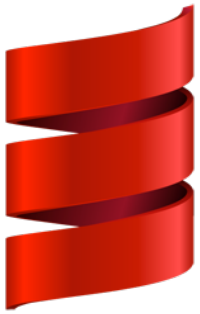


# Scala

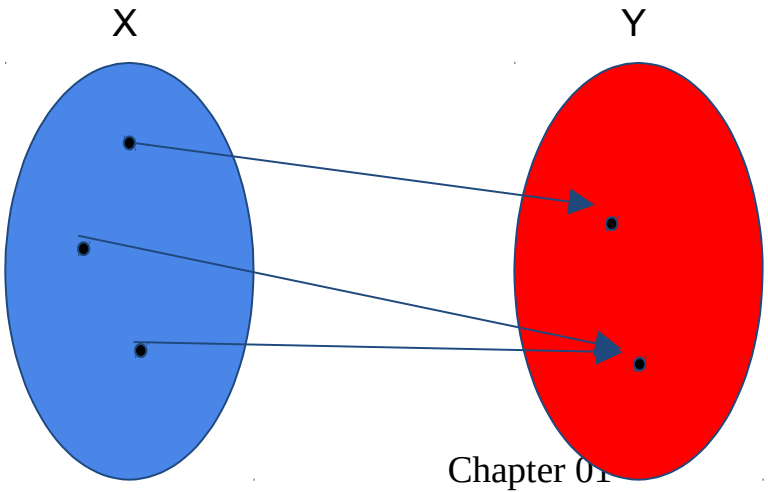
Introduction to Scala

# Object Oriented Programming

Encapsulation/information hiding.
Inheritance.
Polymorphism/dynamic binding.
All predefined types are objects.
All operations are performed by sending messages to objects.
All user-defined types are objects

# Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data, thus avoid any side-effect and mutability.



# Mutable vs. Immutable Data

An object is called mutable when you can alter the contents of the object if you have a reference to it.

```
var x = "foo"
```

The contents of the immutable object can’t be altered even if you have a reference to it.

```
val x = "foo" #This is similar to java final variables
```

# Features of Functional Programing

Higher-order functions
Lexical closures
Pattern matching
Single assignment
Lazy evaluation
Type inference
Tail call optimization
List comprehensions
Monadic effects

# What is Scala?

Scala is an emerging general-purpose, type-safe language for the Java Platform that combines object-oriented and functional programming.
Scala stands for Scalable Language.
Scala Doc
http://www.scala-lang.org/api/current/#package

# Programming for Multicores

Traditional thread-based concurrency model: Split a program into multiple concurrently running tasks (threads) and each operates on shared memory.

This leads to hard-to-find race conditions and deadlock issues.

# Scala Actor Model

An actor encapsulates data, code, and its own thread of control
Actors communicate asynchronously using immutable message-passing techniques
Model relies on a shared-nothing policy
Lightweight in nature.

```
(0 to 100).par.map(x => x * x)
```

# Features of Scala

Scala as an object-oriented language
Scala as a functional language
Scala as a multi-paradigm language - unify OOP and functional programming styles
Scala as a scalable and extensible language
Scala runs on the JVM
Scala integrates well with Java and its ecosystem, including tools, libraries, and IDEs.
Scala compiles to Java byte code
Java class file disassembler javap to disassemble Scala byte code
Harness all the benefits of JVM-like performance and stability out of the box

# Tools for Scala

IDE - Eclipse, IntelliJ
Dependency Managers: SBT, Maven
Scala REPL
Zeppelin Notebook
- Like REPL
- Captures Output
- Good for exploratory analysis
- “Polyglot” notebook

# Static vs Dynamic Type

**Static Typing … compile time checks for type matching**

- Both values and the variables have types.
  - A number variable can't hold anything other than a number.
  - Types are determined and enforced at compile time or declaration time.

**Scala is a statically typed language … type inferred language**

**Dynamic Typing … runtime check for type matching**

- Values have types but the variables don't.
  - It's possible to successively put a number and a string inside the same variable.

# Scala Code Execution

Create a simple HelloWorld.scala

```
object HelloWorld{
   def main(args:Array[String]){
      print("Hello world")
   }
}
```

Execute the scala code

```
$ scala HelloWorld.scala
```

# Scala REPL

Read, Evaluate, Print, Loop

```
scala> :help
:cp <path>                add a jar or directory to the classpath
:history [num]            show the history (optional num is commands to show)
:implicits [-v]           show the implicits in scope
:paste                    enter paste mode: ctrl-D to end
:quit                     exit the interpreter
:type [-v] <expr>         display the type of an expression without evaluating it
```

