Advanced Programming Methods
Seminar
Introduction to Scala
Seminar 14

Content

- Introduction
- Basics
- Unified Types
- Classes
- Traits
- Tuples
- Class Composition with Mixins
- Higher-Order Functions
- Nested Methods
- Multiple Parameter Lists
- Case classes
- Pattern matching

References

NOTE: The slides are based on the following free tutorials. You may want to consult them too.

1. https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html
2. https://docs.scala-lang.org/tour/tour-of-scala.html
3. https://docs.scala-lang.org/overviews/scala-book/introduction.html

Introduction

What is Scala?

- is a modern multi-paradigm programming language
- Is designed to express common programming patterns in a concise, elegant, and type-safe way.
- integrates features of object-oriented and functional languages.

-

Scala-pure object oriented language

- every value is an object.
- types and behaviors of objects are described by classes and traits.
- classes can be extended
  - by subclassing, and
  - by using a mixin-based composition mechanism (as a replacement for multiple inheritance).

-

Scala-functional language

- every function is a value.
- provides a lightweight syntax for defining anonymous functions
- it supports higher-order functions,
- it allows functions to be nested,
- it supports currying.
- case classes and its built-in support for pattern matching provide the functionality of algebraic types
- Singleton objects group functions that aren’t members of a class.

Scala-statically typed language

- type system enforces, at compile-time, that abstractions are used in a safe and coherent manner.
- Type inference means the user is not required to annotate code with redundant type information.
- Type system supports: Generic classes, Variance annotations, Upper and lower type bounds, Inner classes and abstract type members, Compound types, Explicitly typed self references, Implicit parameters and conversions, Polymorphic methods

Scala-is Extensible

- the development of domain-specific applications often requires domain-specific language extensions.
- Scala provides a unique combination of language mechanisms that make it straightforward to add new language constructs in the form of libraries.

Example

```scala
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

- method main takes the command line arguments, an array of strings, as parameter

- the main method does not return a value. Therefore, its return type is declared as Unit.

Example

```scala
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

- **a singleton object, that is a class with a single instance.**

- This instance is created on demand, the first time it is used.

- the main method is not declared as static here

- static members (methods or fields) do not exist in Scala. Rather than defining static members, the Scala programmer declares these members in singleton objects.

Interaction with Java

- **very easy to interact with Java code.**

- java.lang package is imported by default

- there is no need to implement equivalent classes in the Scala class library– we can simply import the classes of the corresponding Java packages

- it is also possible to inherit from Java classes and implement Java interfaces directly in Scala.

- multiple classes can be imported from the same package by enclosing them in curly braces

- when importing all the names of a package or class, one uses the underscore character (_) instead of the asterisk (*)

Interaction with Java

```
import java.util.{Date, Locale}
import java.text.DateFormat._

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

    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
```

- **Methods taking one argument can be used with an infix syntax:**

```
df format now
```

Instead of

```
df.format(now)
```

First way to try Scala code

You can run Scala in your browser with ScalaFiddle:

https://scalafiddle.io.

Basics-Values

You can name the results of expressions using the val keyword:

```
val x = 1 + 1
println(x) // 2
```

Values cannot be re-assigned:

```
x = 3 // This does not compile
```

The type of a value can be omitted and inferred, or it can be explicitly stated:

```
val x: Int = 1 + 1
```

Basics-Variables

Variables are like values, except you can re-assign them:

```
var x = 1 + 1
x = 3 // This compiles because "x" is declared with the "var" keyword.
println(x * x) // 9
```

The var type can be declared or it can be inferred:

```
var x: Int = 1 + 1
```

Basics-Blocks, Functions

The result of the last expression in the block is the result of the overall block:

```
println({
  val x = 1 + 1
  x + 1
}) // 3
```

Functions are expressions that have parameters, and take arguments.

Anonymous function:

```
(x: Int) => x + 1
```

Functions with names:

```
val addOne = (x: Int) => x + 1
println(addOne(1)) // 2
```

Basics-Functions

A function can have 0 or multiple parameters:

```
val add = (x: Int, y: Int) => x + y
println(add(1, 2)) // 3
```

```
val getTheAnswer = () => 42
println(getTheAnswer()) // 42
```

Basics-Methods

Methods look and behave very similar to functions.

