Advanced Programming Methods
Lecture 13 - Scala

Content

- Introduction
- Funtional Programming
- Case classes
- Pattern matching
- Traits
- Genericity
- Tuples

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

Example

```
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)
```

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))
```

Functional Programming

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

- functions are 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.

Functional Programming

```
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++)**

Functional Programming

```
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

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
}
```

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

Methods without arguments

- 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())
  }
}
```

Methods without arguments

- 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
}
```

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"
}
```

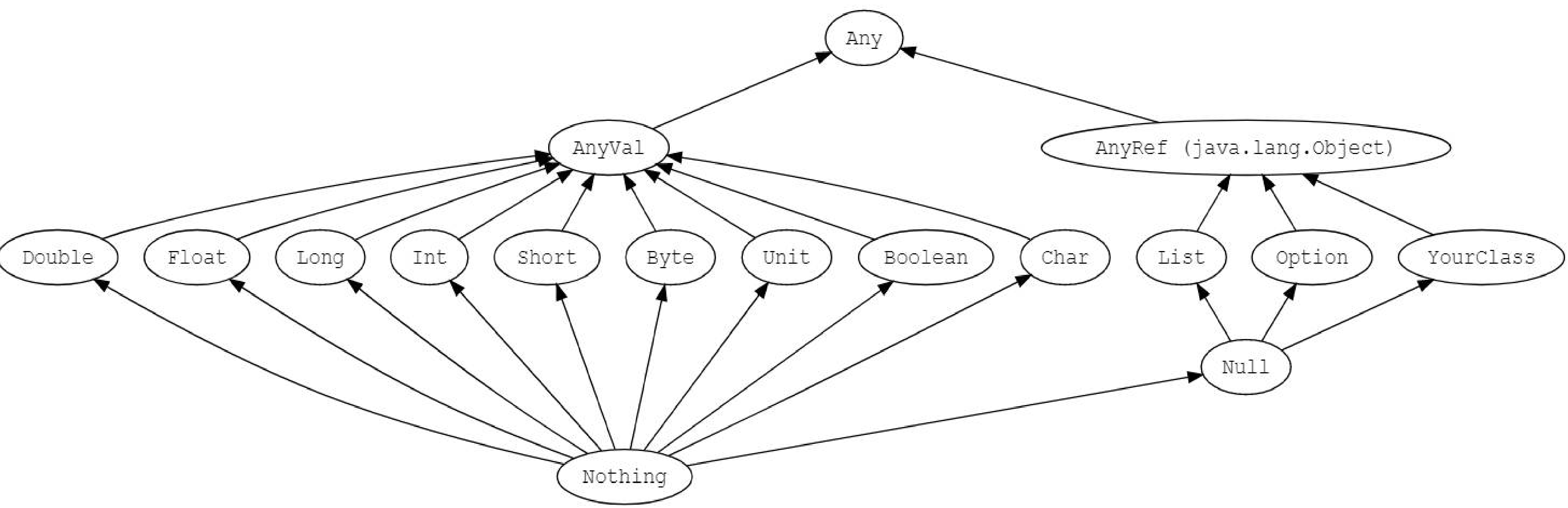
Inheritance and overriding

- **usage of the overriding method**

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

Scala Type Hierarchy

- 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