# Basic Data Types

| Type | Description | |
|---|---|---|
| Byte | 8-bits signed integer. Range: -127 to +127 | `val i:Byte = 1` |
| Short | 16-bits signed integer. Range: -32,768 to + 32,768 | `val b:Short = 1` |
| Int | 32-bit signed integer. Range: –2,147,483,648, 2,147,483,647 | `val i = 1` |
| Long | 64-bit signed integer. Range: -9,223,372,036,854,775,808, 9,223,372,036,854,775,807 | `val l = 1L` |
| Float | A single-precision 32-bit IEEE 754 floating point. | `val f = 0.0f` |
| Double | A double-precision 64-bit IEEE 754 floating point. | `val d = 0.0` |
| Boolean | True / False | `val b = true` |
| Char | A single 16-bit Unicode character. Range: \u0000, \uffff (65,535) | `val ch = 'c'` |

# Variable Assignment

```
val immutableVariable = 100
var mutableVariable = 100
var mutableVariable:Int = _ //set default
```

Default value for class object

```
case class Person(name:String, age: Int)
var p:Option[Person] = None
p = Some(Person("Alex", 30))
```

# Lazy Evaluation

```
var i = 0
lazy val j = i + 10
j #ouput: 10
i = 10
j #output: 10
```

Only val can be declared lazy
Use lazy when you want to defer computation of value of a variable

# Def variables

```
var i = 0
def j = i + 10
j //output: 10
i = 20
j //Output: 30
```

Def also defer the computation of value of a variable.
Unlike val, each time the variable is accessed, the value of variable is recomputed.

# Conditional Statements

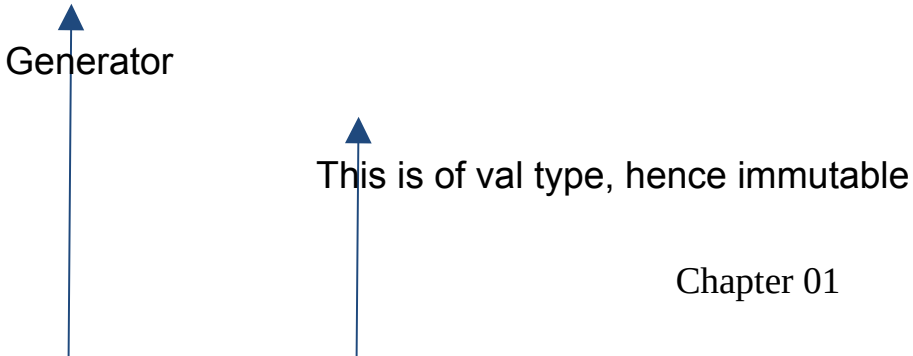
```
if (x % 2 == 0){
    println(“even”)
}else{
    println(“odd”)
}
```

If-else to return value

```
val result = if(x%2 == 0) “even” else “odd”
```

# For loop … imperative form

```
val files = new java.io.File(".").listFiles
for(file <- files) {
     val filename = file.getName
     if(fileName.endsWith(".scala")){
          println(file)
     }
}
```



# More than one generators in for loop

```
scala> val l1 = Array(1, 2, 3)
scala> val l2 = Array("one", "two", "three")
scala> for(v1 <- l1; v2 <- l2) println(v1, v2)
(1,one)
(1,two)
(1,three)
(2,one)
(2,two)
(2,three)
(3,one)
(3,two)
(3,three)
```

Another variable combine two list 1:1

```
scala> l1.zip(l2).foreach(println)
```

# For Comprehension

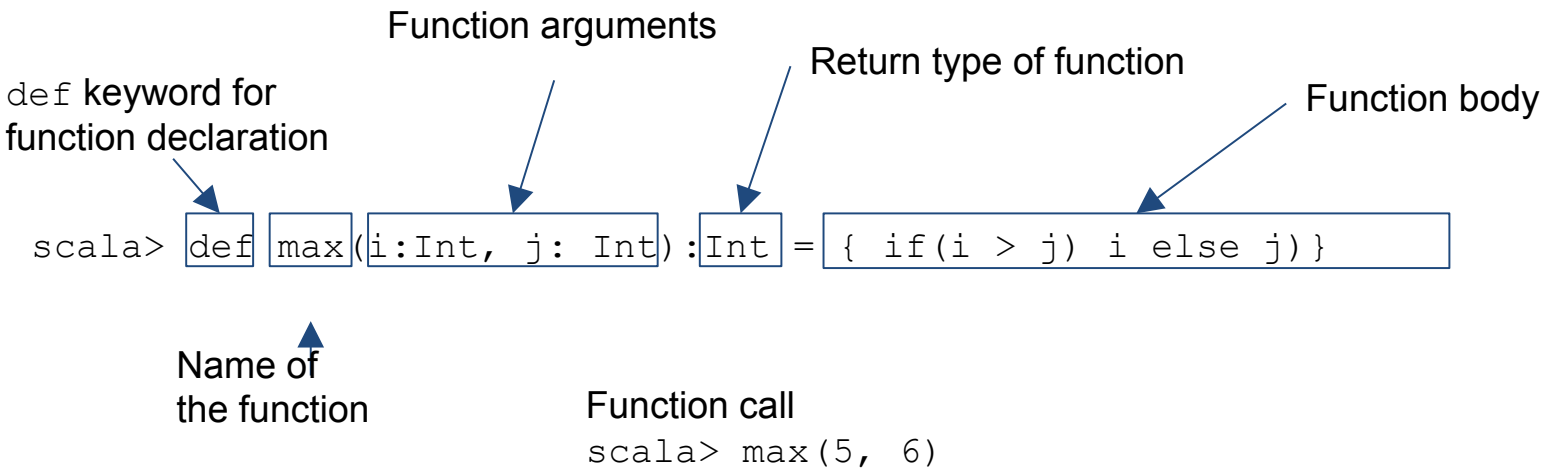
```
scala> val l1 = Array(1, 2, 3)
scala> val l2 = Array(10, 20, 30)
scala> val result = for{v1 <- l1; v2 <- l2} yield v1 + v2
result: Array[Int] = Array(11, 21, 31, 12, 22, 32, 13, 23,
33)
```

