How Classes are Organized
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.

```scala
package progfun.examples

object Hello { ... }
```

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

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

```bash
> scala progfun/examples.Hello
```
Say we have a class `Rational` in package `week3`.

You can use the class using its fully qualified name:

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

Alternatively, you can use an import:

```scala
import week3.Rational
val r = new 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td><code>scala.Int</code></td>
</tr>
<tr>
<td>Boolean</td>
<td><code>scala.Boolean</code></td>
</tr>
<tr>
<td>Object</td>
<td><code>java.lang.Object</code></td>
</tr>
<tr>
<td>require</td>
<td><code>scala.Predef.require</code></td>
</tr>
<tr>
<td>assert</td>
<td><code>scala.Predef.assert</code></td>
</tr>
</tbody>
</table>
You can explore the standard Scala library using the scaladoc web pages.

You can start at

www.scala-lang.org/api/current
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.

```scala
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:

```scala
class Square extends Shape with Planar with Movable ...
```

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

On the other hand, traits cannot have (value) parameters, only classes can.
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)
Scala’s exception handling is similar to Java’s.

The expression

```scala
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.

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

What is the type of

```java
if (true) 1 else false
```

- Int
- Boolean
- AnyVal
- Object
- Any