- Methods are defined with the def keyword. def is followed by a name, parameter list(s), a return type, and a body:

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

- Multiple parameter lists:

```
def addThenMultiply(x: Int, y: Int)(multiplier: Int): Int = (x + y) * multiplier
println(addThenMultiply(1, 2)(3)) // 9
```

- No parameters

```
def name: String = System.getProperty("user.name")
println("Hello, " + name + "!")
```

Basics-Methods

The last expression in the body is the method’s return value.

```
def getSquareString(input: Double): String = {
  val square = input * input
  square.toString
}
println(getSquareString(2.5)) // 6.25
```

Basic- Classes

- Classes in Scala are declared using a syntax which is close to Java's syntax
- Scala classes can have parameters

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

Basic- Classes

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

- Complex class takes two arguments, which are the real and imaginary part

- These arguments must be passed when creating an instance of class Complex, as follows: new Complex(1.5, 2.3)

- the return type of two methods re and im is not given explicitly and it is inferred automatically by the compiler

Basic - Classes

- in order to call the methods re and im, one has to put an empty pair of parenthesis after their name:

```
object ComplexNumbers {
  def main(args: Array[String]): Unit = {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Basic -Classes

- methods without arguments differ from methods with zero arguments in that they don't have parenthesis after their name, neither in their definition nor in their use:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
}
```

Everything is an **OBJECT**

- is a pure object-oriented language in the sense that everything is an object, including numbers or functions

- numbers are objects and operators are methods (operators symbols are valid Scala identifiers)

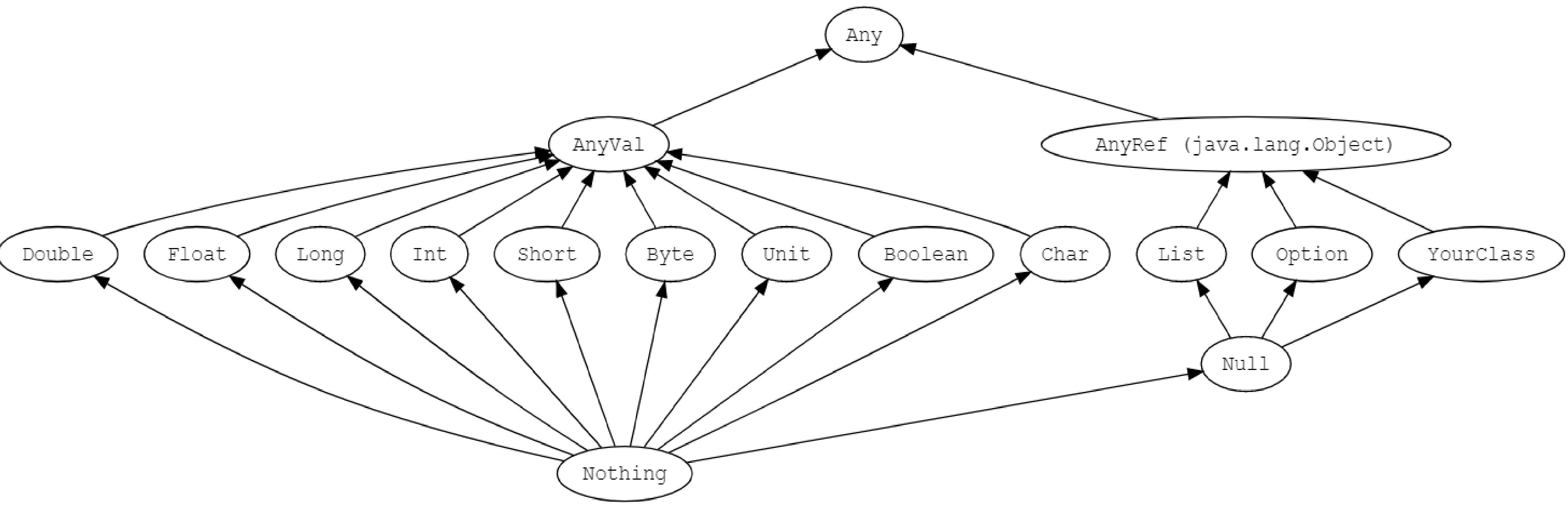
```
1 + 2 * 3 / x
```

is equivalent to

```
1.+(2.*(3)./(x))
```

Scala Unified Type

- unified types for both references and values
- Any is the supertype of all types, also called the top type
- AnyVal represents value types.
- AnyRef represents reference types.



