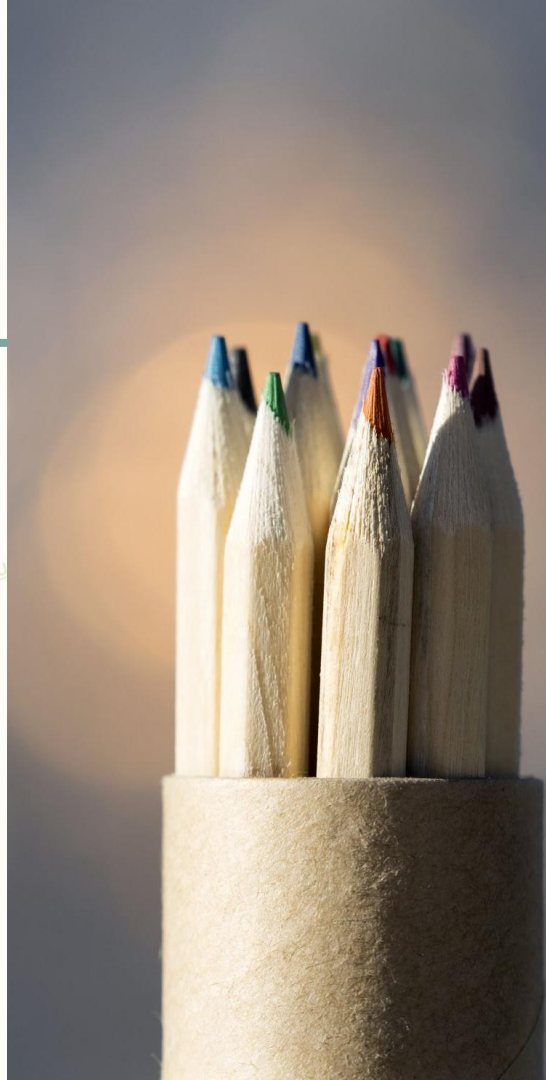
Scala

Muthukumaran Navaneethakrishnan

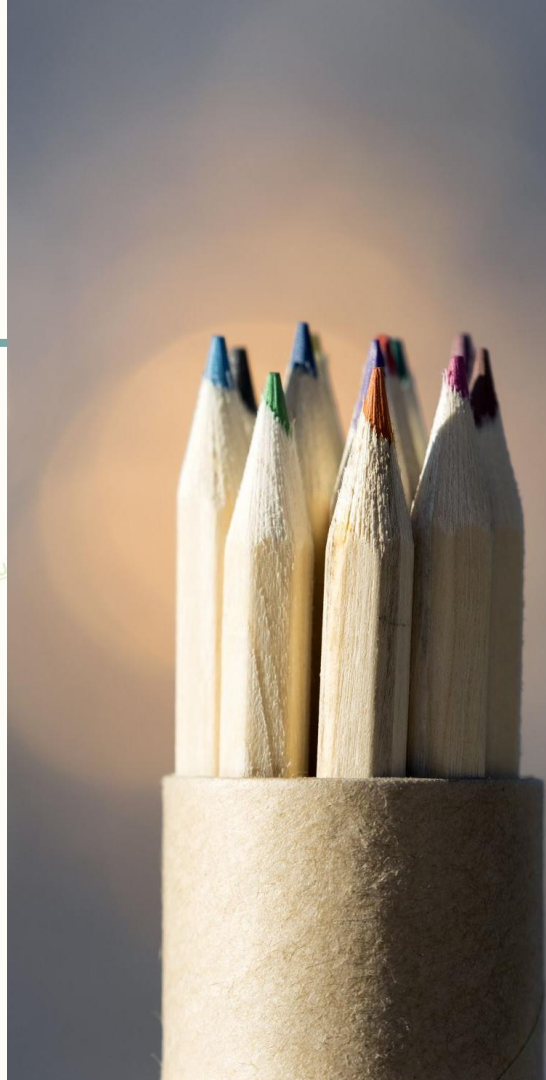
Contents – Day 1

- Historical Background
- Imperative& Declarative Programming
- Scala Introduction & Features
- Object
- Unit
- Print
- String Interpolation
- Functions
- Variables
- REPL
- Statically Typed with Local Type Inference
- Array
- Conditions
- Pattern Matching
- Underscore
- Loop
- Class
- Seq
- List & List Buffer
- HashMap



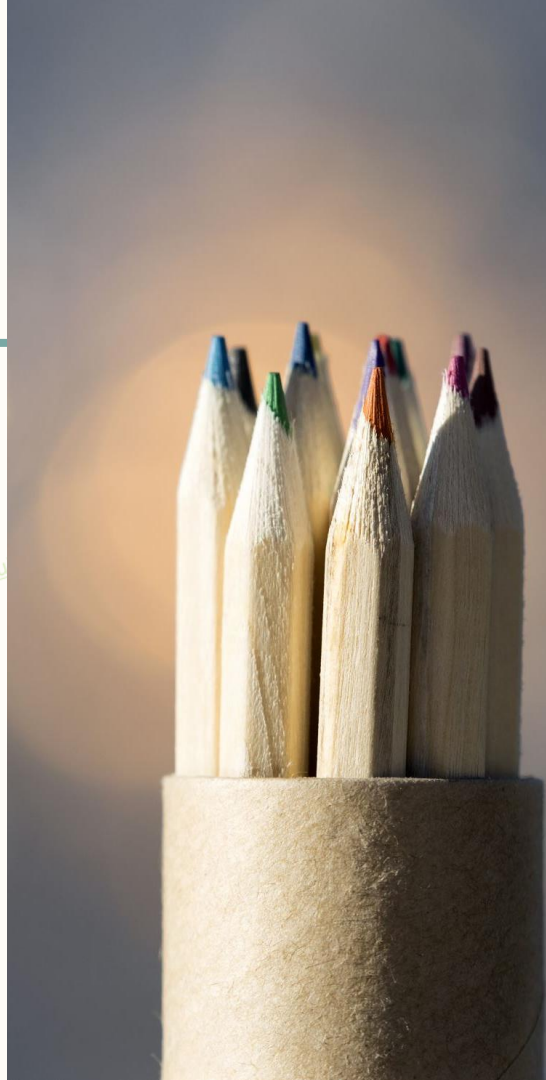
Contents – Day 2

- map
- filter
- reduce
- functional programming
- case class
- traits
- File I/O
- sbt
- Unit Testing
- Akka Framework

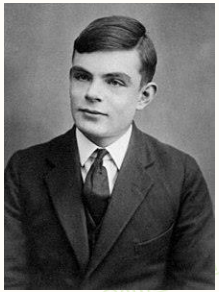


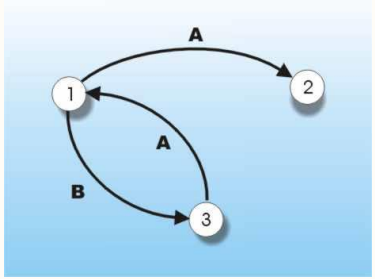
Contents – Day 3

- Play Framework
- Controllers
- View Templates
- Web UIs
- Restful Services
- Connecting with Database
- Authentication



Historical Background





Alan Turing develops Turing Machine, [1937] - Finite state machine with the ability to read and write data to a tape.

Historical Background

Alonzo Church develops the lambda calculus,[1932] a simple but powerful theory of functions.

Imperative Programming

a programming paradigm that uses statements that change a program's state.

Imperative Programming

```
int sum = 0;
for (int i = 1; i <= 10; ++i)
    sum = sum + i;
```

Declarative Programming

A style of programming that treats computation as the evaluation of mathematical functions

Eliminates side effects

Treats data as being immutable

Expressions have referential transparency

Functions can take functions as arguments and return functions as results

Declarative Programming

```
(1 to 10).toList.sum
```

Why do Functional Programming?

Allows us to write easier-to-understand, more declarative, more concise programs than imperative programming

Allows us to focus on the problem rather than the code

Facilitates parallelism

Origin

- Started at 2001 by Martin Odersky at EPFL Lausanne, Switzerland
- Scala 2.0 released in 2006
- Current version 2.13
- Twitter backend runs on Scala

Scala Announcement (15 years 11 months 9 days from now)

Subject: **ANNOUNCEMENT: The Scala Programming Language**
Newsgroups: **gmane.comp.lang.scala**
Date: 2004-01-20 17:25:18 GMT

We'd like to announce availability of the first implementation of the Scala programming language. Scala smoothly integrates object-oriented and functional programming. It is designed to express common programming patterns in a concise, elegant, and type-safe way. Scala introduces several innovative language constructs. For instance:

- Abstract types and mixin composition unify ideas from object and module systems.

- Pattern matching over class hierarchies unifies functional and object-oriented data access. It greatly simplifies the processing of XML trees.

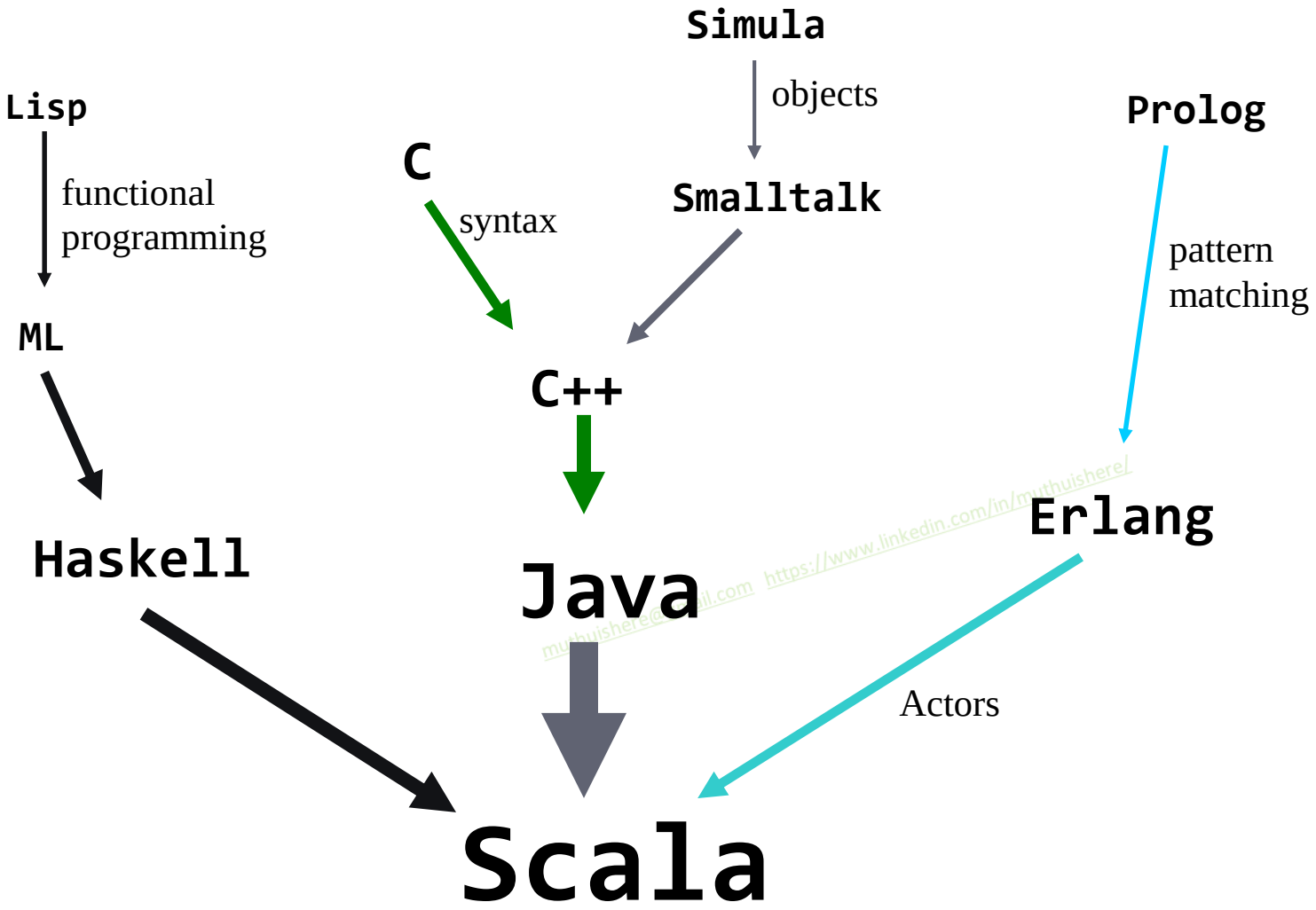
- A flexible syntax and type system enables the construction of advanced libraries and new domain specific languages.

At the same time, Scala is compatible with Java. Java libraries and frameworks can be used without glue code or additional declarations.