Generator expression after for keyword can be encapsulated within either parenthesis or curly braces.

`yield` returns a value … it is list comprehension of scala

# Function

# Function




- The return type is optional. Scala infers the return type of a function automatically.
- “return” is optional

# Function

You can define function any of the following forms

```
def even(i: Int) = i % 2 == 0
def even(i: Int) = {i % 2 == 0}
def even(i: Int):Boolean = i % 2 == 0
def even(i: Int):Boolean = {i % 2 == 0}
```

# Parameterized Function

Need a function with flexible data type for argument?

```
scala> def toList[A](value:A) = List(value)
toList: [A](value: A)List[A]

scala> toList("hello")
res57: List[String] = List(hello)

scala> toList(100)
res58: List[Int] = List(100)
```

- Similar to java generics
- Naming the parameterized types is that they normally start at A and go up to Z as necessary.
- This contrasts with the Java convention of using T, K, V, and E.
- Scala DOES NOT support method overloading

# Anonymous Function

Functions are first class citizen - of the same stature of variables and objects

Functions can be passed as argument or parameter

Anonymous function is a shorthand for creating such function without naming the function

```
val series = Array(3, 6, 7, 10)
series.reduce((i:Int, j:Int) => i + j)
```

Another form of anonymous function … using underscore (_) placeholder

```
series.reduce(_ + _)
```

# Function Variable

```
val mod = (i:Int) => i % 2 == 0
mod(5)
```

Create an alias for a function

```
val c = scala.math.cos _
c(20)
```

# Unit

Similar to java void

```
def show(msg:String):Unit = println(msg)
```

# Partially Applied Function

Create a new function from an existing function

```
def sum(i:Int, j:Int, k:Int) = i + j + k
val f = sum(1, 2, _:Int)
f(3) // This is similar to call sum(1, 2, 3)
```

# Closure Functions

A function that closes over the environment in which it’s defined

```
import scala.collection.mutable.ArrayBuffer
val fruits = ArrayBuffer("Apple")

def addToBasket(s:String):Unit = {
    fruits.add(s)
    println(fruits)
}

def buyStuff(f: (String) => Unit, s:String) = {
    f(s)
}
buyStuff(addToBasket, "Banana")
```

# Closure Function - More Example

```
def genericDateParser(simpleDateFormat:String) = {
    val format = new java.text.SimpleDateFormat(simpleDateFormat)
    format.parse(_:String)
}

val dateParser = genericDateParser("MM-dd-yyyy")
dateParser("07-06-2013")

val utcParser = genericDateParser("yyyy-MM-dd'T'HH:mm:ss.SSSZ")
utcParser("2001-07-04T12:08:56.235-0700")
```

# Higher Order Function

Functions that accept other function as argument

```
Array(0, 3, 4).map(println)
```

# Curried Function

A function with multiple parameter lists of single parameter.

```
def foo(as: Int*)(bs: Int*)(cs: Int*) = as.sum * bs.sum * cs.sum
foo(1, 2, 3)(4, 5, 6, 7, 9)(10, 11)
```

Turn any function as curried function

```
def adder(m: Int, n: Int) = m + n
val curriedAdd = (adder _).curried
val addTwo = curriedAdd(2)
addTwo(4)
```

# Currying Vs Partials

Both look a lot similar

Curried functions are single parameter, while the partial functions are not

For example,

```
def modN(n: Int, x: Int) = ((x % n) == 0)
def modNCurried(n: Int)(x: Int) = ((x % n) == 0)
```

`modN(5, _:Int)` and `modNCurried(5)` work in same way

# Usefulness of Currying

Currying is mostly used if the second parameter section is a function or a by name parameter.

```
def max[T](xs: List[T])(compare: (T, T) => Boolean) = {
    xs.reduceLeft{(a, b) => if(compare(a, b)) a else b}
}
max(List(1, -3, 43, 0))((x, y) => x > y)
```