Unified Type

```
val list: List[Any] = List(
  "a string",
  732,  // an integer
  'c',  // a character
  true, // a boolean value
  () => "an anonymous function returning a string"
 )

 list.foreach(element => println(element))
```

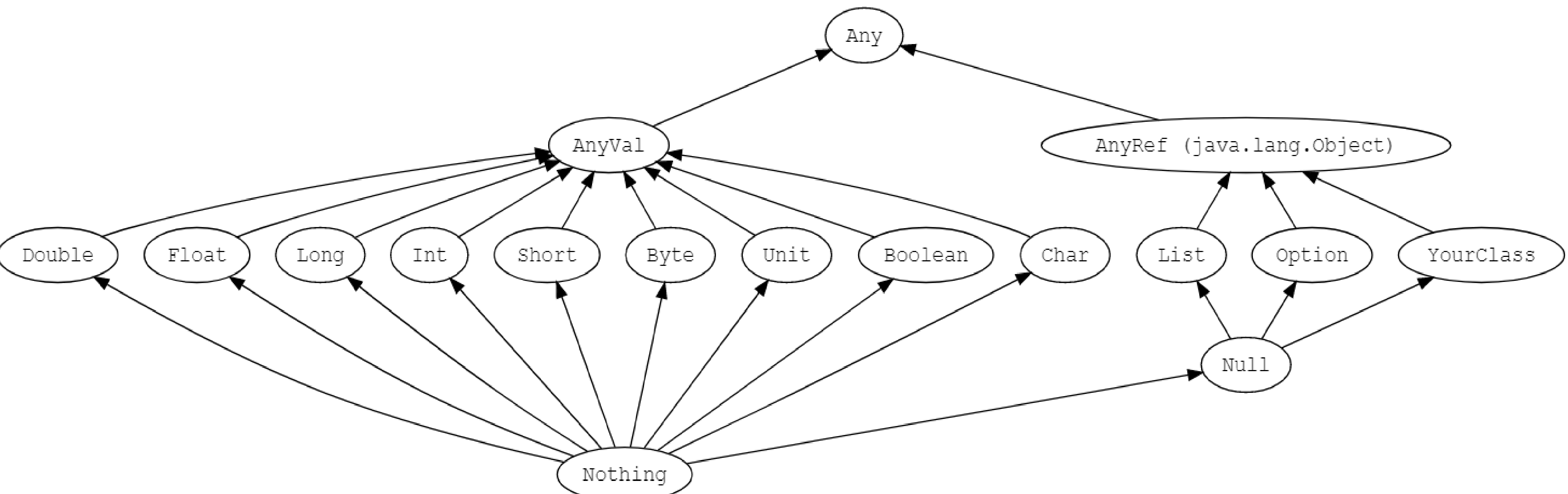
- the output is:

```
a string
732
c
true
<function>
```