```scala
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))
```

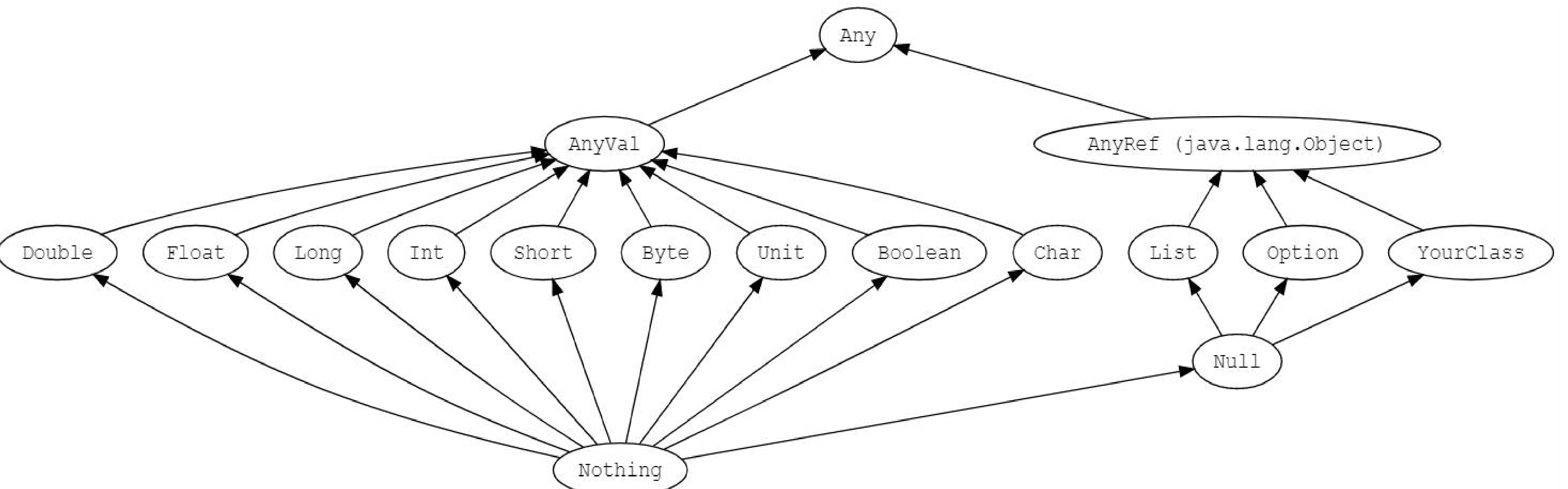
- the output is:

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

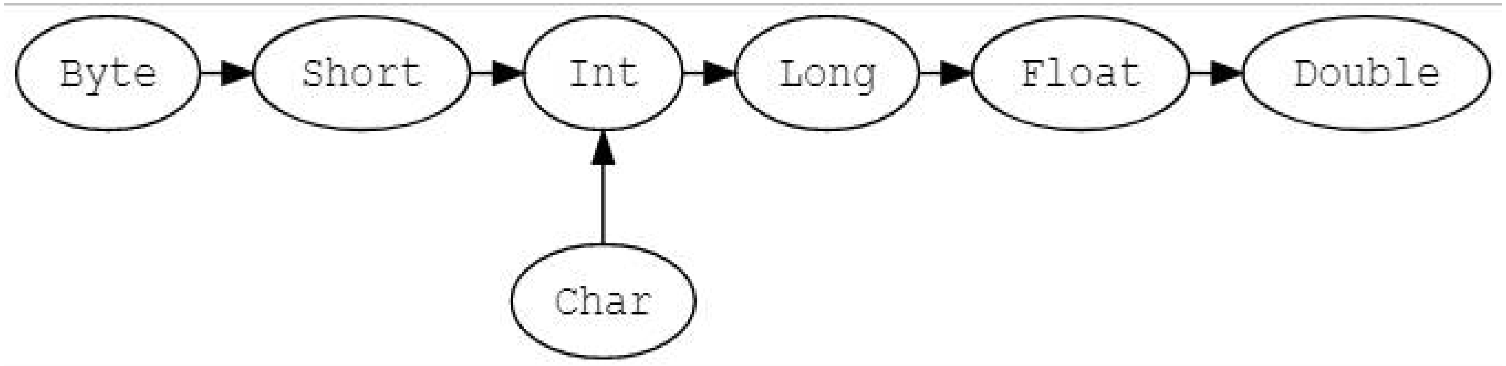
Scala Type Hierarchy

- 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
```

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 represenatation: 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

```scala
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

- **Problem:** **a 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 }
```

Pattern Matching

- 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
```

Pattern Matching

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
}
```

Pattern Matching

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
}
```

Pattern Matching

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
}
```

Pattern Matching

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
}
```

Pattern Matching

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.

Pattern Matching

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)
}
```

Pattern Matching

- **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**

```
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)
}
```

Pattern Matching

```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"))
}
```

Pattern Matching

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)))
```

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

Genericity

- the ability to write code parametrized by types

Java:

- programmers resort to using Object, which is the super-type of all objects.
-this solution is however far from being ideal, since it doesn't work for basic types (int, long, float, etc.) and it implies that a lot of dynamic type casts have to be inserted by the programmer

Genericity

- the initial value given to contents variable is _, which represents a default value.

- default value is 0 for numeric types, false for the Boolean type, () for the Unit type and null for all object types.

```
class Reference[T] {
  private var contents: T = _
  def set(value: T) { contents = value }
  def get: T = contents
}
```

Genericity

```
object IntegerReference {
  def main(args: Array[String]): Unit = {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

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

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

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

Tuples

```scala
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

- Or, in for comprehension:

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

For comprehensions

- a lightweight notation for expressing sequence comprehensions

- have the form for (enumerators) yield e, where

- enumerators refers to a semicolon-separated list of enumerators. An enumerator is either a generator which introduces new variables, or it is a filter.

- A comprehension evaluates the body e for each binding generated by the enumerators and returns a sequence of these values

For comprehensions

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

val userBase = List(User("Travis", 28),
  User("Kelly", 33),
  User("Jennifer", 44),
  User("Dennis", 23))

val twentySomethings = for (user <- userBase if (user.age >=20 && user.age < 30))
  yield user.name  // i.e. add this to a list

twentySomethings.foreach(name => println(name))  // prints Travis Dennis
```

For comprehensions

```scala
def foo(n: Int, v: Int) =
   for (i <- 0 until n;
        j <- 0 until n if i + j == v)
   yield (i, j)

foo(10, 10) foreach {
  case (i, j) =>
    println(s"($i, $j) ")  // prints (1, 9) (2, 8) (3, 7) (4, 6) (5, 5) (6, 4) (7, 3) (8, 2) (9, 1)
}
```

For comprehensions

```scala
def foo(n: Int, v: Int) =
    for (i <- 0 until n;
         j <- 0 until n if i + j == v)
    println(s"($i, $j)")

foo(10, 10)
```

Mixins

Mixins are traits which are used to compose a class

```
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
```