The current implementation of Scala runs on Java VM. It requires JDK 1.4 and can run on Windows, MacOS, Linux, Solaris, and most other operating systems. A .net version of Scala is currently under development.

For further information and downloads, please visit:

muthuishere@gmail.com http://www.linkedin.com/in/muthuishere/


Simula
objects
Lisp
functional
programming
C
syntax
Smalltalk
Prolog
pattern
matching
ML
C++
Haskell
Java
Erlang
Actors
Scala

Features of Scala

- Scala is compatible
- Scala is concise
- Scala is both functional and object-oriented
- Scala is statically typed
- Safe refactorings.
- `REPL`

Functional programming

- Functions are first class citizens
- Immutability
- Recursion
- Higher Order Functions

Main Method


Singleton Class
Command line parameters
function declaration
Return type , Unit Refers to void
object HelloWorld {
def main(args: Array[String]): Unit = {
print("Hello Scala")
}
}

Object

- An object is a class that has exactly one instance
- The definition of an object looks like a class, but uses the keyword object

Unit

- Use Unit return type to return nothing

print

```
println("Hello ,I am Print Statement")
println("I support","Multiple arguments printing","with comma")
```

- Print to Output
- println method can be used to print with new line

Main Method Other Ways

```
object Main extends App{

  println("Hello I am from App")

}
```

String Interpolation

- The Usual way of appending strings

```
val name = "Aegon"
println("Hello, " +  name)
```


- Using String interpolation prefix **s** and within double quotes use it with $

```
val name = "Aegon"
println(s"Hello, $name")
```

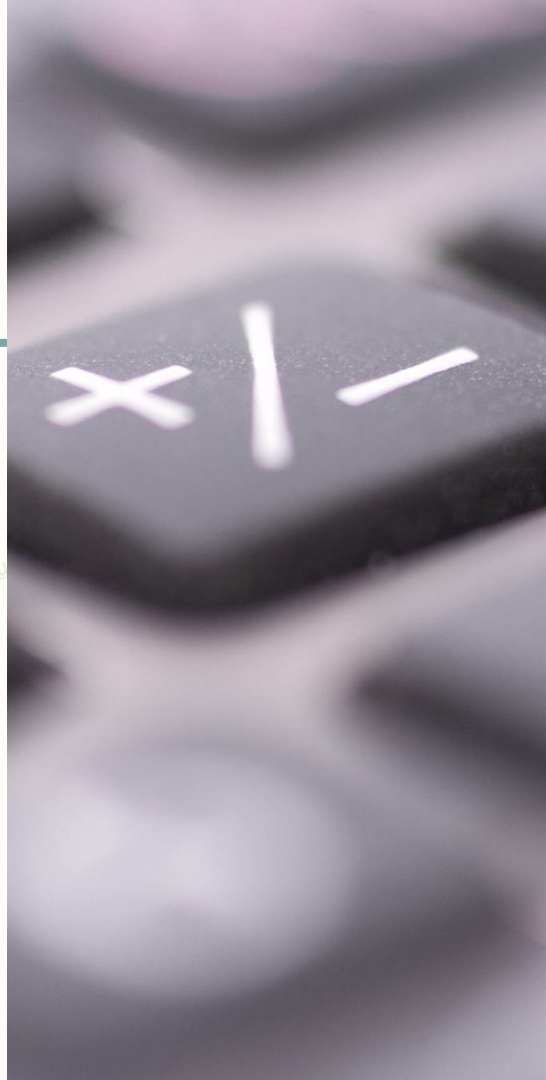
String Interpolation Exercise

- Will this work ???

```
println(s"21 + 23 = ${21 + 23}")
```

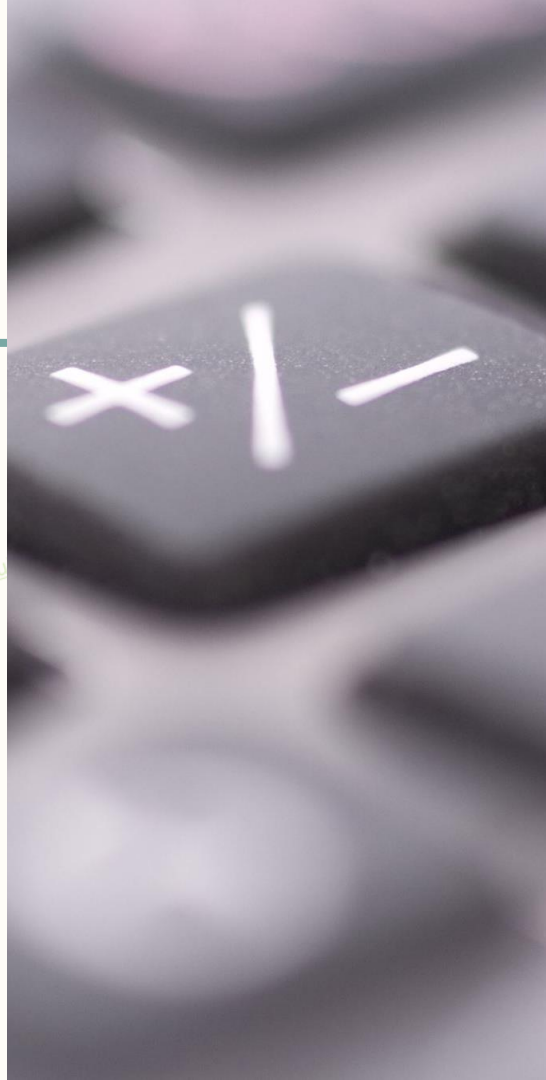
Operators

- **Addition(+)** operator adds two operands. For example, **x+y**.
- **Subtraction(-)** operator subtracts two operands. For example, **x-y**.
- **Multiplication(*)** operator multiplies two operands. For example, **x*y**.
- **Division(/)** operator divides the first operand by the second. For example, **x/y**.
- **Modulus(%)** operator returns the remainder when the first operand is divided by the second. For example, **x%y**.



Operators

- **Equal To(==)** operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false. For example, **5==5** will return true.
- **Not Equal To(!=)** operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise it returns false. It is the exact boolean complement of the **'=='** operator. For example, **5!=5** will return false.
- **Greater Than(>)** operator checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise it returns false. For example, **6>5** will return true.
- **Less than(<)** operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise it returns false. For example, **6<5** will return false.



Operators

- **Logical AND(&&)** operator returns true when both the conditions in consideration are satisfied. Otherwise it returns false. Using "*and*" is an alternate for && operator. For example, **a && b** returns true when both a and b are true (i.e. non-zero).
- **Logical OR(||)** operator returns true when one (or both) of the conditions in consideration is satisfied. Otherwise it returns false. Using "*or*" is an alternate for || operator. For example, **a || b** returns true if one of a or b is true (i.e. non-zero). Of course, it returns true when both a and b are true.
- **Logical NOT(!)** operator returns true the condition in consideration is not satisfied. Otherwise it returns false. Using "*not*" is an alternate for ! operator. For example, **!true** returns false.

Operators

- **Simple Assignment (=)** operator is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left. Can also use **(:=)**

Assignment Operators

> *Initially we had **colon-equals** for assignment—just as in Pascal, Modula, and Ada—and a single equals sign for equality. A lot of programming theorists would argue that that's the right way to do it. Assignment is not equality, and you should therefore use a different symbol for assignment. But then I tried it out with some people coming from Java. The reaction I got was, "Well, this looks like an interesting language. But why do you write **colon-equals**? What is it?" And I explained that its like that in Pascal. They said, "Now I understand, but I don't understand why you insist on doing that." Then I realized this is not something we wanted to insist on. We didn't want to say, "We have a better language because we write colon-equals instead of equals for assignment." It's a totally minor point, and people can get used to either approach. So we decided to not fight convention in these minor things, when there were other places where we did want to make a difference.*

Martin Odersky

Functions

```
def sumOfTwoNumbers(a:Int,b:Int):Int =
{
  return a + b
}
```

"def starts function definition

Function name

Parameter List in parenthesis

Equals sign

Function body between curly braces

Return statement

Functions

- Function definitions start with def.
- The function's name, is followed by a comma separated list of parameters in parentheses.
- A type annotation must follow every function parameter preceded by colon


```
def sumOfTwoNumbers(a:Int,b:Int):Int =
{
  return a + b
}
```

Function Declaration

- return is optional

```
def sumOfTwoNumbers(a:Int,b:Int):Int =
{
    a + b
}
```

Variables

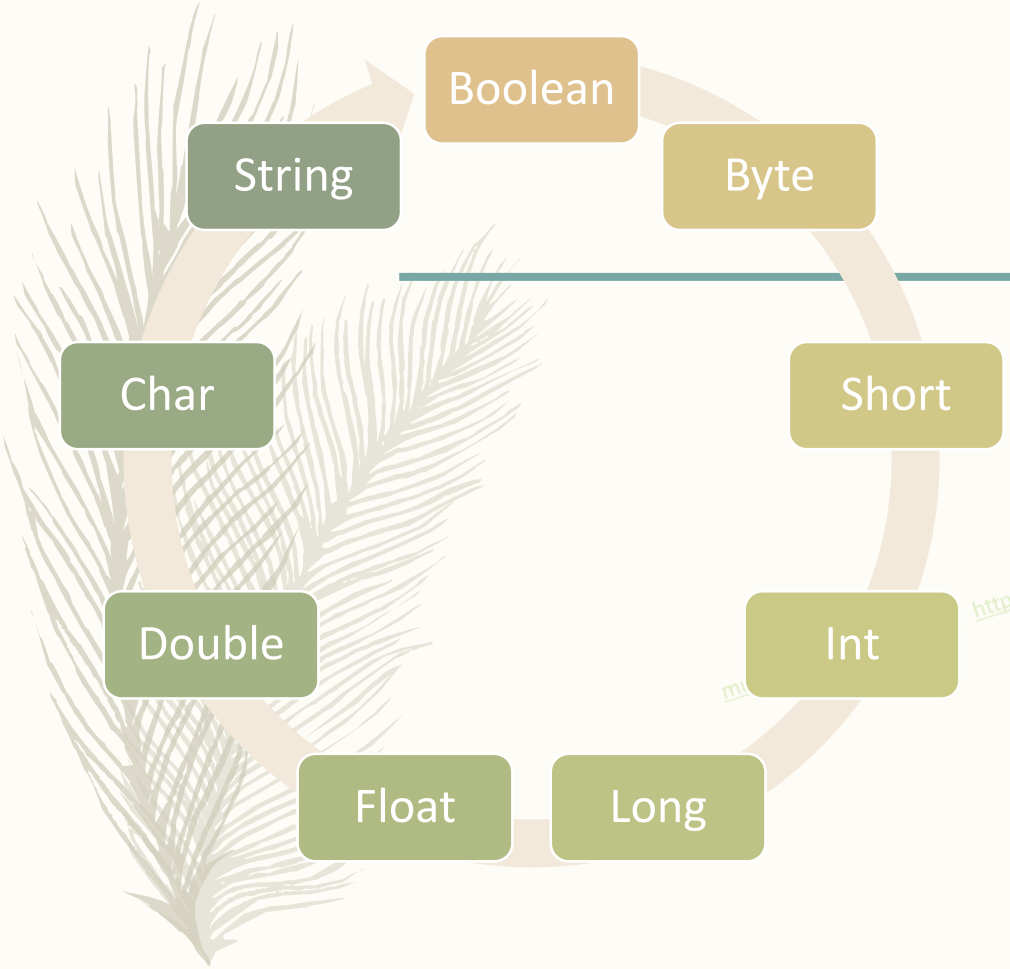
- Scala has two kinds of variables
  - val
  - var
- A val is like a final variable in Javlist.
- Once initialized, a val can never be reassigned.
- A var, by contrast, is like a non-final variable in Javlist.
- A var can be reassigned throughout its lifetime.

```
val firstName = "Pete"
val lastName:String = "Maverick"
```