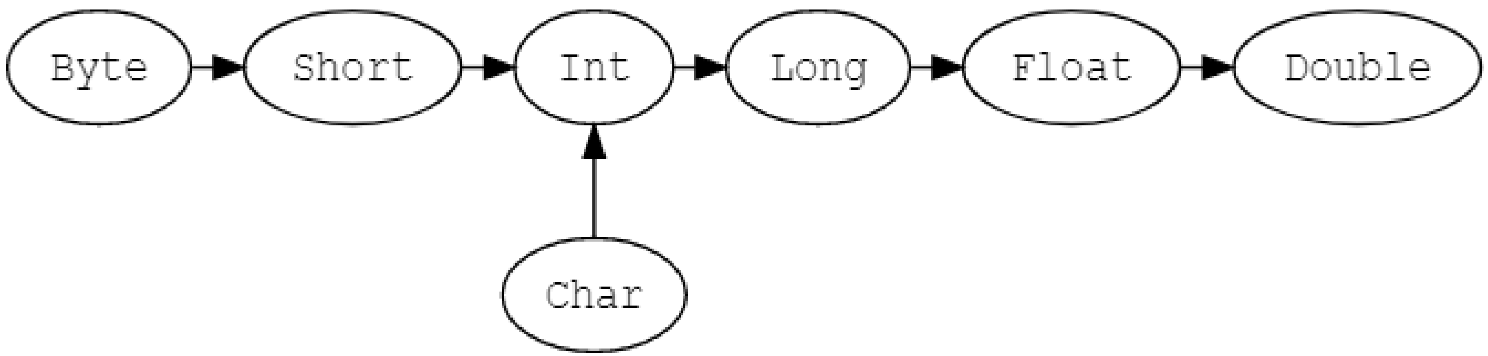
Scala Type Hierarchy

- **Nothing is a subtype of all types, also called the bottom type. There is no value that has type Nothing. A common use is to signal non-termination such as a thrown exception, program exit, or an infinite loop (i.e., it is the type of an expression which does not evaluate to a value, or a method that does not return normally).**

- **Null is a subtype of all reference types (i.e. any subtype of AnyRef). It has a single value identified by the keyword literal null. Null is provided mostly for interoperability with other JVM languages**



Value Type Casting




```
val x: Long = 987654321
val y: Float = x  // 9.8765434E8 (note that some precision is lost in this case)

val face: Char = '☺'
val number: Int = face  // 9786
```

Casting is unidirectional, the followings do not compile:
val x: Long = 987654321
val y: Float = x // 9.8765434E8
val z: Long = y // Does not conform

Classes-constructors

```
class Point(var x: Int = 0, var y: Int = 0)

val origin = new Point  // x and y are both set to 0
val point1 = new Point(1)
println(point1.x)  // prints 1
```

```
class Point(var x: Int = 0, var y: Int = 0)
val point2 = new Point(y=2) //good practice
println(point2.y)  // prints 2
```

Classes-Private Members

Members are public by default

```
class Point {
  private var _x = 0
  private var _y = 0
  private val bound = 100

  def x = _x
  def x_= (newValue: Int): Unit = {
    if (newValue < bound) _x = newValue else printWarning
  }

  def y = _y
  def y_= (newValue: Int): Unit = {
    if (newValue < bound) _y = newValue else printWarning
  }

  private def printWarning = println("WARNING: Out of bounds")
}

val point1 = new Point
point1.x = 99
point1.y = 101 // prints the warning
```

Classes-Constructors

Primary constructor parameters with val and var are public, but val 's are immutable:

```
class Point(val x: Int, val y: Int)
val point = new Point(1, 2)
point.x = 3  // <-- does not compile, it is immutable
```

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

```
class Point(x: Int, y: Int)
val point = new Point(1, 2)
point.x  // <-- does not compile, it is not visible
```

Classes- Inheritance and overriding

- all classes in Scala inherit from a super-class
- when no super-class is specified, as in the Complex example **scala.AnyRef** is implicitly used.
- **AnyRef** corresponds to **java.lang.Object**.

- It is mandatory to explicitly specify that a method overrides another one using the **override** modifier, in order to avoid accidental overriding.

