How Classes are Organized

Principles of Functional Programming

Packages

Classes and objects are organized in packages.

To place a class or object inside a package, use a package clause at the top of your source file.

```
package progfun.examples

object Hello
  ...
```

This would place `Hello` in the package `progfun.examples`.

You can then refer it by its *fully qualified name*, `progfun.examples.Hello`. For instance, to run the `Hello` program:

```
> scala progfun.examples.Hello
```

Imports

Say we have a class `Rational` in package `week3`.

You can use the class using its fully qualified name:

```
val r = week3.Rational(1, 2)
```

Alternatively, you can use an import:

```
import week3.Rational
val r = Rational(1, 2)
```

Forms of Imports

Imports come in several forms:

```
import week3.Rational           // imports just Rational
import week3.{Rational, Hello}  // imports both Rational and Hello
import week3._                  // imports everything in package week3
```

The first two forms are called *named imports*.

The last form is called a *wildcard import*.

You can import from either a package or an object.

Automatic Imports

Some entities are automatically imported in any Scala program.

These are:

- All members of package `scala`
- All members of package `java.lang`
- All members of the singleton object `scala.Predef`.

Here are the fully qualified names of some types and functions which you have seen so far:

`Int`	`scala.Int`
`Boolean`	`scala.Boolean`
`Object`	`java.lang.Object`
`require`	`scala.Predef.require`
`assert`	`scala.Predef.assert`

Scaladoc

You can explore the standard Scala library using the scaladoc web pages.

You can start at

www.scala-lang.org/api/current

Traits

In Java, as well as in Scala, a class can only have one superclass.

But what if a class has several natural supertypes to which it conforms or from which it wants to inherit code?

Here, you could use `traits`.

A trait is declared like an abstract class, just with `trait` instead of `abstract class`.

```
trait Planar:
  def height: Int
  def width: Int
  def surface = height * width
```

Traits (2)

Classes, objects and traits can inherit from at most one class but arbitrary many traits.

Example:

```
class Square extends Shape, Planar, Movable ...
```

Traits resemble interfaces in Java, but are more powerful because they can have parameters and can contain fields and concrete methods.

Extensions

Having to define all methods that belong to a class inside the class itself can lead to very large classes, and is not very modular.

Methods that do not need to access the internals of a class can alternatively be defined with extensions.

For instance, we can add comparison methods to class `Rational` like this:

```
extension on (x: Rational):
  def < (y: Rational): Boolean = x.numer * y.denom < y.numer * x.denom
  def > (y: Rational): Boolean = y < x
```

Extensions (2)

Extensions can also be named, e.g.

```
object Rational:
  extension RationalComparisons on (x: Rational):
    def < (y: Rational): Boolean = x.numer * y.denom < y.numer * x.denom
    def > (y: Rational): Boolean = y < x
```

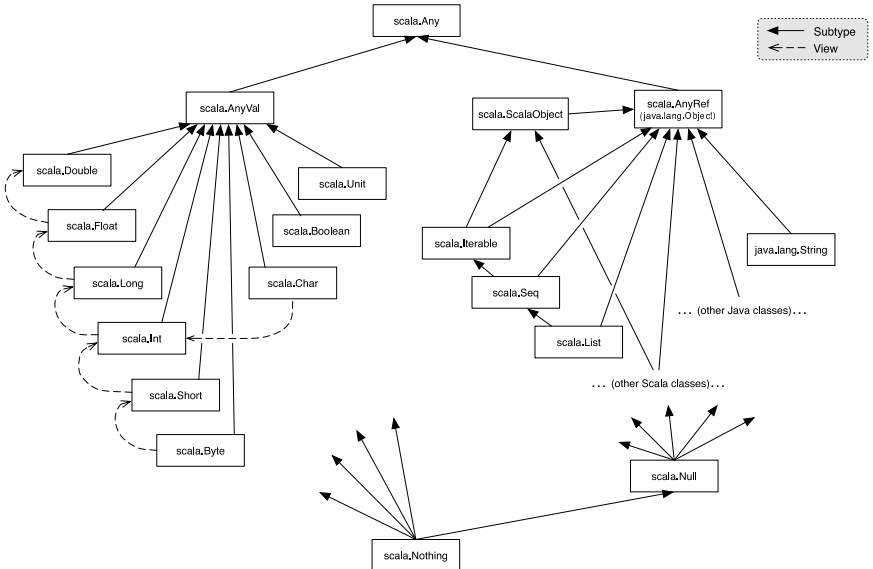
Extension of a class are visible if they are listed in the companion object of a class (as in the code above) or if they defined or imported in the current scope.

Members of a visible extension of class `C` can be called as if they were members of `C`. E.g.

```
assert(Rational(1/2) < Rational(2/3))
```

Caveat: Extensions do *not* support dynamic dispatch. An extension is selected based on the *static* type of the receiver.

Scala's Class Hierarchy



Top Types

At the top of the type hierarchy we find:

`Any`	the base type of all types Methods: ‘==‘, ‘!=‘, ‘equals‘, ‘hashCode, ‘toString‘
`AnyRef`	The base type of all reference types; Alias of ‘java.lang.Object‘
`AnyVal`	The base type of all primitive types.

The Nothing Type

`Nothing` is at the bottom of Scala's type hierarchy. It is a subtype of every other type.

There is no value of type `Nothing`.

Why is that useful?

- To signal abnormal termination
- As an element type of empty collections (see next session)

Exceptions

Scala's exception handling is similar to Java's.

The expression

```
throw Exc
```

aborts evaluation with the exception `Exc`.

The type of this expression is `Nothing`.

The Null Type

Every reference class type also has `null` as a value.

The type of `null` is `Null`.

`Null` is a subtype of every class that inherits from `Object`; it is incompatible with subtypes of `AnyVal`.

```
val x = null          // x: Null
val y: String = null // y: String
val z: Int = null     // error: type mismatch
```

The Problem With Nulls

Having `null` as a value in every reference type has been called by its inventor Tony Hoare “A Billion-Dollar Mistake”.

Why?

The Problem With Nulls

Having `null` as a value in every reference type has been called by its inventor Tony Hoare "A Billion-Dollar Mistake".

Why?

- High risk of failing at run-time with a `NullPointerException`.
- It would be safer to reflect the possibility that a reference is `null` in its static type.

Making Null Safer

It is planned to add explicit null types to Scala. I.e.

```
String            // The type of strings, no ‘null‘ allowed
String | Null     // Either a string or ‘null‘.
```

Therefore, the following would be an error:

```
def f(x: String) = println(x + "!")
f(null)     // error, expected: String, found: Null
```

Work on implementing this feature is ongoing.

Exercise

What is the type of

```
if true then 1 else false
```

- 0 `Int`
- 0 `Boolean`
- 0 `AnyVal`
- 0 `Object`
- 0 `Any`