Uncurried version

```
max(List(1, -3, 43, 0), (x: Int, y: Int) => x < y)
```

The curried version is cleaner than uncurried version

# Pattern Matching

# Pattern Matching On Value

```
val dayOfWeek = 4

val day = dayOfWeek match {
    case 1 => "Sunday"
    case 2 => "Monday"
    case 3 => "Tuesday"
    case 4 => "Wednesday"
    case 5 => "Thursday"
    case 6 => "Friday"
    case 7 => "Saturday"
    case _ => "Invalid" //matches everything else
}
```

Pattern match is possible on strings and complex values, types, variables, constants, and constructors

# Pattern Matching On Object Type

```
def printType(obj: AnyRef) = obj match {
  case s: String => println("This is string")
  case l: List[_] => println("This is List") //_ represents AnyRef
  case a: Array[_] => println("This is an array")
  case d: java.util.Date => println("This is a date")
  case p: Person => println(s”${p.name}”)
  case User(name, age) => println(s”${name}”)
  case (category: String, count: Int) => println(s“$category, $count”)

}

printType((“hello”, 1))
```

# Pattern matching with condition

```
val d = 100
val result = d match {
    case d if d < 10 => "Less than 10"
    case d if d < 100 => "Between 11 to 99"
    case _ => "100 or more"
}
println(result) //Output: 100 or more
```

# Exception Handling

```
val breakException = new RuntimeException("break exception")
//java.lang.RuntimeException

val l = Array(0, 2, 4, "Six")

try{
    l.map{v => v match {
        case i:Int => i * 2
        case s:String => throw breakException
    }}
}catch{case _:RuntimeException => }
```

- Scala exceptions are unchecked
- You can do type check using case pattern matching

# Importing Libraries

# Importing Libraries

Simple import statement

```
import java.util.Date
```

Control the name of the imported class

```
import java.sql.{Date => SqlDate}
```

Import members of package … members can be classes, objects, methods

```
import System._
```

Hide certain class of a package from users

```
import java.util.{Date => _}
```

Imports can be relatively loaded

```
import scala.util
import util.Random
```

# Importing Libraries

Multiple import

```
import java.util.{Date, Calendar}
```

# Predef

Classes, objects, and methods defined in Predef will be automatically available in scala without explicit import

Example:

scala.collection.immutable.Map, scala.collection.immutable.Set, the scala.collection.immutable.List scala.collection.immutable.Nil, implicit conversion

http://www.scala-lang.org/api/current/scala/Predef$.html

# Class

# Classes and Constructor

```
scala> class SparkSession(val name:String, val master:String)
```

Class Name

Primary Constructor

```
scala> val c = new SparkSession("Spark Client", "spark://localhost:7077")
```

When val/var are missed, members are treated as private … not accessible outside

# Zero Argument Constructor

```
class SparkSession(val name:String, val master:String){
    require(name != null, "Name is required")
    require(master != null, "Master is required")
def this() = this("Spark Client Application", "localhost[*]")
}

val c = new SparkSession() //Works with zero argument constructor
val c = new SparkSession(null, null) //Throws exception
```

# Auxiliary Constructor

```
class Pizza(var crustSize:Int, var crustType: String){
    def this(crustSize:Int) {
        this(crustSize, Pizza.DEFAULT_CRUST_TYPE)
    }
    def this(crustType:String) {
        this(Pizza.DEFAULT_CRUST_SIZE, crustType)
    }
}
object Pizza{
    val DEFAULT_CRUST_TYPE = "Thin"
    val DEFAULT_CRUST_SIZE = 10
}
```

# Singleton

Scala does not have static variables … use Singleton instead

A singleton object restricts the instantiation of a class to one object.

```
object SparkSession{
  def areuok() = print("I am ok")
}

SparkSession.areuok
```

# Factory Pattern

```
abstract class ProductBase {def features(): Array[String]}
class Book extends ProductBase {
    override def features() = Array("Authors", "Page Count", "Binding")
}
class Apparel extends ProductBase {
    override def features() = Array("Brand", "For", "Color", "Fabric")
}
object Product {
    def apply(productType: String) = productType match {
        case "Book" => new Book
        case "Apparel" => new Apparel
    }
}
val book = Product("Book") #Syntactic Sugar for Product.apply("Book")
val apparel = Product("Apparel")
```

# Companion Object

Companion Class

`private` keyword hides the class constructor

```
class SparkSession private (val name: String, val master: String){
    override def toString() =  s"name: ${name}, master: ${master}"
}
object SparkSession{
    private var _session:SparkSession = null
    def apply(name: String, master: String) = {
        if(this._session == null){
            this._session = new SparkSession(name, master)
        }
        this._session
    }
}
val ss = SparkSession("Spark Client", "local")
println(ss)
```

Companion Object

- Object shares the same name as class
- Companion is often used in factory model
- Companion class and objects must be defined in the same scala file

# Package Object

Create package.scala for a package directory
Objects and methods in package.scala is accessible to all members of package
Normally helper functions are put in the package.scala

# Traits

A mixin is a class that provides certain functionality that could be used by other classes
… like java interface, abstract class.
A trait is like an abstract class meant to be added to other classes as a mixin.
A trait cannot be instantiated

```
class WoodPecker extends Bird with TreeScaling with Pecking
```

# Abstracts and Interface in Java World

**Abstract:**

- An abstract class is a class that is only partially implemented by the programmer.
- It may contain one or more abstract methods.
- An abstract method is simply a function definition that serves to tell the programmer that the method must be implemented in a child class.