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im >= 0) "+" else "") + im + "i"
}
```

Classes-Inheritance and overriding

- **usage of the overriding method**

```scala
object ComplexNumbers {
  def main(args: Array[String]): Unit = {
    val c = new Complex(1.2, 3.4)
    println("Overridden toString(): " + c.toString)
  }
}
```

Traits

- can be viewed as interfaces which can also contain code. Since Java 8, Java interfaces can also contain code, either using the default keyword, or as static methods

- In Scala, when a class inherits from a trait, it implements that trait's interface, and inherits all the code contained in the trait.

Example:
- When comparing objects, six different predicates can be useful: smaller, smaller or equal, equal, not equal, greater or equal, and greater.
- defining all of them is fastidious, especially since four out of these six can be expressed using the remaining two.
- given the equal and smaller predicates (for example), one can express the other ones.

Traits

- a new type called Ord, which plays the same role as Java's Comparable interface,
- default implementations of three predicates in terms of a fourth, abstract one.
- the predicates for equality and inequality do not appear here since they are by default present in all objects.

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean =  (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

Traits

- To make objects of a class comparable, it is therefore sufficient to define the predicates which test equality and inferiority, and mix in the Ord class

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d
  override def toString(): String = year + "-" + month + "-" + day
```

Traits

- we redefine the equals method, inherited from Object, so that it correctly compares dates by comparing their individual fields

```
override def equals(that: Any): Boolean =
  that.isInstanceOf[Date] && {
    val o = that.asInstanceOf[Date]
    o.day == day && o.month == month && o.year == year
  }
```

- **isInstanceOf, corresponds to Java's instanceof operator, and returns true if and only if the object on which it is applied is an instance of the given type**

- asInstanceOf, corresponds to Java's cast operator: if the object is an instance of the given type, it is viewed as such, otherwise a ClassCastException is thrown

Traits

```
def <(that: Any): Boolean = {
  if (!that.isInstanceOf[Date])
    sys.error("cannot compare " + that + " and a Date")

  val o = that.asInstanceOf[Date]
  (year < o.year) ||
  (year == o.year && (month < o.month ||
                     (month == o.month && day < o.day)))
}
```

- error from the package object scala.sys, which throws an exception with the given error message

Traits - Subtyping

Where a given trait is required, a subtype of the trait can be used instead.

```
import scala.collection.mutable.ArrayBuffer

trait Pet {
  val name: String //abstract field implemented by any subtype
}

class Cat(val name: String) extends Pet
class Dog(val name: String) extends Pet

val dog = new Dog("Harry")
val cat = new Cat("Sally")

val animals = ArrayBuffer.empty[Pet]
animals.append(dog)
animals.append(cat)
animals.foreach(pet => println(pet.name))  // Prints Harry Sally
```

Tuples

- a tuple is a value that contains a fixed number of elements, each with a distinct type.

- are immutable

- are especially handy for returning multiple values from a method

```
val ingredient = ("Sugar" , 25)
```

- **The inferred type of ingredient is (String, Int), which is shorthand for Tuple2[String, Int]**
- **To represent tuples, Scala uses a series of classes: Tuple2, Tuple3, etc., through Tuple22**

Tuples

- One way of accessing tuple elements is by position.

- The individual elements are named _1, _2, and so forth

```
println(ingredient._1) // Sugar
println(ingredient._2) // 25
```

Tuples -Pattern Matching

- **A tuple can also be taken apart using pattern matching**

```
val (name, quantity) = ingredient
println(name) // Sugar
println(quantity) // 25
```

Tuples – Pattern Matching

```
val planets =
  List(("Mercury", 57.9), ("Venus", 108.2), ("Earth", 149.6),
       ("Mars", 227.9), ("Jupiter", 778.3))
planets.foreach{
  case ("Earth", distance) =>
    println(s"Our planet is $distance million kilometers from the sun")
  case _ =>
}
```

Tuples - Comprehension

```
val numPairs = List((2, 5), (3, -7), (20, 56))
for ((a, b) <- numPairs) {
  println(a * b)
}
```

Class Composition with Mixins

Mixins are traits which are used to compose a class (as a replacement for multiple inheritance).
Classes can only have one superclass but many mixins (using the keywords extends and with respectively).
The mixins and the superclass may have the same supertype.

```
abstract class A {
  val message: String
}
class B extends A {
  val message = "I'm an instance of class B"
}
trait C extends A {
  def loudMessage = message.toUpperCase()
}
class D extends B with C

val d = new D
println(d.message)  // I'm an instance of class B
println(d.loudMessage)  // I'M AN INSTANCE OF CLASS B
```

Mixins - Example

The class has an abstract type T and the standard iterator methods.

