

b) Error

c) 2

d) None of these

Check your Understanding – Solution

What is the output of the following?

```
val new = List(1,2,3,4) the new._2
```

a) 3

b) Error ✓

c) 2

d) None of these

Error

Questions

skillspeed
for the serious learner

thank you!