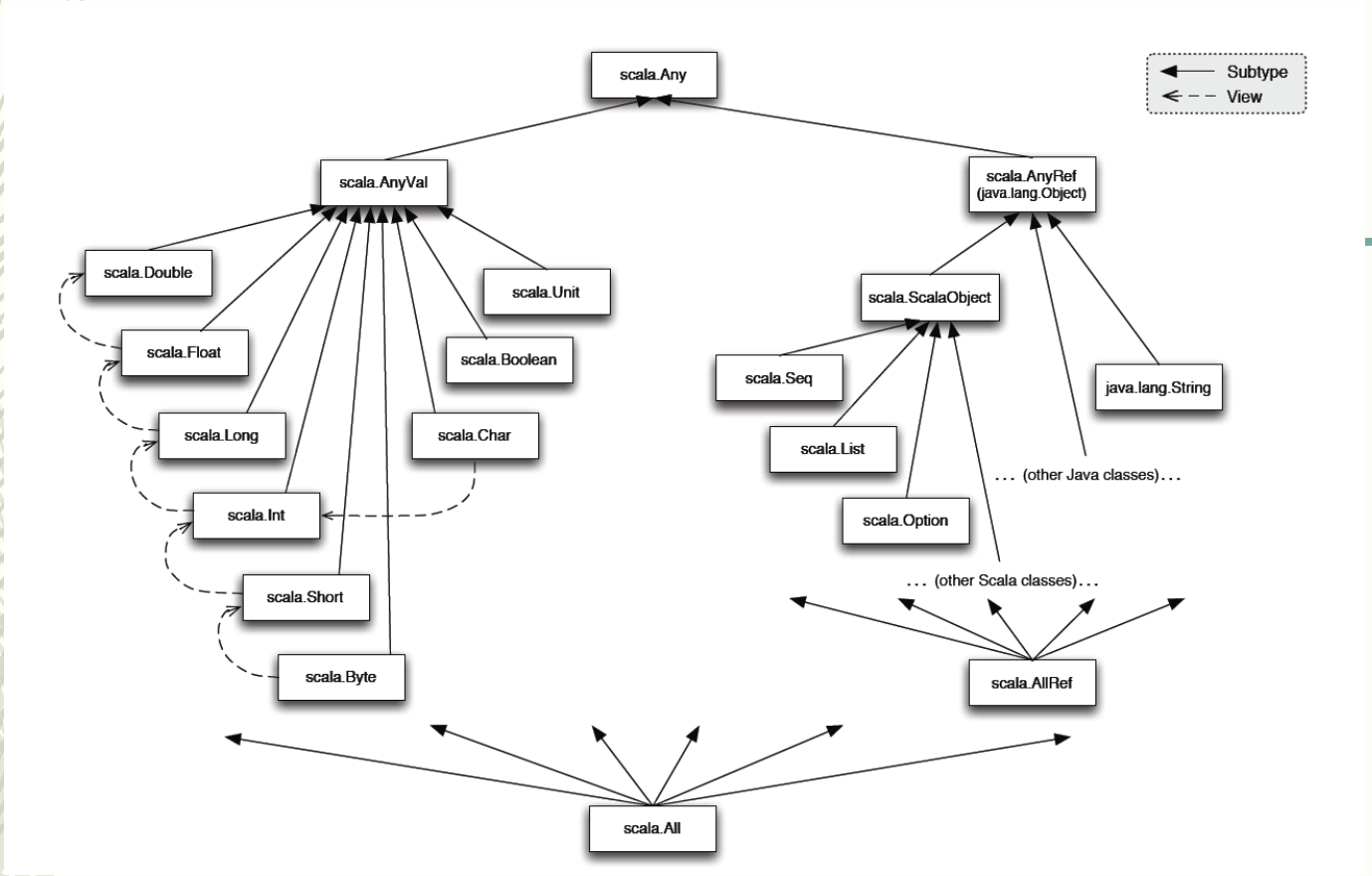
Immutability

- Why?
  - Immutable objects are automatically thread-safe

    (you don't have to worry about object being changed by another thread)
  - Compiler can reason better about immutable values -> optimization
  - Steve Jenson from Twitter: *"Start with immutability, then use mutability where you find appropriate."*



Variable - DataTypes


scala.Any
Subtype
View
scala.AnyVal
scala.AnyRef
(java.lang.Object)
scala.Double
scala.Unit
scala.ScalaObject
scala.Float
scala.Boolean
scala.Seq
java.lang.String
scala.Long
scala.Char
scala.List
... (other Java classes)...
scala.Int
scala.Option
... (other Scala classes)...
scala.Short
scala.Byte
scala.AllRef
scala.All

Variables – Lazy val

- Scala allows the special keyword lazy in front of val in order to change the val to one that is lazily initialized.

```
lazy val two: Int = 1 + 1
```

- we expect that the expression 1 + 1 is bound to two, but the expression is not yet evaluated. On the first (and only on the first) access of two from somewhere else, the stored expression 1 + 1 is evaluated and the result (2 in this case) is returned

```
scala> 5 * 6
res0: Int = 30

scala> val number = 30
number: Int = 30

scala> val incrementedNumber = number + 1
incrementedNumber: Int = 31

scala> number = number + 1
               ^
       error: reassignment to val
```

Scala REPL

Statically Typed

- Scala is statically typed
  - includes a local type inference system

```
scala> def add(a:Int, b:Int) = a + b
add: (a: Int, b: Int)Int

scala> val m = add(1, 2)
m: Int = 3

scala> println(m)
3

scala>
```

We have NOT specified the return type of the function or the type of the variable

Type Inference

```
def add(a, b) = a + b

val m = add(1, 2)
println(m)
```

This Will not Work

Type Inference

Concatenate

```
print("Hello " + "World" )
```

- Use +

Array

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

  print("Hello " + args(0))
}
```

- Arrays are zero based
- Access an element by specifying an index in parentheses.
- The first element in a Scala array named args is args(0), not args[0], as in Java

Array

```
val weekDays = Array("Sunday", "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday",
  "Saturday" )

println("Days are : ", weekDays)
```

Read Input from Console

- Use readLine method from class scallist.io.StdIn


```
print("Enter Input : ")
val input = scala.io.StdIn.readLine()
```

If

- If followed by a Boolean expression


```
if(input >= 18)
  println("You are Eligible to vote")
else
  println("Wait till you are 18")
```

If - expression

```
if (age >= 18) "Eligible" else "Not Allowed"
```

```
val result = if (age >= 18) "Eligible" else "Not Allowed"
```

Pattern Matching

- Pattern matching is a mechanism for checking a value against a pattern.
- A successful match can also deconstruct a value into its constituent parts.
- It is a more powerful version of the switch statement in Java and it can likewise be used in place of a series of if/else statements.

Pattern Matching

```
val x: Int = Random.nextInt(10)

val res = x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}

print(res)
```

Pattern Matching

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

}

print(matchTest(5))
```

Underscore

- It’s similar to Joker in cards
  - Its for everything

  - `Give me a variable name`
    - *`But I don’t care of what it is`*

Underscore

- Replacement of * while importing all

```
import  java.util._
```

Underscore

- Initialize with default values

```
//Error Member variables
not Initialized
class Product {

  var name:String

  var price:Float
}
```



```
class Product {

  var name:String=_
  var price:Float=_
}
```

Underscore

- Pattern Matching


```
List(1, "data", 3,8.9).foreach {
  case elem: String => println("String value is  $elem")
  case elem: Int => println("Integer Value is $elem")
  case _ => println("something else " )

}
```

Loop

```
for(arg<- args){
  println("The Argument is "  + arg)
}
```

- for
  - The parentheses after the "for" contain a variable name followed by symbol
  - The symbol <- denotes the current element is assigned
  - The variable assigned is always immutable (val not var)

Loop

```
for(arg<- args){
  println("The Argument is "  + arg)
}
```

- Run in terminal as

```
scala-programs>scala ForLoop.scala aegon danny tyrion
The Argument is aegon
The Argument is danny
The Argument is tyrion
```

Loop

```
for( i <- 1 until  10){
  println( "until Value of i: " + i );

}
```

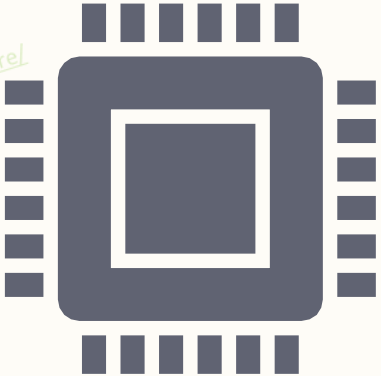


```
for( i <- 1 to 10){
  println( "to Value of i: " + i );

}
```

```
for{
  i <- 1 to  10
  j <- 1 to  10
} {

  println( " Value of i: " + i +" j" + j );

}
```

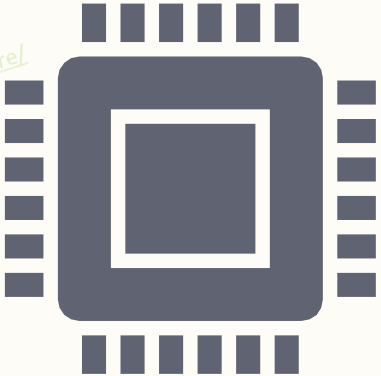
Exercise

- Create a Scala Program SummingArguments.scala to print sum of all arguments received from command line
- Input
  - SummingArguments.scala 12 24
- Should print
  - 36
- Input
  - SummingArguments.scala 1 9 56 23
- Should print
  - 89



Exercise

- Create a Scala Program to print Squares of numbers till 20



Class

- Java

```
class MyClass {
private int index;
private String name;

public MyClass(int index, String name) {
this.index = index;
this.name = name;
}
}
```

- Scala

```
class MyClass(index: Int, name: String)
```

Class

- Similar to class in Java

```
class Person {
  var name = ""
  var age = 0
  var eligibleVoter=false
}
```



```
val person = new Person()
person.name="Aegon"
person.age=29
person.eligibleVoter=true

// Print the object's age and name properties
println(person.age)
println(person.name)
```

Class- Parameterized

```
class Person(var name: String,  var age:Int, var eligibleVoter:Boolean)
```



```
val person = new Person("Aegon", 29, true)
```

Class-Parameterized

- Parameters without val or var are private values, visible only within the class.

```
class Person(name: String, age:Int, eligibleVoter:Boolean)
```

Class- Secondary constructor

```
class Person(var name: String, var age: Int, var eligibleVoter: Boolean) {

  def this() {
    this("", 0, false)
    println("Zero-argument secondary constructor")
  }
}
```

```
val person = new Person("Aegon", 29, true)

val anotherPerson =  new Person()
```

Class- Encapsulation

```
class Person(var name: String,  var age:Int){
  def isEligibleVoter ():Boolean ={

    if(age >= 18)
      true
    else
      false
  }
}
```

```
val person = new Person("Aegon", 9)

// Print the object's age and name properties

println("Name=>" + person.name,"age=>"+person.age,"eligible for voting=>"+ person.isEligibleVoter)
```

Class- Private members & init functions

```
class Person(var _name: String, private var _age:Int) {


  private var _eligibleVoter = false
  var name = ""
  // Getter
  def age = _age
  def eligibleVoter = _eligibleVoter


  // Setter
  def age_= (value:Int):Unit = {

    if(value >= 18)
      _eligibleVoter=true

    _age = value
  }

  //Init Functions
  println("Hello from default constructor")
  age_=(_age)

}
```


```
val person = new Person("Aegoon",12);
println("Name" ,person.name,"age",person.age,"eligible for voting",person.eligibleVoter)
```

Class - Inheritance

- Use extends , Similar to Java

```
class Vehicle(var name: String) {

  def getTyres: Int = {
    4
  }
}

class BigLorry(name: String) extends Vehicle (name) {

  override def  getTyres:Int= { 8  }
}
```

Abstract Class

- Similar to abstract class in Java

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

class Dog (name: String) extends Pet (name) {
  def speak { println(s"Hi , My Name is $name ,woof!!!") }

}
```

Class –Optional Parameters

- Constructors can have optional parameters by providing a default value


```
class Person(var name: String="Anonymous",  var age:Int =0)
```

null

– “I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

--Tony Hoare

null in Scala

- In Java, any method that is *supposed* to return an object *could* return null
  - Here are your options:
    - *Always check for null*
    - *Always put your method calls inside a try...catch*
    - *Make sure the method can't possibly return null*
- Yes, Scala has null--but only so that it can talk to Java
- In Scala, if a method *could* return "nothing," write it to return an Option object, or check with option Object
  - *Some is a method to create Option of Object*


```
val name:String=null
val result = Option(name).getOrElse("Anonymous")
println(result);
```

```
def getName():Option[String] = {
  return Some("User");
}
```

Seq

- A Seq is an Iterable that has a defined order of elements. Sequences provide a method apply() for indexing, ranging from 0 up to the length of the sequence. Seq has many subclasses including Queue, Range, List, Stack, and LinkedList.

```
val seq:Seq[Int] = Seq(21,11,81,21,1,82)

seq.foreach((element:Int) => print(element+" "))

println("\nAccessing element with index")


println(seq(2))
```

Lists

- A List is a Seq that is implemented as an immutable linked list. It's best used in cases with last-in first-out (LIFO) access patterns.


```
val weekDays = List("Sunday", "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday",
  "Saturday" )

println("Days are : ")

for ( currentDay <-weekDays )
{
  println(currentDay )
}
```

Lists

- Using cons Operator :: should end with Nil

```
val weekDays = "Sunday":: "Monday":: "Tuesday"::"Wednesday"
            :: "Thursday":: "Friday"::"Saturday":: Nil
```

- The first element of a list is its head, the rest is its tail.

Lists

- Uniform List can be created in Scala using List.fill() method. List.fill() method creates a list and fills it with zero or more copies of an element.