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next(): T
}
```

A concrete class:

```
class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length
  def next() = {
    val ch = s charAt i
    i += 1
    ch
  }
}
```

Mixins - Example

A trait that extends the abstract class:

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit): Unit = while (hasNext) f(next())
}
```

Combining the functionalities into a single class:

```
class RichStringIter extends StringIterator("Scala") with RichIterator
val richStringIter = new RichStringIter
richStringIter.foreach(println)
```

Higher-order Functions

- pass functions as arguments
- store them in variables
- return them from other functions.

- functions are first class value (also objects) in Scala

- function passing should be familiar to many programmers: it is often used in user-interface code, to register call-back functions which get called when some event occurs.

Higher-order Functions

```
object Timer {
  def oncePerSecond(callback: () => Unit): Unit = {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies(): Unit = {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]): Unit = {
    oncePerSecond(timeFlies)
  }
}
```

- **a call-back function as argument.**

- **() => Unit is the type of all functions which take no arguments and return nothing (the type Unit is similar to void in C/C++)**

Higher-order Functions

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit): Unit = {

    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]): Unit = {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

- in Scala we can use anonymous functions, when a function is only used once

Higher-order Functions

```
object SalaryRaiser {

  private def promotion(salaries: List[Double], promotionFunction: Double => Double): 
List[Double] =
    salaries.map(promotionFunction)

  def smallPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * 1.1)

  def greatPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * math.log(salary))

  def hugePromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * salary)
}
```

Higher-order Functions

```
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"
}


val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)
val endpoint = "users"
val query = "id=1"

val url = getURL(endpoint, query) // "https://www.example.com/users?id=1": String
```

Nested Methods

```
def factorial(x: Int): Int = {
   def fact(x: Int, accumulator: Int): Int = {
     if (x <= 1) accumulator
     else fact(x - 1, x * accumulator)
   }
   fact(x, 1)
 }

 println("Factorial of 2: " + factorial(2))
 println("Factorial of 3: " + factorial(3))
```

Multiple Parameter Lists

An example defined in TraversableOnce trait:

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val res = numbers.foldLeft(0)((m, n) => m + n)
println(res) // 55
```

Scala can infer the types, so we can write:

```
numbers.foldLeft(0)(_ + _)
```

Multiple Parameter Lists – Partial Application

When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val numberFunc = numbers.foldLeft(List[Int]()) _

val squares = numberFunc((xs, x) => xs :+ x*x)
println(squares) // List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

val cubes = numberFunc((xs, x) => xs :+ x*x*x)
println(cubes)  // List(1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)
```

Case Classes

- **Problem:** a program to manipulate very simple arithmetic expressions composed of sums, integer constants and variables, for instance 1+2 and (x+x)+(7+y)

- **Problem Representation:** as a tree, where nodes are operations (here, the addition) and leaves are values (here constants or variables).
- Java representation: an abstract super-class for the trees, and one concrete sub-class per node or leaf.
- functional programming language: an algebraic data-type
- Scala: **case classes** which is somewhat in between the two

Case Classes

classes Sum, Var and Const are declared as case classes

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Case Classes

Differences from standard classes:

- the new keyword is not mandatory to create instances of these classes (i.e., one can write Const(5) instead of new Const(5))

- getter functions are automatically defined for the constructor parameters (i.e., it is possible to get the value of the v constructor parameter of some instance c of class Const just by writing c.v)

- default definitions for methods equals and hashCode are provided, which work on the structure of the instances and not on their identity

- a default definition for method toString is provided, and prints the value in a "source form" (e.g., the tree for expression x+1 prints as Sum(Var(x),Const(1)))

- instances of these classes can be decomposed through 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

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
matchTest(3)  // other
matchTest(1)  // one
```

Pattern Matching – Case Classes

```
 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


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"
  }
}
val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

println(showNotification(someSms))  // prints You got an SMS from 12345! Message: Are
you there?
println(showNotification(someVoiceRecording))  // you received a Voice Recording from
Tom! Click the link to hear it: voicerecording.org/id/123
```

Pattern Matching – Guards