**Interface:** An interface is a fully abstract class; none of its methods are implemented

# How Scala Traits compares with those Java

Scala trails may have abstracts members/methods like Java Interface
Scala traits may have implemented members/methods like Java Abstract
You can mix more than one traits into a class
Trait can control which classes it can be mixed into
Scala also has abstract class, but it is more common to use trait

# Trait Example

```
trait PizzaTrait{
    var numToppings: Int //abstract
    var size = 14    //concrete
    val maxNumToppings = 10 //concrete
}

class Pizza extends PizzaTrait{
    var numToppings = 0 //override not needed
    size = 16 //var or override not neede
override val maxNumToppings = 10 //val requires override keyword
}

val p = new Pizza()
```

# Case Class

A special type of class
All arguments are prefixed by val
`equals` and `hashCode` are implemented
`toString` is implemented to return class name along with members
`copy` is implemented to create a clone of the object
A companion object is created (Class name ends with $)
Commonly used to easily wrap a data structure
Cannot have more than 22 parameters
Case class automatically implements two traits - Serializable and Product

```
case class Person(name:String, age: Int)
```

# Case Class Pattern Match

```
case class Person(name:String, age: Int)
val p = Person("Martin Odersky", 58)

p match {
    case Person(name, _) => println(name),
    case p:Person => println(p.name)
    case _ => print("Not Person Type")
}
```

Scala handles this pattern matching using a method called unapply

# Named Argument

```
case class Person(name:String, age: Int)
val p = Person(age = 58, name = "Martin Odersky")
```

Order of the arguments does not matter
You can mix positional and named arguments … but avoid it.

# Implicit Conversion

It is a method that takes one type of parameter and returns another type

```
val i:Int = 2.3 // Throws error: type mismatch
def double2Int(d:Double) = d.toInt
val i:Int = double2Int(2.3) //returns 2
```

Implicit Model

```
implicit def double2Int(d: Double): Int = d.toInt
val i:Int = 2.3 // Now, it does not throw any error and returns 2
```

# Define New Operator

Suppose, you want to define a new operator --> to do the following

```
scala> 2 --> 100 // Create a generator from 2 to 100
```

One Possible Solution

```
class generatorInt(i:Int){
   def -->(j: Int) = i to j
}
implicit def int2generatorInt(i:Int): generatorInt = new
generatorInt(i)

8 --> 100   This is equivalent to 8.-->(100)
```

# Implicit Class

```
implicit class RangeMaker(left: Int) {
  def -->(right: Int): Range = left to right
}

scala> 3 --> 5
res0: Range = Range(3, 4, 5)
```

- Implicit class must have a primary constructor with one argument
- Implicit classes cannot be top level classes (package level)
- Define them inside a class or object
- Powerful feature but creates code readability and maintenance issues