```
val listWithDefaultValues =List.fill(3)("Default Element")
```

List

- list.length is the length (number of elements) of the sequence;
- list.isEmpty is the same as list.length == 0;
- list.nonEmpty is the same as list.length != 0;
- list.last the last element of the sequence

- list `contains x` tests if the sequence contains an element equal to `x`;

List

- list.sum sum of all the elements of the list.;

ListBuffer

- The ListBuffer object is convenient when we want to build a list from front to back. It supports efficient prepend , append and delete operations

```
val list =ListBuffer(1,2,3,4,5)

//Append 6
list += 6;

//Prepend 0
0 +=: list;
```

ListBuffer

- Delete

```
val list:ListBuffer[Int] =
ListBuffer(1,2,3,4,5)

//To remove element 3
list -= 3;



//To remove element at index 0
list.remove(0)
```

HashMap

- HashMap is a part of Scala Collection's.
- It is used to store element and return a map. A HashMap is a combination of key and value pairs

```
val hashMap = HashMap("A"->"Apple","B"->"Ball","C"->"Cat")
//Adding entry
hashMap += ("D"->"Doctor")

//removing entry
hashMap -= "A"

//Iterate
hashMap.foreach {
  case (key, value) => println (key + " -> " + value)        // Iterating
elements
}
```

Multi Line Strings

- Create multiline strings by surrounding your text with three double quotes


```
val res=  """
    Scala Supports Multi line strings
    I am writing this to confirm

  """
```

File I/O

- We don’t have a class to write a file, in the Scala standard library
- We should import java.io.File and java.io.PrintWriter

- Write File


```
//fileName refer to full path of file
 val writer=new PrintWriter(new File(fileName))
 writer.write("This is a File Writing demo in Scala")
 writer.close()
```

File I/O

- Read File as String

```
//fileName refer to full path of file
val contents = Source.fromFile(fileName).mkString;
```

yield

- Does for loop return value???

yield

- Does for loop return value???

- For each iteration of your for loop, yield generates a value which will be remembered. It's like the for loop has a buffer you can't see, and for each iteration of your for loop another item is added to that buffer. When your for loop finishes running, it will return this collection of all the yielded values.

Seq

- A Seq is an Iterable that has a defined order of elements. Sequences provide a method apply() for indexing, ranging from 0 up to the length of the sequence. Seq has many subclasses including Queue, Range, List, Stack, and LinkedList.

```
val seq:Seq[Int] = Seq(21,11,81,21,1,82)

seq.foreach((element:Int) => print(element+" "))

println("\nAccessing element with index")


println(seq(2))
```

Lists

- A List is implemented as an immutable linked list. It's best used in cases with last-in first-out (LIFO) access patterns.


```
val weekDays = List("Sunday", "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday",
  "Saturday" )

println("Days are : ")

for ( currentDay <-weekDays )
{
  println(currentDay )
}
```

Lists

- Using cons Operator :: should end with Nil

```
val weekDays = "Sunday":: "Monday":: "Tuesday"::"Wednesday"
            :: "Thursday":: "Friday"::"Saturday":: Nil
```

- The first element of a list is its head, the rest is its tail.

Lists

- Uniform List can be created in Scala using List.fill() method. List.fill() method creates a list and fills it with zero or more copies of an element.

```
val listWithDefaultValues =List.fill(3)("Default Element")
```

List

- list.length is the length (number of elements) of the sequence;
- list.isEmpty is the same as list.length == 0;
- list.nonEmpty is the same as list.length != 0;
- list.last the last element of the sequence

- list `contains x` tests if the sequence contains an element equal to `x`;

List

- list.sum sum of all the elements of the list.;

ListBuffer

- The ListBuffer object is convenient when we want to build a list from front to back. It supports efficient prepend , append and delete operations

```
val list =ListBuffer(1,2,3,4,5)

//Append 6
list += 6;

//Prepend 0
0 +=: list;
```

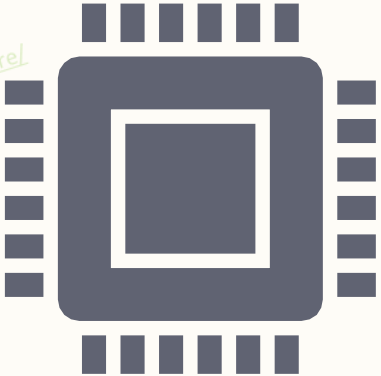
ListBuffer

- Delete

```
val list:ListBuffer[Int] =
ListBuffer(1,2,3,4,5)

//To remove element 3
list -= 3;



//To remove element at index 0
list.remove(0)
```

Exercise

- val weekDays = ("Monday", "Tuesday",
- "Wednesday")
-
- Add to end
  - Thursday
  - Friday
  - Saturday
  - someOtherDay
- Add Sunday at front
- remove someOtherDay



HashMap

- HashMap is a part of Scala Collection's.
- It is used to store element and return a map. A HashMap is a combination of key and value pairs

```
val hashMap = HashMap("A"->"Apple","B"->"Ball","C"->"Cat")
//Adding entry
hashMap += ("D"->"Doctor")

//removing entry
hashMap -= "A"

//Iterate
hashMap.foreach {
  case (key, value) => println (key + " -> " + value)        // Iterating
elements
}
```

Multi Line Strings

- Create multiline strings by surrounding your text with three double quotes


```
val res=  """
    Scala Supports Multi line strings
    I am writing this to confirm

  """
```

File I/O

- We don’t have a class to write a file, in the Scala standard library
- We should import java.io.File and java.io.PrintWriter

- Write File


```
//fileName refer to full path of file
 val writer=new PrintWriter(new File(fileName))
 writer.write("This is a File Writing demo in Scala")
 writer.close()
```

File I/O

- Read File as String

```
//fileName refer to full path of file
val contents = Source.fromFile(fileName).mkString;
```

map()

- Map applies a function on all elements of a sequence.

```
val numbers = List(1,3,5,9)

val squareOfNumbers = numbers.map(number =>
number * number)
```

filter()

- Filter selects a set of values from a sequence based on the boolean value returned by a function passed as its parameter


```
val numbers = List(1,3,5,9)

val numbersGreaterThanFive = numbers.filter(number => number > 5)
```

reduce()

- Reduce combines the elements of a sequence into a single element.
- The method takes two parameter , One is the accumulator and other is current value
-


```
val numbers = List(1,3,5,9)

val sumOfNumbers = numbers.reduce((accumulator,currentNumber) => accumulator+currentNumber)
```

Functional Programming

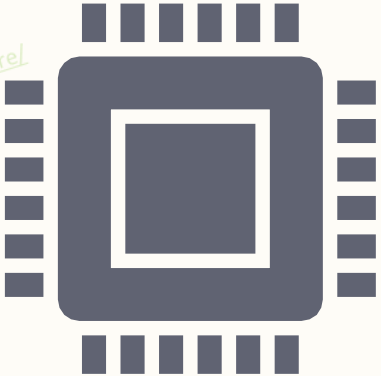
```
def isOdd(x: Int) = (x % 2) != 0

val numbers = List(1,2,3,4,5,6,7)

println("Numbers", numbers)

// are all numbers Odd?
println("Is all Numbers Odd",numbers.forall(isOdd))
```

Functional Programming

```
def isOdd(x: Int) = (x % 2) != 0

val numbers = List(1,2,3,4,5,6,7)

// is there an Odd element in numbers?

println("Is Odd Number exists",numbers.exists(isOdd))

//partition into two sublists:  odd & Even
println("Odd Number List & even Number list ",numbers.partition(isOdd))
```

Exercise

- Create class person with fields name & age
-
- Add to end
  - Thursday
  - Friday
  - Saturday
  - someOtherDay
- Add Sunday at front
- remove someOtherDay



Functional Programming

```
def isLesserThanFive(x: Int) = x < 5

val numbers = List(1,2,3,4,5,6,7)

println("Numbers", numbers)
```

fold()

- fold combines the elements of a sequence into a single element from an initialized.
- The method takes one parameter , followed by a function with two parameter , One is the accumulator and other is current value
-


```
val numbers = List(21, 10, 12, 81)
numbers.fold(2)((accumulator,currentNumber) => accumulator+currentNumber)
```

traits

- Scala presents a half-way point between full multiple inheritance and Java's single inheritance with interfaces model by giving you traits.
- Traits give you the ability to recreate interfaces as Java does but also allows you to go further.
- 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:

Traits – use like abstract class

```
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.") }
}
```

Traits – extend trait with another trait

```
trait Player {
  def play
  def close
  def pause
  def stop
  def resume
}


trait AudioPlayer extends Player {
  def setVolume(volume: Double):Unit = { print(s"setting volume to $volume")}
}
trait VideoPlayer extends AudioPlayer {
  def increaseBrightness
}
```

trait - mixins

- To implement mixin, define the methods in trait,
- Then add the trait to class using extends or with

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

abstract class Pet (var name: String) {
  def speak  // abstract
  def walk { println("Walking") }
}
class Dog (name: String) extends Pet (name) with Tail {
  def speak { println(s"Hi , My Name is $name ,woof!!!") }

}
```

case class

- case classes are just regular classes which are immutable by default
- It does not use new keyword to instantiate object.
- All the parameters listed in the case class are public and immutable by default. (No re assigning)

case class - comparision

```
case class Person(id: String)
```

```
//Will return true , since it uses case class
val isSame = Person("21") == Person("21")
```

case class - immutability

```
case class Person(id: String)
```

```
var personX =  Person("21")
personX.id="45";// This Line Wont compile
```

case class – pattern matching

```
abstract class Notification

case class Email(sender: String, title: String, body: String) extends Notification

case class SMS(caller: String, message: String) extends Notification

case class VoiceRecording(contactName: String, link: String) extends Notification
```

case class – pattern matching

```
def showNotification(notification: Notification): String = {
  notification match {
    case Email(sender, title, _) =>
      s"You got an email from $sender with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"You received a Voice Recording from $name! Click the link to hear it:
$link"
  }
}
```

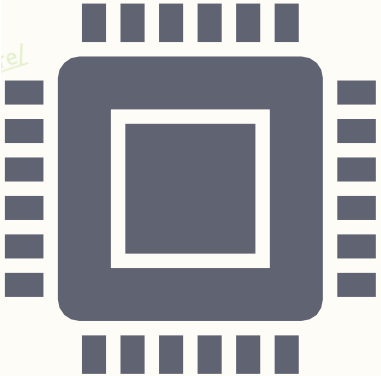
https://github.com/muthuishere/scala-workshop/blob/master/data/VehicleService.scala

- Create a class Vehicle

```
class Vehicle
(sNo:Int,name:String,manufacturer:String,year:Int,
fuelType:String,transmission:String)
```

- Use the mock data available from **https://github.com/muthuishere/scala-workshop/blob/master/data/VehicleService.scala**

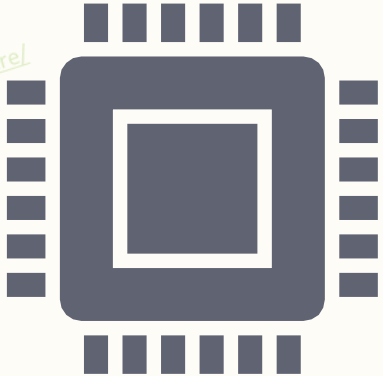
- Perform the following Operations through FP
  - Find all diesel vehicles
  - Find the total count of vehicles by Maruti
  - Find the vehicles manufactured in 2010 by Volvo
  - Split the vehicles based on Maruti and Others



SBT

Exercise

- Create a class Calculator
  - Create a method add , accepts two arguments and returns the addition
  - Call the add method from main for arguments 56 & 74



What is Sbt

- sbt is a build tool for Scala, Java
- Similar to Java's Maven and Ant.
- Compile and run code with SBT
- Add managed dependencies to your project
- Run tests

Sbt - Prerequisties

- Install SBT
- Set the SBT bin path to environment variable path

Sbt

- To move an existing application to sbt
- Create a folder structure like shown in right
- Use the script https://github.com/muthuishere/scala-workshop/blob/master/init-for-sbt.sh