```
def showImportantNotification(notification: Notification, importantPeopleInfo: Seq[String]): String = {
  notification match {
    case Email(sender, _, _) if importantPeopleInfo.contains(sender) =>
      "You got an email from special someone!"
    case SMS(number, _) if importantPeopleInfo.contains(number) =>
      "You got an SMS from special someone!"
    case other =>
      showNotification(other) // nothing special, delegate to our original showNotification function
  }
}
```

```
val importantPeopleInfo = Seq("867-5309", "jenny@gmail.com")
val importantSms = SMS("867-5309", "I'm here! Where are you?")

println(showImportantNotification(importantSms, importantPeopleInfo))
//prints You got an SMS from special someone!
```

Pattern Matching – on Type

```
abstract class Device
case class Phone(model: String) extends Device {
  def screenOff = "Turning screen off"
}

case class Computer(model: String) extends Device {
  def screenSaverOn = "Turning screen saver on..."
}


def goIdle(device: Device) = device match {
  case p: Phone => p.screenOff
  case c: Computer => c.screenSaverOn
}
```

Pattern Matching – Sealed Classes

Traits and classes can be marked sealed which means all subtypes must be declared in the same file. This assures that all subtypes are known.

```
sealed abstract class Furniture
case class Couch() extends Furniture
case class Chair() extends Furniture

def findPlaceToSit(piece: Furniture): String = piece match {
  case a: Couch => "Lie on the couch"
  case b: Chair => "Sit on the chair"
}
```

Example

- **Problem:** a function to evaluate an expression in some environment.
- The aim of the environment is to give values to variables.
- For example, the expression x+1 evaluated in an environment which associates the value 5 to variable x, written { x -> 5 }, gives 6 as result.

- Environment representation:
- some associative data-structure like a hash table
- a function which associates a value to a (variable) name

- **Scala**: a function which, when given the string "x" as argument, returns the integer 5, and fails with an exception otherwise.

```
{ case "x" => 5 }
```

Example

- use the type **String => Int** for environments, but it simplifies the program if we introduce a name for this type, and makes future changes easier

- the type **Environment** can be used as an alias of the type of functions from **String to Int**

```
type Environment = String => Int
```

Example

Pattern matching over the tree t:

1. checks if the tree t is a Sum, and if it is, it binds the left sub-tree to a new variable called l and the right sub-tree to a variable called r, and then proceeds with the evaluation of the expression following the arrow;

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

Pattern matching over the tree t:

2. if the tree is not a Sum, it goes on and checks if t is a Var; if it is, it binds the name contained in the Var node to a variable n and proceeds with the right-hand expression

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

Pattern matching over the tree t:

3. if the second check also fails, that is if t is neither a Sum nor a Var, it checks if it is a Const, and if it is, it binds the value contained in the Const node to a variable v and proceeds with the right-hand side,

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

Pattern matching over the tree t:

4. finally, if all checks fail, an exception is raised to signal the failure of the pattern matching expression; this could happen here only if more sub-classes of Tree were declared

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

why we did not define eval as a method of class Tree and its subclasses?

Deciding whether to use pattern matching or methods has important implications on extensibility:

- when using methods: it is easy to add a new kind of node as this can be done just by defining a sub-class of Tree for it; on the other hand, adding a new operation to manipulate the tree is tedious, as it requires modifications to all sub-classes of Tree

- when using pattern matching: the situation is reversed: adding a new kind of node requires the modification of all functions which do pattern matching on the tree, to take the new node into account; on the other hand, adding a new operation is easy, by just defining it as an independent function.

Example

Derivative Example:

1. the derivative of a sum is the sum of the derivatives
2.the derivative of some variable v is one if v is the variable relative to which the derivation takes place, and zero otherwise
3.the derivative of a constant is zero

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Example

- **the case expression for variables has a guard, an expression following the if keyword. This guard prevents pattern matching from succeeding unless its expression is true**
- **the wildcard, written _, which is a pattern matching any value, without giving it a name**

```scala
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Example

```scala
def main(args: Array[String]): Unit = {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Example

the output:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```