Class Hierarchies

Abstract Classes

Consider the task of writing a class for sets of integers with the following operations.

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

`IntSet` is an *abstract class*.

Abstract classes can contain members which are missing an implementation (in our case, `incl` and `contains`).

Consequently, no instances of an abstract class can be created with the operator `new`.

Class Extensions

Let's consider implementing sets as binary trees.

There are two types of possible trees: a tree for the empty set, and a tree consisting of an integer and two sub-trees.

Here are their implementations:

```
class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}
```

Class Extensions (2)

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {

  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true

  def incl(x: Int): IntSet =
    if (x < elem) new NonEmpty(elem, left incl x, right)
    else if (x > elem) new NonEmpty(elem, left, right incl x)
    else this
}
```

Terminology

`Empty` and `NonEmpty` both *extend* the class `IntSet`.

This implies that the types `Empty` and `NonEmpty` *conform* to the type `IntSet`

- an object of type `Empty` or `NonEmpty` can be used wherever an object of type `IntSet` is required.

Base Classes and Subclasses

`IntSet` is called the *superclass* of `Empty` and `NonEmpty`.

`Empty` and `NonEmpty` are *subclasses* of `IntSet`.

In Scala, any user-defined class extends another class.

If no superclass is given, the standard class `Object` in the Java package `java.lang` is assumed.

The direct or indirect superclasses of a class `C` are called *base classes* of `C`.

So, the base classes of `NonEmpty` are `IntSet` and `Object`.

Implementation and Overriding

The definitions of `contains` and `incl` in the classes `Empty` and `NonEmpty` *implement* the abstract functions in the base trait `IntSet`.

It is also possible to *redefine* an existing, non-abstract definition in a subclass by using `override`.

Example

```
abstract class Base {
  def foo = 1
  def bar: Int
}
```

```
class Sub extends Base {
  override def foo = 2
  def bar = 3
}
```

Object Definitions

In the `IntSet` example, one could argue that there is really only a single empty `IntSet`.

So it seems overkill to have the user create many instances of it.

We can express this case better with an *object definition*:

```
object Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, Empty, Empty)
}
```

This defines a *singleton object* named `Empty`.

No other `Empty` instances can be (or need to be) created.

Singleton objects are values, so `Empty` evaluates to itself.

Programs

So far we have executed all Scala code from the REPL or the worksheet.

But it is also possible to create standalone applications in Scala.

Each such application contains an object with a `main` method.

For instance, here is the "Hello World!" program in Scala.

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

Once this program is compiled, you can start it from the command line with

```
> scala Hello
```

Exercise

Write a method `union` for forming the union of two sets. You should implement the following abstract class.

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
  def union(other: IntSet): IntSet
}
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty contains 1
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty contains 1
```

$\rightarrow$ `[1/x] [Empty/this] false`

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty contains 1
```

$\rightarrow$ `[1/x] [Empty/this] false`

$=$ `false`

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ $[7/\texttt{elem}]$ $[7/\texttt{x}]$ $[\texttt{new NonEmpty(7, Empty, Empty)}/\texttt{this}]$

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ $[7/\texttt{elem}]$ $[7/\texttt{x}]$ $[\texttt{new NonEmpty(7, Empty, Empty)}/\texttt{this}]$

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

$=$

```
if (7 < 7) new NonEmpty(7, Empty, Empty).left contains 7
    else if (7 > 7) new NonEmpty(7, Empty, Empty).right
        contains 7 else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ $[7/\texttt{elem}]$ $[7/\texttt{x}]$ $[\texttt{new NonEmpty(7, Empty, Empty)}/\texttt{this}]$

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

$=$

```
if (7 < 7) new NonEmpty(7, Empty, Empty).left contains 7
    else if (7 > 7) new NonEmpty(7, Empty, Empty).right
        contains 7 else true
```

$\rightarrow$ `true`

Something to Ponder

Dynamic dispatch of methods is analogous to calls to higher-order functions.

Question:

Can we implement one concept in terms of the other?

- Objects in terms of higher-order functions?
- Higher-order functions in terms of objects?

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.

```
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:

```
> 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 = new week3.Rational(1, 2)
```

Alternatively, you can use an import:

```
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:

`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
}
```

Single inheritance

Traits (2)

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

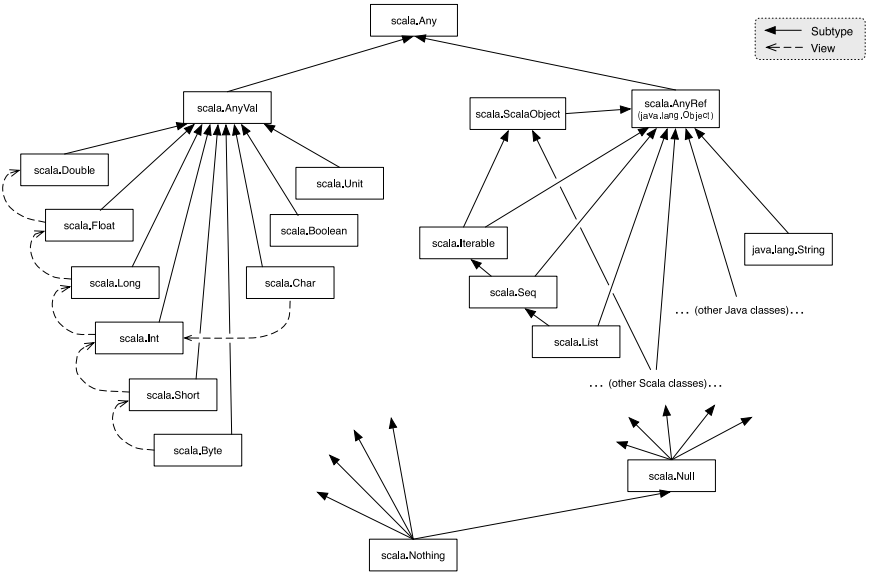
Example:

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

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

On the other hand, traits cannot have (value) parameters, only classes can.

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
```

Exercise

What is the type of

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

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

Polymorphism

Cons-Lists

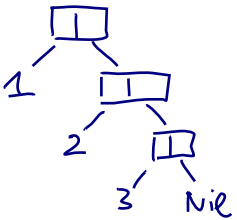
A fundamental data structure in many functional languages is the immutable linked list.

It is constructed from two building blocks:

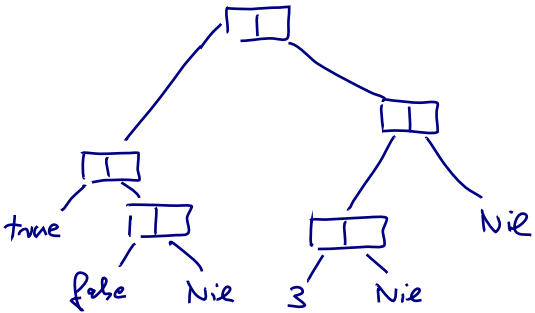
`Nil`	the empty list
`Cons`	a cell containing an element and the remainder of the list.

Examples for Cons-Lists

```
List(1, 2, 3)
```




```
List(List(true, false), List(3))
```



Cons-Lists in Scala

Here's an outline of a class hierarchy that represents lists of integers in this fashion:

```
package week4

trait IntList ...
class Cons(val head: Int, val tail: IntList) extends IntList ...
class Nil extends IntList ...
```

A list is either

- an empty list `new Nil`, or
- a list `new Cons(x, xs)` consisting of a `head` element `x` and a `tail` list `xs`.

Value Parameters

Note the abbreviation `(val head: Int, val tail: IntList)` in the definition of `Cons`.

This defines at the same time parameters and fields of a class.

It is equivalent to:

```
class Cons(_head: Int, _tail: IntList) extends IntList {
  val head = _head
  val tail = _tail
}
```

where `_head` and `_tail` are otherwise unused names.

Type Parameters

It seems too narrow to define only lists with `Int` elements.

We'd need another class hierarchy for `Double` lists, and so on, one for each possible element type.

We can generalize the definition using a type parameter:

```
package week4

trait List[T]
class Cons[T](val head: T, val tail: List[T]) extends List[T]
class Nil[T] extends List[T]
```

Type parameters are written in square brackets, e.g. `[T]`.

Complete Definition of List

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  def isEmpty = false
}

class Nil[T] extends List[T] {
  def isEmpty = true
  def head = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")
}
```

Generic Functions

Like classes, functions can have type parameters.

For instance, here is a function that creates a list consisting of a single element.

```
def singleton[T](elem: T) = new Cons[T](elem, new Nil[T])
```

We can then write:

```
singleton[Int](1)
singleton[Boolean](true)
```

Type Inference

In fact, the Scala compiler can usually deduce the correct type parameters from the value arguments of a function call.

So, in most cases, type parameters can be left out. You could also write:

```
singleton(1)
singleton(true)
```

Types and Evaluation

Type parameters do not affect evaluation in Scala.

We can assume that all type parameters and type arguments are removed before evaluating the program.

This is also called *type erasure*.

Languages that use type erasure include Java, Scala, Haskell, ML, OCaml.

Some other languages keep the type parameters around at run time, these include C++, C#, F#.

Polymorphism

Polymorphism means that a function type comes “in many forms”.

In programming it means that

- the function can be applied to arguments of many types, or
- the type can have instances of many types.

We have seen two principal forms of polymorphism:

- subtyping: instances of a subclass can be passed to a base class
- generics: instances of a function or class are created by type parameterization.

Exercise

Write a function `nth` that takes an integer `n` and a list and selects the `n`'th element of the list.

Elements are numbered from 0.

If index is outside the range from `0` up the the length of the list minus one, a `IndexOutOfBoundsException` should be thrown.