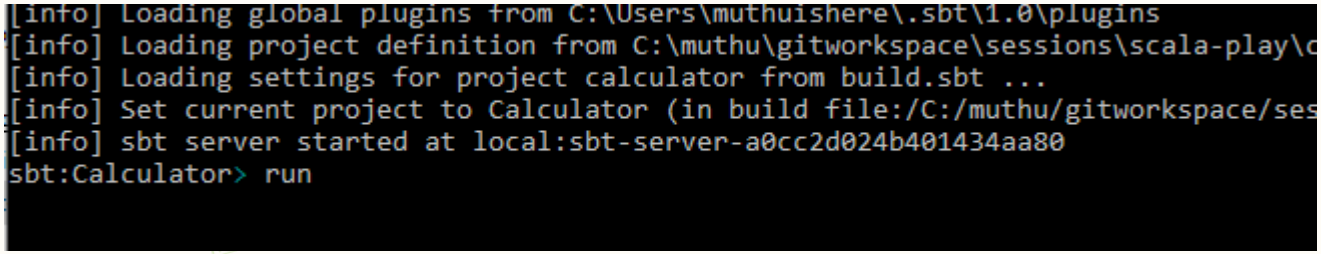
```
lib/
src/
        main/
                resources/
                scala/
                java/
        test/
                resources/
                scala/
                java/
```

Sbt

- Create a file build.sbt on the root folder
- Add the following in build.sbt file
  - name := “Calculator project”
  - version := "1.0“
- Move the calculator.scala to **\src\main\scala\Calculator.scala**
- Now Open the command prompt on the root folder and type sbt

Sbt

- After command prompt is shown type run



Sbt

- Sbt has various default tasks built in
  - `clear`
  - `run`
  - `test`
  - `etc....`

Sbt – Manage dependency

- Add your dependency to libraryDependencies variable

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.8" % "test"
```

Sbt – Import to intellij

- Open project root directory in intellij , it will automatically identify sbt based project and it will open in that view

Sbt – Create Custom Task

```
lazy val runStaticCodeAnalysis = taskKey[Unit](“Will
Run ‘Static'")

runStaticCodeAnalysis := {
  println(“connect to Static Code Analysis Server and
run ")
}
```

Unit Testing

Every developer would have done this.

- Have debugging sessions (with the help of great IDE),
- Add lots of log messages
- Click through the user interface of your application,
- Perform frequent code reviews.

“

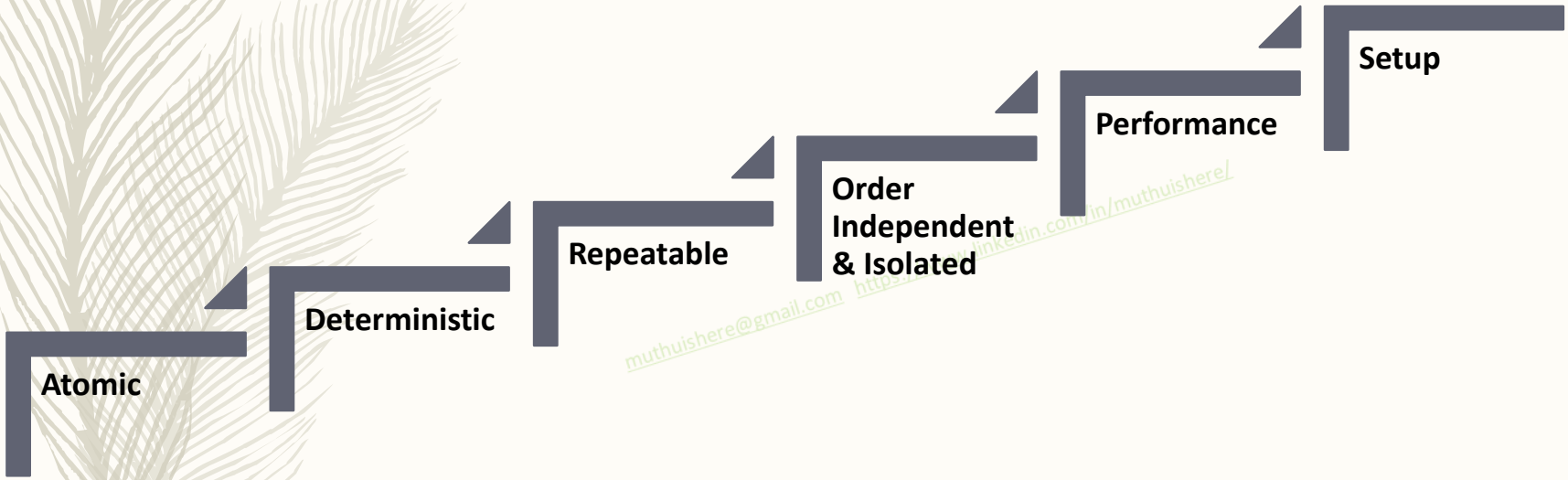
UNIT TESTING is a level of software testing where individual units/ components of a software are tested.

”

The purpose is to validate that each unit of the software performs as designed

A unit is the smallest testable part of any software

Good Unit Test



A test is not a unit test if:

- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Unit Testing In Scala

- For every class in src/main/scala
  - write a test class in src/test/scala
-
- Test classes ~ Test suites
- Methods ~ Tests

- Run the suite and see the output

>ScalaTest™

simply productive

ScalaTest is designed to increase your team's productivity through simple, clear tests and executable specifications that improve both code and communication.

- Uses BDD style
- Integrates with
  - Sbt

Write Scala Test

- Add the following dependency in build.sbt

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.8" % "test"
```

- Create a file CalculatorTest.scala in src\test\scala folder

Write Scala Test

- For teams coming from xUnit, FunSuite feels comfortable and familiar while still giving some of the benefits of BDD:
- FunSuite makes it easy to write descriptive test names, natural to write focused tests, and generates specification-like output
- Just extend the class with FunSuite and implement test methods

```
import org.scalatest.FunSuite

class CalculatorTest extends FunSuite {
  test("Calculator.add") {

    val calculator = new Calculator
    assert(calculator.add(2,3) === 5)
  }
}
```

Write Scala Test

- Every test should have a description in parameter
- Assert for verifying
- Note the triple equals

```
class CalculatorTest extends FunSuite {
  test("Calculator.add") {
    val calculator = new Calculator
    assert(calculator.add(2,3) === 5)
  }
}
```

running Scala Test

- Open Sbt console
- Type test

```
[info] sbt server started at local:sbt-server-a0cc2d024b401434aa80
sbt:Calculator> test
[info] Compiling 1 Scala source to C:\muthu\gitworkspace\sessions\scala-play\code\scala\calculator\target\scala-2.12\cla
[info] Compiling 1 Scala source to C:\muthu\gitworkspace\sessions\scala-play\code\scala\calculator\target\scala-2.12\tes
[info] CalculatorTest:
[info] - Calculator.add
[info] Run completed in 298 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 4 s, completed Dec 30, 2019 12:02:32 AM
sbt:Calculator>
```

Write Scala Test - Triple Equals

- ScalaTest lets you use Scala's assertion syntax but defines a triple equals operator (===) to give better error messages.

- The following code would give an error indicating only that an assertion failed:

```
assert(1 == 2)
```

- Using triple equals instead would give more informative error message, "1 did not equal 2":

```
assert(1 === 2)
```

Concurrency

Future

- A Future is an object holding a value which may become available at some point. This value is usually the result of some other computation:
  - If the computation has not yet completed, we say that the Future is not completed.
  - If the computation has completed with a value or with an exception, we say that the Future is completed.
- Completion can take one of two forms:
  - When a Future is completed with a value, we say that the future was successfully completed with that value.
  - When a Future is completed with an exception thrown by the computation, we say that the Future was failed with that exception.

Future

- A Future has an important property that it may only be assigned once. Once a Future object is given a value or an exception, it becomes in effect immutable– it can never be overwritten.

- The simplest way to create a future object is to invoke the Future.apply method which starts an asynchronous computation and returns a future holding the result of that computation. The result becomes available once the future completes..

Future

```
// 1 - the imports
import scala.concurrent.{Await, Future}
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

object AddingFuture extends App {
```

Future

```
  // 2 - create a Future

  val addOneAndOne = Future {
    Thread.sleep(500)
    1 + 1
  }

  // 3 - this is blocking (blocking is bad)
  val result = Await.result(addOneAndOne, 1 second)
  println(result)
  Thread.sleep(1000)

}
```

Future

- **Run one thing, but don’t block, use callback**
- A better approach to working with a future is to use its callback methods. There are three callback methods:
  - onComplete
  - onSuccess,
  - onFailure.

Future

```
def sleep(time: Long) { Thread.sleep(time) }
println("starting calculation ...")
val firstEmployeeId = Future {
  sleep(Random.nextInt(500))
  42
}
println("before onComplete")
firstEmployeeId.onComplete {

  case Success(value) => println(s"Got the callback, meaning = $value")
  case Failure(e) => e.printStackTrace
}
// do the rest of your work
println("Some work ..."); sleep(100)
```

Reactive Manifesto

- **Responsive**: The system responds in a timely manner if at all possible.
- **Resilient**: The system stays responsive in the face of failure.
- **Elastic:** The system stays responsive under varying workload
- **Message Driven**: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

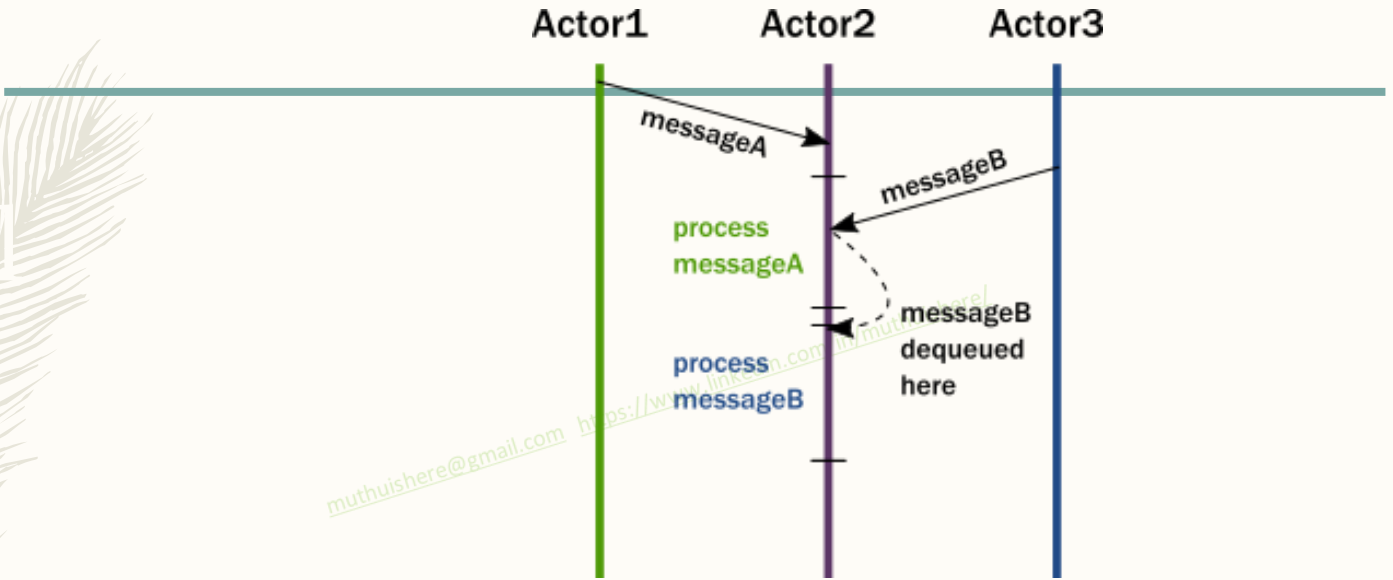
Writing Distributed Systems

- Writing concurrent programs is hard.
  - Having to deal with threads, locks, race conditions, and so on is highly error-prone and can lead to code that is difficult to read, test, and maintain..
- Many therefore prefer to avoid multithreading altogether.
  - Instead, they employ single-threaded processes exclusively, relying on external services (such as databases, queues, etc.)

Akka Framework

Akka Framework

- Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant applications on the JVM.
- Akka is written in Scala, with language bindings provided for both Scala and Java.
- Akka's approach to handling concurrency is based on the Actor Model.


Actor1
Actor2
Actor3
messageA
messageB
process
messageA
messageB
dequeued
here
process
messageB

Akka - Actor Model

In summary, this is what happens when an actor receives a message:

The actor adds the message to the end of a queue.

If the actor was not scheduled for execution, it is marked as ready to execute.

A (hidden) scheduler entity takes the actor and starts executing it.

Actor picks the message from the front of the queue.

Actor modifies internal state, sends messages to other actors.

The actor is unscheduled.

Akka- Actor Model

- To Accomplish this
  - A mailbox (the queue where messages end up).
  - A behavior (the state of the actor, internal variables etc.).
  - Messages (pieces of data representing a signal, similar to method calls and their parameters).
  - An execution environment (the machinery that takes actors that have messages to react to and invokes their message handling code).
  - An address (more on this later).