# Type Conversion

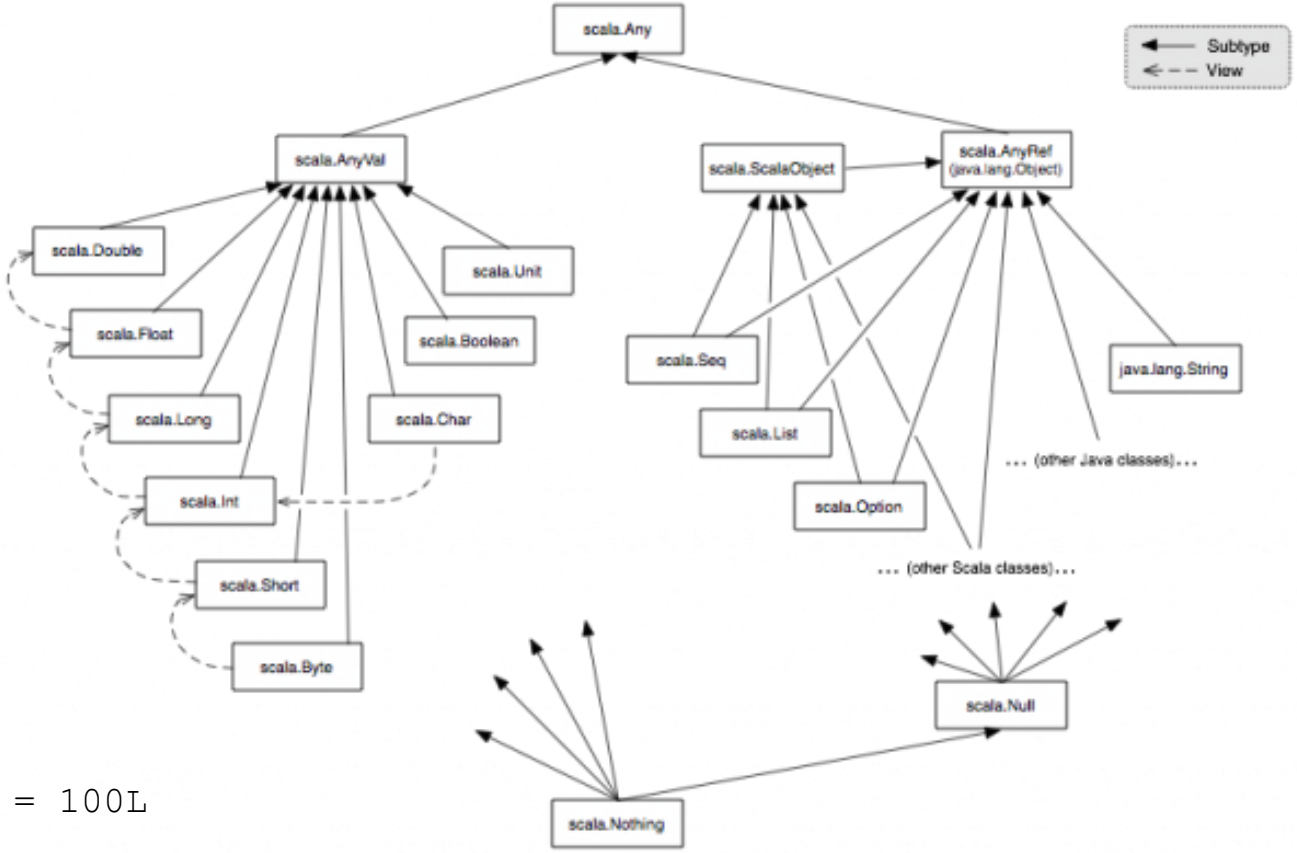
Use `asInstanceOf` method to cast an instance to desired type.

```
val recognizer =
cm.lookup("recognizer").asInstanceOf[Recognizer]
```

This is similar to java statement

```
Recognizer recognizer = (Recognizer) cm.lookup("recognizer")
```

# Class Hierarchy



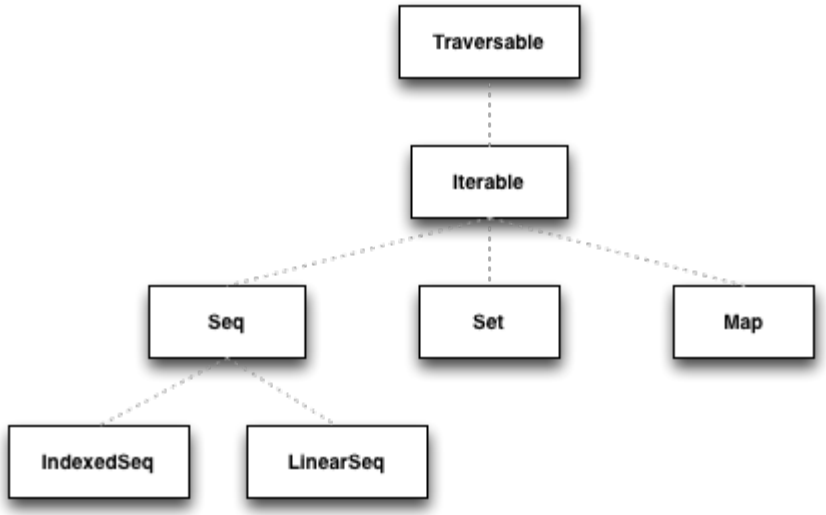

Valid Statement:

```
val x: Float = 100L
```

# Collection

# Collection Hierarchy

**Sequence**: linear collection - indexed or linear
**Map**: collection of key-value pairs
**Set**: collection of unique values