Akka- Actor Model

- Actors handle error situations gracefully ,as there is no shared call stack
  - The first case is when the delegated task on the target actor failed due to an error in the task (typically some validation issue, like a non-existent user ID). The sender should be notified immediately
  - The second case is when a service itself encounters an internal fault.
    - *Akka enforces that all actors are organized into a tree-like hierarchy*
    - *An actor that creates another actor becomes the parent of that new actor. This is very similar how operating systems organize processes into a tree. Just like with processes, when an actor fails, its parent actor can decide how to react to the failure*

Akka- Hello Mesage

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloActor extends Actor {

  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}
```

Akka- Hello Mesage

```
object Main extends App {
  val system = ActorSystem("HelloSystem")
  // default Actor constructor
  val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")

  helloActor ! "hello"

  helloActor ! "Whatszzzz Uppp"

}
```

Akka- Hello Mesage

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloActor extends Actor {

  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}
```

Akka- Auto Engine

```
case object StartMessage
case object StartEngineMessage
case object StopEngineMessage
case object StopSpeedoMeterMessage
case object DriveMessage
case object IncreaseGearMessage
case object DecreaseGearMessage
case object UnableToChangeGearMessage
case object ChangedGearMessage
case object StopMessage
case object UpdateSpeedoMeterMessage{
  var speed:Int=0
}
```

Akka- Auto Engine

```
class SpeedoMeter extends Actor {
  def receive = {
    case UpdateSpeedoMeterMessage =>
      println("  updating Speedometer with " + UpdateSpeedoMeterMessage.speed)
    case StopMessage =>
      println("Stopping  Speedometer")
     // context.stop(self)
  }
}
```

Akka- Auto Engine

```
class Engine(speedoMeter: ActorRef) extends Actor {
  var currentGear = 0




  def IncreaseGear { currentGear += 1; println(s"Current Gear => $currentGear" ) }
  def DecreaseGear { currentGear -= 1; println(s"Current Gear => $currentGear" ) }


  def receive = {
    case StartEngineMessage =>
       println(Thread.currentThread().getName() +"Starting Engine ")
      currentGear=0
```

Akka- Auto Engine

```
case StopEngineMessage =>
      speedoMeter!StopSpeedoMeterMessage
       println("Stopping  Engine")
    case DriveMessage =>
      UpdateSpeedoMeterMessage.speed=currentGear *30
      speedoMeter!UpdateSpeedoMeterMessage


    case DecreaseGearMessage =>

      if (currentGear < 0) {
        sender ! UnableToChangeGearMessage

      } else {
        DecreaseGear
        sender ! ChangedGearMessage
      }
```

Akka- Auto Engine

```
    case IncreaseGearMessage =>

      if (currentGear > 4) {
        sender ! UnableToChangeGearMessage
      } else {
        IncreaseGear
        sender ! ChangedGearMessage
      }
  }
}
```

Akka- Auto Engine

```
class Administrator(engine: ActorRef)  extends Actor {
  def receive = {
    case StartMessage =>
      println(Thread.currentThread().getName() +"Starting ")
      engine ! StartEngineMessage
      engine ! IncreaseGearMessage
    case DriveMessage =>
        engine ! DriveMessage
    case UnableToChangeGearMessage =>
      println( Thread.currentThread().getName() + "  UnableToChangeGearMessage")


    case ChangedGearMessage =>
      println(Thread.currentThread().getName() +"  ChangedGearMessage")

    case StopMessage =>
      engine ! StopMessage
      println("Stopping Engine")

  }
}
```

Akka- Auto Engine

```
object AutoDriveCar extends App {
  val system = ActorSystem("AutoDriveVehicle")

  val speedoMeter = system.actorOf(Props[SpeedoMeter], name = "speedoMeter")
  val engine = system.actorOf(Props(new Engine(speedoMeter)), name = "engine")
  val administrator = system.actorOf(Props(new Administrator(engine)), name = "administrator")
  // start them going
  administrator ! StartMessage

  administrator !DriveMessage

}
```

Play Framework

- Based on a lightweight, stateless and web-friendly architecture
- High scalability
- Develop in Scala and/or Java
- Developer friendly
- Focus on productivity
  - Hit refresh workflow
  - Type safety
  - Open Source

Who Uses Play Framework

Play Framework

- Model (Java and/or Scala)
- Controller (Java and/or Scala)
- View (templates, HTML and Scala)
- Routes
  - Mapping url paths and Controllers

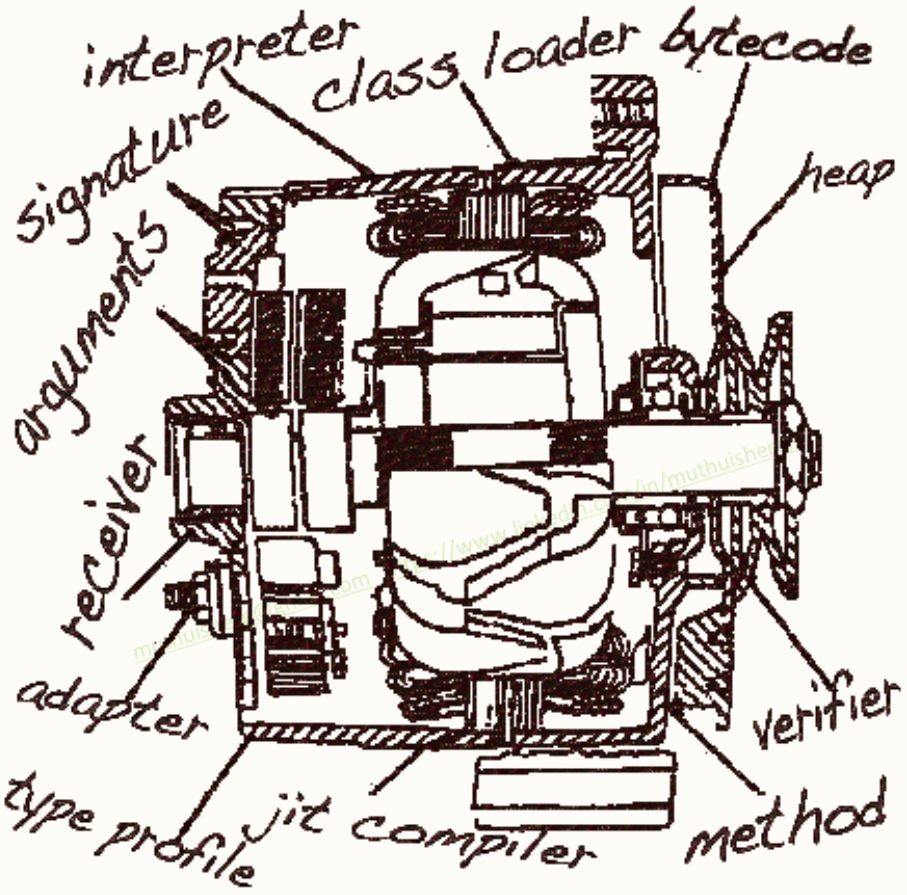
Play Framework

- Majority of code gets compiled including:
- - Templates
  - - Assets (Javascript, CSS)
- Detect errors as early as possible

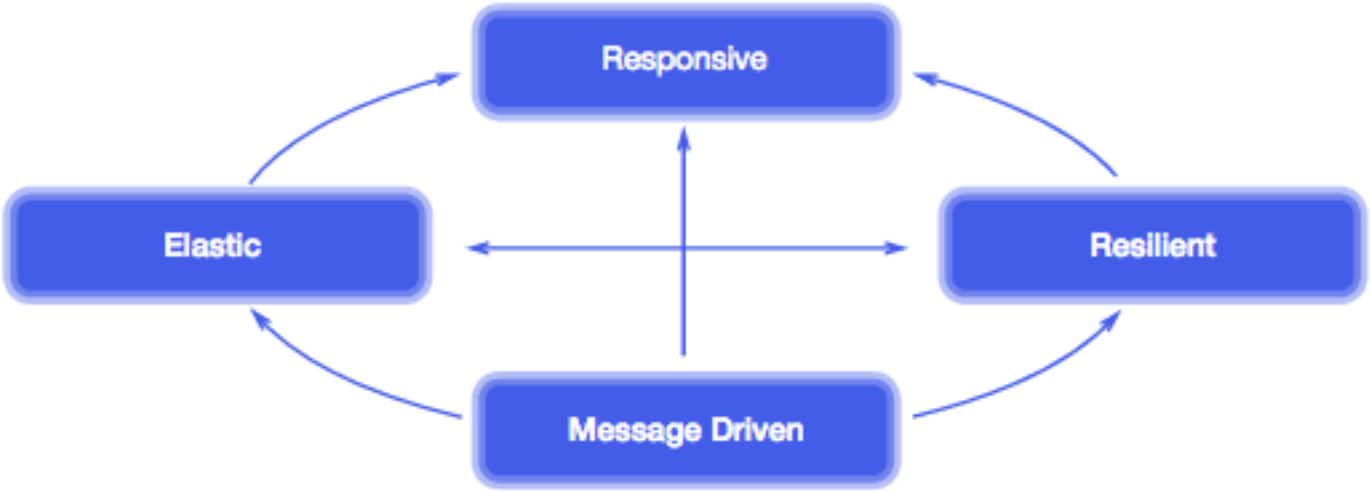
Play Framework

- Majority of code gets compiled including:
- - Templates
  - - Assets (CoffeeScript, LESS)
- Detect errors as early as possible

JVM


interpreter
class loader
bytecode
signature
heap
arguments
receiver
adapter
verifier
type profile
jit compiler
method

Reactive



Configuration

```
# The application languages
# ~~~~~
application.langs="en"

# Global object class
# ~~~~~
# Define the Global object class for this application.
# Default to Global in the root package.
# application.global=Global

# Database configuration
# ~~~~~
#You can declare as many datasources as you want.
#By convention, the default datasource is named `default`

db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
```

- The configuration file of a Play application must be defined in conf/application.conf. It uses the HOCON format

Routing

- conf/routes is the configuration file used by the router. This file lists all of the routes needed by the application.
- Each route consists of an HTTP method and URI pattern, both associated with a call to an Action generator.

Dependency Injection

- Dependency injection is a widely used design pattern that helps separate your components' behaviour from dependency resolution.

- Runtime dependency injection is so called because the dependency graph is created, wired and validated at runtime
- If a dependency cannot be found for a particular component, you won't get an error until you run your application.

Dependency Injection

- If you have a component, such as a controller, and it requires some other components as dependencies, then this can be declared using the @Inject annotation.
- The @Inject annotation can be used on fields or on constructors

Dependency Injection

```
 import javax.inject._
import play.api.libs.ws._

class MyComponent @Inject() (ws: WSClient) {
  // ...
}
```

- the @Inject annotation must come after the class name but before the constructor parameters, and must have parentheses.

Singleton

```
import javax.inject._

@Singleton
class CurrentSharePrice {
  @volatile private var price = 0

  def set(p: Int) = price = p
  def get         = price
}
```

- If a component that holds some state, such as a cache, or a connection to an external resource, or a component might be expensive to create.
- In these cases it may be important that there is only be one instance of that component. This can be achieved using the @Singleton annotation

Clean up

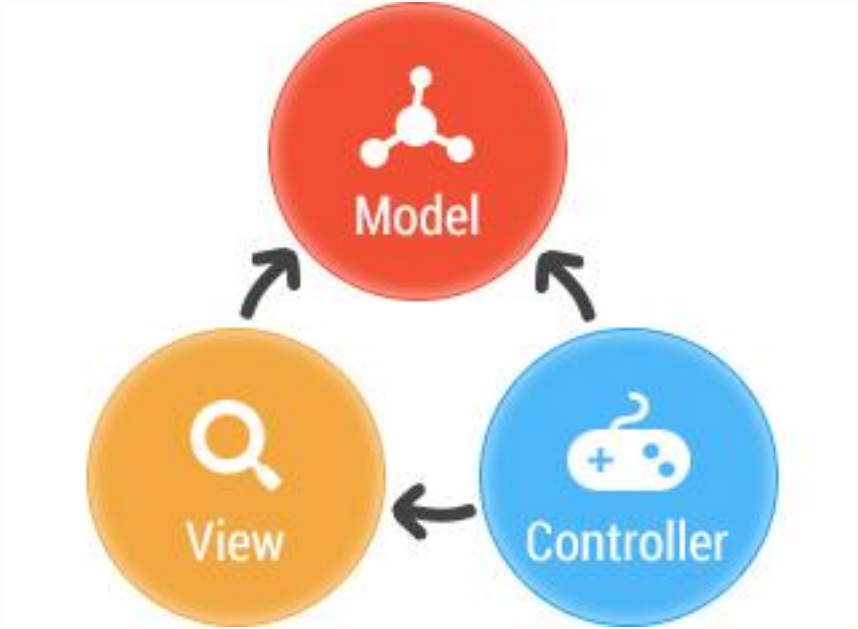
- components may need to be cleaned up when Play shuts down, for example, to stop thread pools.
- Play provides an ApplicationLifecycle component that can be used to register hooks to stop your component when Play shuts down:

Clean up

```
import scala.concurrent.Future
import javax.inject._
import play.api.inject.ApplicationLifecycle

@Singleton
class MessageQueueConnection @Inject() (lifecycle: ApplicationLifecycle) {
  val connection = connectToMessageQueue()
  lifecycle.addStopHook { () =>
    Future.successful(connection.stop())
  }

  //...
}
```

- ApplicationLifecycle will stop all components in reverse order from when they were created.
- any components that you depend on can still safely be used in your component's stop hook.
- Because you depend on them, they must have been created before your component was, and therefore won't be stopped until after your component is stopped.

MVC



Vehicle Display

- We are going to build an application to
  - Display list of vehicles
  - Add Vehicle

Create Project from Play Seed

- Open Command prompt and type the following
  - `sbt new playframework/play-scala-seed.g8`


```
scala-session >sbt new playframework/play-scala-seed.g8
```

Create Project from Play Seed

```
scala-session > sbt new playframework/play-scala-seed.g8
[info] Loading global plugins from C:\Users\muthuishere\.sbt\1.0\plugin
[info] Set current project to play (in build file:/C:/muthu/gitworkspac
[info] Set current project to play (in build file:/C:/muthu/gitworkspac

This template generates a Play Scala project

name [play-scala-seed]: vehicle-list
organization [com.example]: sessions.playframework
```

- Provide name as vehicle list
- Provide organization as `sessions.playframework`
- Import the project to Intellij

Run the project

- Open Command prompt and got project root folder and type
  - `sbt run 9000`

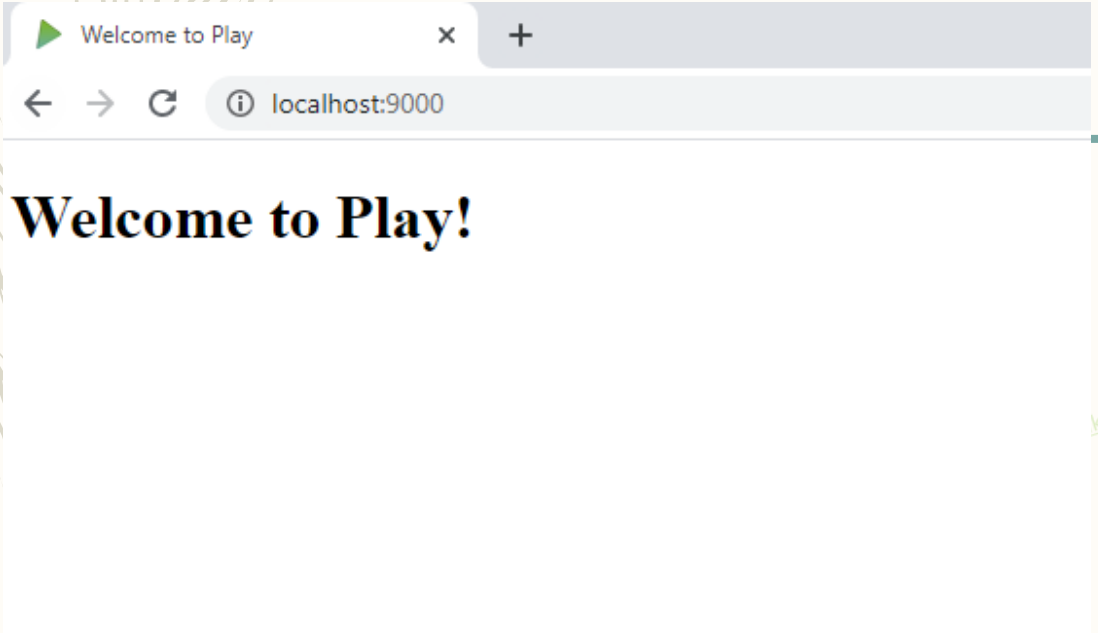

```
scala-session > sbt run 9000
```

Run the project

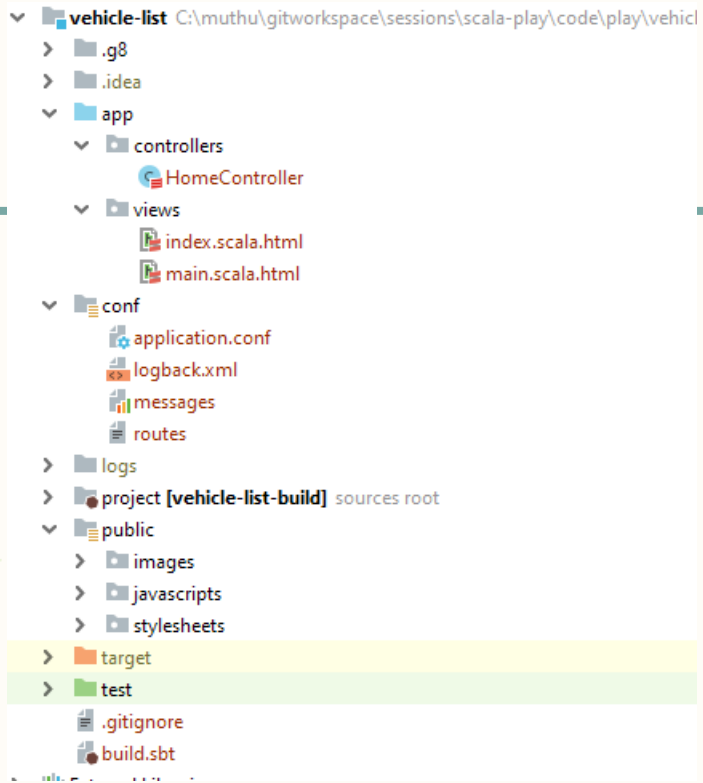
```
scala-session > sbt run 9000
[info] Loading global plugins from C:\Users\muthuishere\.sbt\1.0\plugins
[info] Loading settings for project vehicle-list-build from plugins.sbt ...
[info] Loading project definition from C:\muthu\gitworkspace\sessions\scala
[info] Loading settings for project root from build.sbt ...
[info] Set current project to vehicle-list (in build file:/C:/muthu/gitwork
ist/)

--- (Running the application, auto-reloading is enabled) ---

[info] p.c.s.AkkaHttpServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

(Server started, use Enter to stop and go back to the console...)
```

- Open the browser and goto http://localhost:9000

Run the project



- Open the browser and goto http://localhost:9000



Controllers

Play applications use controllers to handle HTTP requests and responses.

Play controllers are composed of actions that have specific functionality

A Controller is a singleton class extending class of type play.mvc. BaseController


```
@Singleton
class HomeController @Inject() (val controllerComponents: ControllerComponents) extends BaseController {
```

Controllers

- A method name index will create an instance of Action to render Html Page

```
def index() = Action { implicit request: Request[AnyContent] =>
  Ok(views.html.index())
}
```

Implicit

```
def index() = Action { implicit request: Request[AnyContent] =>
  Ok(views.html.index())
}
```

- An implicit parameter is just a function parameter annotated with the implicit keyword.
- if no value is supplied when called, the compiler will look for an implicit value and pass it in for you.

Parameters – Without Implicit

```
class PreferredDish(val preference: String)

object Restaurant {

  def serve(dish: PreferredDish = null) = {
    println("Welcome,Your Preferred dish is  " + dish.preference)

  }

  def main(args: Array[String]): Unit = {
    Restaurant.serve();
  }
}
```

Parameters – With Implicit

```
object UserPreferences {
  implicit val dish = new PreferredDish("Briyani ")
}
```

```
object Restaurant {
  def serve(implicit dish: PreferredDish) = {
    println("Welcome,Your Preferred dish is  " + dish.preference)
  }

  def main(args: Array[String]): Unit = {
    println("Attempting to serve through implicit value")
    Restaurant.serve;
  }
}
```

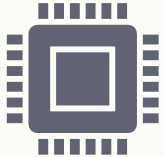
Controllers

- Request AnyContent defines the request may be any type
- Ok refers to HTTP status ok

```
def index() = Action { implicit request: Request[AnyContent] =>
  Ok(views.html.index())
}
```

- Views.html.index() refers views folder with file named index.scala.html

Exercise



- Modify the index method in HomeController

```
def index() = Action { implicit request: Request[AnyContent] =>
  Ok(views.html.vehicles())
}
```

- Fix the issues

Routes

- Play applications use a router to map HTTP requests to controller actions
- All the routes are configured in the file `\conf\routes`
- Each route starts with the HTTP method, followed by the URI pattern. The last element is the call definition.

```
# An example controller showing a sample home page
GET     /                           controllers.HomeController.index
```

- Add comments with character #

Http Methods

GET

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

POST

The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

PUT

The PUT method replaces all current representations of the target resource with the request payload.

DELETE

The DELETE method deletes the specified resource.

Routes

- Play offers default route controller for managing actions

```
# Redirects to https://www.infosys.com/ with 303 See Other
GET    /about       controllers.Default.redirect(to = "https://www.infosys.com/")

# Responds with 404 Not Found
GET    /orders      controllers.Default.notFound

# Responds with 500 Internal Server Error
GET    /clients     controllers.Default.error

# Responds with 501 Not Implemented
GET    /posts       controllers.Default.todo
```

Http Status Codes

100–199	• Informational responses
200–299	• Successful responses
300–399	• Redirects
400–499	• Client errors
500–599	• Server errors

Application Health

- We will create an endpoint health to denote the status of application
- It just need to response UP , if the application is running
- How can we do it?

Application Health

- Create a HealthController class annotate with singleton and extend with BaseController and inject controller components

```
@Singleton
class HealthController @Inject()(val controllerComponents: ControllerComponents) extends BaseController
```

Application Health

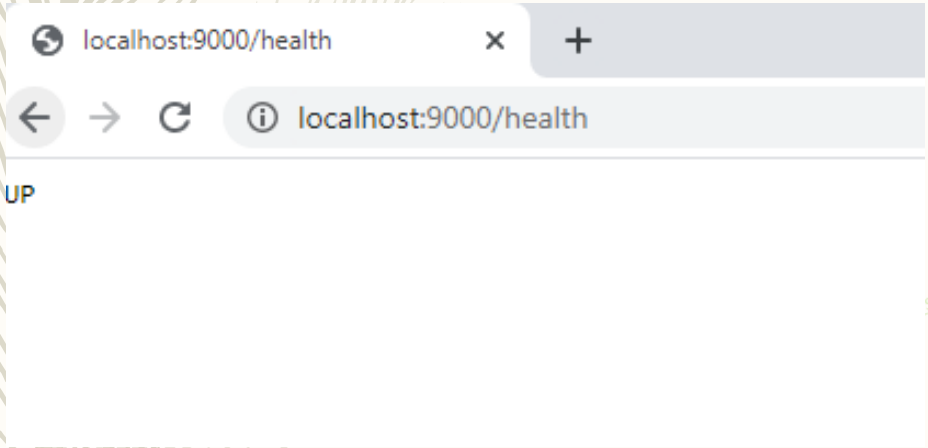
- Create method to return Action with text UP

```
def index = Action{
  Ok("UP")
}
```

- Add a route in the file \conf\routes

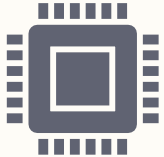
```
GET     /health          controllers.HealthController.index
```

Application Health



- Goto Url
  http://localhost:9000/health

Exercise



- Create an endpoint time to display current time

List all Vehicles

- We will update our home page to display all vehicles available on https://github.com/muthuishere/scala-workshop/tree/master/data
- Copy the files and place it under models folder
- Change the Vehicle class to case class for object decomposition


```
case class Vehicle (var sNo:Int,var name:String,
          var manufacturer:String,var year:Int,var fuelType:String,var
transmission:String)
```

List all Vehicles

- Create operations for CRUD in Vehicle.scala

```
object Vehicle{

    var vehicles = ListBuffer.empty ++ VehicleService.getAllVehicles()

    def all(): List[Vehicle] = {
      vehicles.toList
    }

    def create(vehicle: Vehicle): Unit = {
      vehicles += vehicle
    }
```

List all Vehicles

```
    def getById(id: Long): Vehicle = {
      vehicles.find(_.sNo == id).getOrElse(null)
    }


    def delete(id: Long): Unit = {
      vehicles -= getById(id)
    }


}
```

List all Vehicles

- Add a input parameter in vehicles.scala.html at top

```
@(vehicles: List[Vehicle])
```