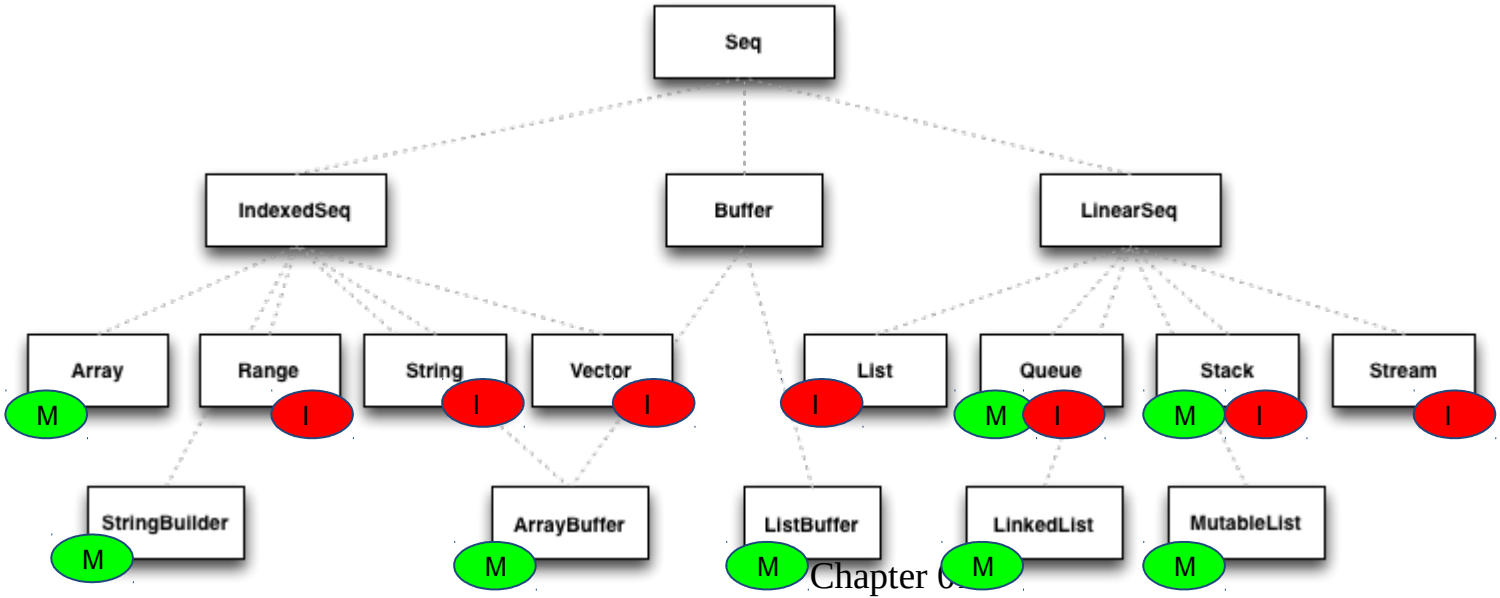


# Sequences

M Mutable Sequence

I Immutable Sequence

| | Immutable | Mutable |
|---|---|---|
| IndexedSeq (efficient for random access) | Vector | ArrayBuffer |
| LinearSeq (linear access) | List | ListBuffer |



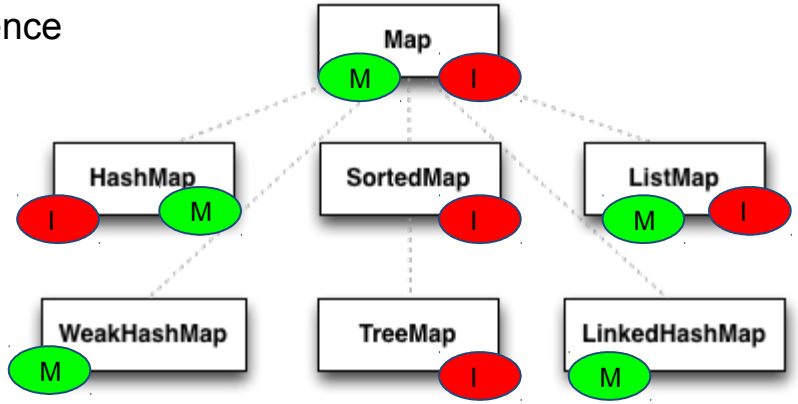
# Immutable Sequence

| | Indexed | Linear | Description |
|---|---|---|---|
| List | | Yes | A singly linked list. Suitable for recursive algorithm |
| Queue | | Yes | First-in first-out data structure |
| Range | Yes | | A range of integer value |
| Stack | | Yes | A last-in and first-out data structure |
| Stream | | Yes | Similar to list but lazy and persistent |
| String | Yes | | Immutable indexed sequence of characters |
| Vector | Yes | | Immutable indexed sequence |