List all Vehicles

- Add bootstrap CSS and proceed

```
<div class="table-responsive">
<table class="table table-striped table-hover">
    <thead>
        <tr>
            <th scope="col">#</th>
            <th scope="col">Name</th>
            <th scope="col">Manufacturer</th>

            <th scope="col">Year</th>
            <th scope="col">Type</th>
            <th scope="col">Transmission</th>
        </tr>
    </thead>
```

List all Vehicles

- Iterate and print

```
@vehicles.map { vehicle =>

    <tr>
        <th scope="row">@vehicle.sNo</th>
        <td>@vehicle.name</td>
        <td>@vehicle.manufacturer</td>
        <td>@vehicle.year</td>
        <td>@vehicle.fuelType</td>
        <td>@vehicle.transmission</td>

        <td></td>
    </tr>
}
```

Create Vehicle

- We are going to add a button for creating vehicle

Create vehicle - routes

- Add a route in routes

```
POST    /vehicles              controllers.HomeController.newVehicle
```

Create vehicle - html

- Add a parameter form

```
@(vehicles: List[Vehicle],vehicleForm: Form[Vehicle])
            (implicit request: RequestHeader, messagesProvider: MessagesProvider)
```

Create vehicle - html

- Import helper classes

```
@import helper._
```

Create vehicle - html

- Use form fields

```
<h2>Add a new Vehicle</h2>
@form(routes.HomeController.newVehicle) {

    @inputText(vehicleForm("sNo"))
    @inputText(vehicleForm("name"))
@inputText(vehicleForm("manufacturer"))
@inputText(vehicleForm("year"))
@inputText(vehicleForm("fuelType"))
@inputText(vehicleForm("transmission"))
    <input type="submit" value="Create">
    }
```

Create Vehicle - Form

- Create a new method in `HomeController for form`

```
val vehicleForm = Form(
  mapping(
    "sNo" -> number,
    "name" -> text,
    "manufacturer" -> text,
    "year" -> number,
    "fuelType" -> text,
    "transmission" -> text,

  )(Vehicle.apply)(Vehicle.unapply)
)
```

apply

- Mathematicians have their own little funny ways, so instead of saying "then we call function f passing it x as a parameter" as we programmers would say, they talk about "applying function f to its argument x".
- It comes from the idea that you often want to apply something to an object.
- Scala guys thought that, as it occurs in many situation, it could be nice to have a shortcut to call apply.

apply – unapply

- Thus when you give parameters directly to an object, it's desugared as if you pass these parameters to the apply function of that object:

- Apply takes arguments and by convention will return a value related to the object's name. If we take Scala's case classes as "correct" usage then the object's apply will construct a instance without needing to add "new".

apply – unapply

- Sample case

```
object ModernPerson {

  def apply(name: String, suffix: String) = name + "#" + suffix

  def unapply(name: String): Option[(String, String)] = {
    //simple argument extractor
    val parts = name.split("#")
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
```

apply – unapply

```
//It actually calls ModernPerson.apply("Gandhi","MK") and returns Gandhi.mk
val mkg =  ModernPerson("Gandhi","MK")

//
//If you want to deconstruct, call
val ModernPerson(values) = mkg

print(values)
```

Create vehicle - Handler

- Create a new method in `HomeController`

```
def newVehicle =  Action { implicit request =>
  vehicleForm.bindFromRequest.fold(
    errors => BadRequest(views.html.vehicles(Vehicle.all(), errors)),
    current => {
      Vehicle.create(current)
      Redirect(routes.HomeController.index)

    }
  )
}
```

Create vehicle- Controller

- Implement trait `I18nSupport to HomeController`

```
class HomeController @Inject()(val controllerComponents: ControllerComponents) extends
                        BaseController  with I18nSupport
```

Create Vehicle

- Update views method in `HomeController`


```
def index() = Action { implicit request: Request[AnyContent] =>

  Ok(views.html.vehicles(Vehicle.all(),vehicleForm))

}
```

Web API

- We will create a REST ful API end points

Web API - Create Project from Play Seed

- Open Command prompt and type the following
  - `sbt new playframework/play-scala-seed.g8`


```
scala-session >sbt new playframework/play-scala-seed.g8
```

Web API – Use the same Vehicle Model

- Use the same vehicle class
- Create a VehicleController class

```
@Singleton
class VehicleController @Inject()( val controllerComponen
ts: ControllerComponents) extends BaseController {
```

- Add CRUD operation methods

Web API – List API

- Write the below code in VehicleController

```
def list = Action { implicit request =>

    Ok(Json.toJson(Vehicle.all()))
 }
```


- Add the route

GET /vehicles controllers.VehicleController.list()

Web API – List API

- Update the Vehicle Object

```
object Vehicle{
  implicit val writes = Json.writes[Vehicle]
  implicit val reads = Json.reads[Vehicle]
```

Web API – Create API

- Write the below code

```
def create() = Action { implicit request: Request[AnyContent] =>
    val json = request.body.asJson.get
    val newVehicle = json.as[Vehicle]
    Vehicle.create(newVehicle)
    Ok

 }
```


- Add the route

POST /vehicles controllers.VehicleController.create()

Web API – Exercise

- Write for delete and getbyID

Web API - Create User Model

- Create User class

```
case class User(var username:String, var password:String, var userrole:String)
```

Web API – Generate Users

- Generate Users

```
def getAllUsers():List[User] ={
  var users = new ListBuffer[User]()
  users += new User("xuser","xuser","guest");
  users += new User("xadmin","xadmin","admin");

  return users.toList
}
```

Web API – Initialize User List

- Load Users

```
var users = ListBuffer.empty ++ UserService.getAllUsers()
```

Web API – Login Validate

- Perform the following

```
def validateUserCredentials(input: UserInput):User ={

   users.find(currentuser=> currentuser.username == input.username && currentuser.pass
word == input.password).getOrElse(throw new Exception("Invalid User"))

}
```

Web API – Login Validate

- Get input as User Input and return LoginResponse

```
def login() = Action { implicit request: Request[AnyContent] =>
 val json = request.body.asJson.get
  val userInput = json.as[UserInput]

val user = User.validateUserCredentials(userInput);
val token = AuthService.createToken(user)
val result= TokenResponse.getAllUserResponse(user,token)
Ok(Json.toJson(result))


}
```

Web API – Login Validate

- Get input as User Input and return LoginResponse

```
def login() = Action { implicit request: Request[AnyContent] =>
 val json = request.body.asJson.get
  val userInput = json.as[UserInput]


   val user = User.validateUserCredentials(userInput);
   val token = ""
   val result= LoginResponse.getAsUserResponse(user,token)
   Ok(Json.toJson(result))

}
```

```
case class LoginResponse(var username:String, var role:String,var token:String)
```

Web API – Authentication

- Include dependencies

```
libraryDependencies += "io.jsonwebtoken" % "jjwt" % "0.9.1"
```

Web API – Authentication

- Add Auth Service

```
val JwtSecretKey = "secretKey"
val JwtSecretAlgo = "HS256"
```

```
def createToken(user: User): String = {
  Jwts.builder().setSubject(user.username).claim("roles", user.userrole).setIssuedAt(new Date())
    .signWith(SignatureAlgorithm.HS256, JwtSecretKey).compact();
}

def getUser(jwtToken: String): LoginResponse = {
    val claims = Jwts.parser.setSigningKey(JwtSecretKey).parseClaimsJws(jwtToken).getBody

    val tokenResponse = new LoginResponse(claims.getSubject, claims.get("roles").toString, "")

    println(tokenResponse)


  tokenResponse
}
```

Web API – Authentication

- Add Auth Service

```
val JwtSecretKey = "secretKey"
val JwtSecretAlgo = "HS256"
```

```
def createToken(user: User): String = {
  Jwts.builder().setSubject(user.username).claim("roles",
user.userrole).setIssuedAt(new Date())
    .signWith(SignatureAlgorithm.HS256, JwtSecretKey).compact();
}
```

Web API – Authentication

- Add Auth Service

```
def getUser(jwtToken: String): LoginResponse = {
    val claims = Jwts.parser.setSigningKey(JwtSecretKey).parseClaimsJws(jwtToken).getBody

    val loginResponse = new LoginResponse(claims.getSubject, claims.get("roles").toString, "")
loginResponse
}
```

Web API – Authentication

- Update token Creation in user controller

```
def login() = Action { implicit request: Request[AnyContent] =>
 val json = request.body.asJson.get
  val userInput = json.as[UserInput]


   val user = User.validateUserCredentials(userInput);
   val token = AuthService.createToken(user)
   val result= LoginResponse.getAsUserResponse(user,token)
   Ok(Json.toJson(result))

}
```

Web API – Authentication

- Create a class GuestAuthentication

```
import filters.AuthService
import javax.inject.Inject
import models.users.LoginResponse
import play.api.Logging
import play.api.mvc.Results._
import play.api.mvc._

import scala.concurrent.{ExecutionContext, Future}

class GuestAuthentication @Inject()(parser: BodyParsers.Default)(implicit ec: ExecutionContext)
  extends ActionBuilderImpl(parser)
    with Logging {
```

Web API – Authentication

- Update vehiclecontroller to use Authentication

```
class VehicleController @Inject()(val guestAuthentication: GuestAuther
Controller {
```

- Change the Action to guestAuthentication for list

```
def list = guestAuthentication { implicit request =>

  Ok(Json.toJson(Vehicle.all()))
}
```

Db Connectivity

- Add resolver and dependencies

```
resolvers +=   "Java.net Typesafe Repository" at "https://repo.typesafe.com/typesafe/ivy-releases/"
resolvers +=  "Java.net Maven Repository" at "https://repo.typesafe.com/typesafe/maven-releases/"
```

```
libraryDependencies += jdbc
libraryDependencies += evolutions
libraryDependencies += "org.playframework.anorm" %% "anorm" % "2.6.5"
libraryDependencies += "com.h2database" % "h2" % "1.4.199"
```

Db Connectivity

- Update Configuration

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"

db.default.user="sa"
db.default.password=""
```

Db Connectivity

- Update Configuration

```
fixedConnectionPool = 9
play.db {
  prototype {
    hikaricp.minimumIdle = ${fixedConnectionPool}
    hikaricp.maximumPoolSize = ${fixedConnectionPool}
  }
}
```

Db Connectivity

- Update Evolutions file from git

- Create DatabaseExecutionContext.scala

```
import javax.inject._

import akka.actor.ActorSystem
import play.api.libs.concurrent.CustomExecutionContext

/**
  * This class is a pointer to an execution context configured to point to "d
atabase.dispatcher"
  * in the "application.conf" file.
  */
```

Db Connectivity

- Create DatabaseExecutionContext.scala

```
@Singleton
class DatabaseExecutionContext @Inject()(system: ActorSyst
em) extends CustomExecutionContext(system, "database.dispa
tcher")
```

Db Connectivity

- Create VehicleRepository.scala

```
import anorm._
import javax.inject.Inject
import models.DatabaseExecutionContext
import play.api.db.DBApi

import scala.concurrent.Future


@javax.inject.Singleton
class VehicleRepository @Inject()(dbapi: DBApi)(implicit ec: DatabaseExecutionContext)
```

Db Connectivity

- Create VehicleRepository.scala

```
private val db = dbapi.database("default")

val parser: RowParser[Vehicle] = Macro.namedParser[Vehicle]

def list(): Future[List[Vehicle]] = Future {


  db.withConnection { implicit connection =>



    val result: List[Vehicle] = SQL"SELECT * FROM VEHICLE".as(parser.*)
    result
  }
}(ec)
```

Db Connectivity

- Create VehicleRepository.scala

```
  def insert(vehicle: Vehicle): Option[Long] = {
    val id: Option[Long] = db.withConnection { implicit c =>
      SQL("""INSERT INTO VEHICLE ( "NAME",  "MANUFACTURER", "YEAR", "FUELTYPE", "TRANS
MISSION") VALUES ({name},{manufacturer},{year},{fuelType},{transmission})""")
        .on("name" -> vehicle.name,"manufacturer" -> vehicle.manufacturer,"year" -
> vehicle.year,"fuelType" -> vehicle.fuelType,"transmission" -> vehicle.transmission)
        .executeInsert()
    }

    id
  }
```

Db Connectivity

- Create VehicleRepository.scala

```
def insertAll(vehicles:List[Vehicle]): Future[List[Vehicle]] = Future {

  vehicles.foreach(vehicle=>insert(vehicle))

  null
}(ec)
```

Db Connectivity

- Inject VehicleRepository in controller


```
@Singleton
class VehicleController @Inject()(val  vehicleRepository: VehicleRepository,
```

Db Connectivity

- Update the list method to async

```
def list = Action.async { implicit request =>

  vehicleRepository.list().map { vehicles =>
    Ok(Json.toJson(vehicles))
  }

}
```