# Maps

M Mutable Sequence

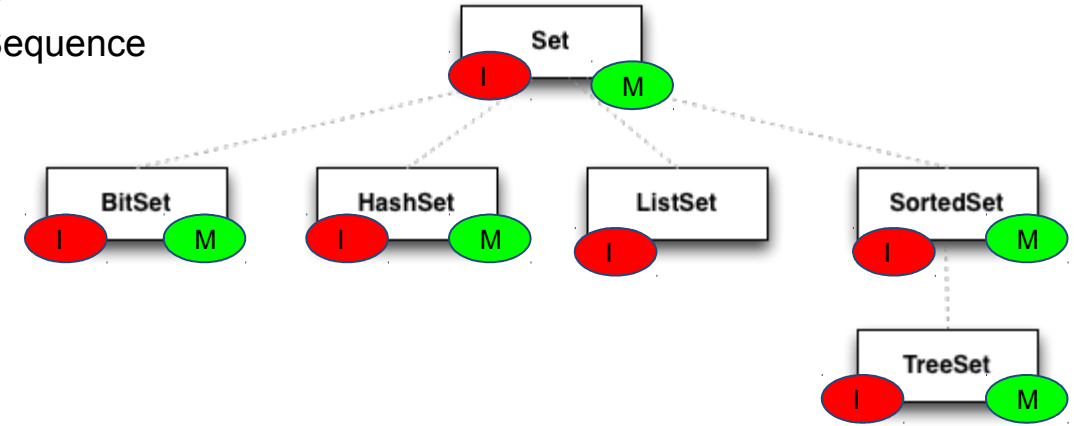
I Immutable Sequence



# Sets

M Mutable Sequence

I Immutable Sequence



# Other Collections

| | Description |
|---|---|
| Enumerate | A finite list of constant values, e.g. months |
| Iterator | It is Travesable*Once*. Useful for stateful traversal. |
| Option | Collection of 0 or 1 value - Some[T] or None[T] |
| Tuple | Heterogeneous collection of elements. Max 22 elements. |

# Performance Consideration

Collection types are optimized for their use case

Performance Characteristics for sequential collections

Head

Tail

Apply

Update

Prepend

Append

Insert

# Immutable Sequence

For fast, general purpose, immutable sequential collection choose Vector
Vector is indexed, immutable sequential collection
Random access any element in constant time
“When in doubt use Vector”

```
val v = Vector(3, 6, 1, 10, 5)
v.take(2)
v.filter(_ > 5)
v.update(0, 100)
v.reduce(_ + _)
val newV = v.updated(0, 100)
```

# Mutable Sequence

Choose ArrayBuffer for general purpose mutable sequence

ArrayBuffer is indexed

Efficient in building large collection whenever new items are added to the end

```
import scala.collection.mutable.ArrayBuffer
val prices  = ArrayBuffer[Double]()
prices.append(10.0)
prices.appendAll(Array(40.0, 20.4, 10, 30))
println(prices)
```

# Loop over a collection

```
val fruits = "Apple" :: "Banana" :: "Orange" :: Nil
```

Option 1:

```
for(fruit <- fruits){
    println(fruit)
}
```

Option 2:

```
fruits.foreach(println)
```


Option 3:

```
fruits.zipWithIndex.foreach{
    case(fruit, i) => println(s"[$i] $fruit ")
}
```

# Lazy Transformer

By default transformation over collections exception Stream is “strict”
Strict operation allocates memory and computes when invoked
Lazy operation computed when results are actual needed through reference
Create a “view” to invoke a lazy transformation

```
(0 to 100).view.map{e => Thread.sleep(10); e * 2} //Generates another view,
  no computation
(0 to 100).view.map{e => Thread.sleep(10); e * 2}.toVector //force
  calculation
```

# Sorting Collection

```
class User(var name:String, var age: Int) extends Ordered[User]{
    override def toString() = s"name: $name, age: $age"
    def compare(that: User):Int = {
        if(this.name == that.name)
            0
        else if(this.name > that.name)
            1
        else
            -1
    }
}
val users = List(
new User("Mark", 40),
new User("Harry", 50),
new User("Ed", 20))
users.sorted
```

# Concatenate Collection of String

```
val planets = Array("Venus", "Mars", "Earth")
planets.mkString(",")
```

# Option, Some, None Pattern

```
def toInt(s:String): Option[Int] = {
    try{
        Some(Integer.parseInt(s.trim))
    }catch{
        case e:Exception => None
    }
}
toInt("100")
toInt("foo")
```

Getting value from Option

```
toInt("100").getOrElse(0) // returns 100
toInt("foo").getOrElse(0) // returns 0
Array("0", "2", "3", "foo").map(e => toInt(e).getOrElse(0))reduce(_+_)
Array("0", "2", "3", "foo").map(toInt).collect{case Some(i) => i}
```

# Try,Success,Failure Pattern

```
import scala.util.{Try,Success,Failure}
def toInt(s:String): Try[Int] = Try(s.toInt) //Try captures any exceptions

toInt("100")
toInt("foo")
```

Getting value from Option

```
toInt("100").getOrElse(0) //returns 100
toInt("foo").getOrElse(0) //return 0
Array("0", "2", "3", "foo").map(toInt).collect{case Success(i) => i}
```

Here is one liner for conversion

```
Try{"abcd".toInt}.getOrElse(0) //returns 0
```

# Null, null, Nil, Nothing, None, Unit

Null– It’s a Trait.
null– It’s an instance of Null- Similar to Java null.
Nil– Represents an empty List of anything of zero length. It’s not that it refers to nothing but it refers to List which has no contents.
Nothing is a Trait. It’s a subtype of everything. But not superclass of anything. There are no instances of Nothing.
None– Used to represent a sensible return value. Just to avoid null pointer exception. Option has exactly 2 subclasses- Some and None. None signifies no result from the method.
Unit– Type of method that doesn’t return a value of anys sort.