Week 3 Data and Abstraction

Class Hierarchies

Abstract classes

```
abstract class IntSet {
    def incl(x: Int): IntSet          <- not implemented
    def contains(x: Int): Boolean
}
```

Abstract classes can contain members without implementation

No instances can be created with "new" operator

Extension

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

NonEmpty: binary tree

new Empty, new Empty: left and right are empty

```
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
```

binary tree

Empty and NonEmpty both extend IntSet
they both conform to the type IntSet

An object of type Empty or NonEmpty
can be used wherever an object
of type IntSet is required

IntSet - superclass of Empty and NonEmpty

Empty and NonEmpty are subclasses of IntSet

Any user-defined class extends another class.
If no superclass is given, it extends
java.lang.Object

Base Classes - all super classes

e.g. for Empty - its IntSet and Object

The definitions of contains and incl
in Empty and NonEmpty implement the
abstract functions in the base class IntSet

It's also possible to redefine an
existing, non-abstract definition by
using override

```
abstract class Base {          class Sub extends Base {
  def foo = 1                    override def foo = 2
  def bar: Int          ⇒        def bar = 3
}                              }
```

But there's only one true Empty set -
all of them have the same behavior.
So we can create only one object

Object definition:

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

- this defines a singleton object named Empty.
- No other instances of Empty can be created
- Singleton objects are values - so Empty evaluates to itself

Programs

Standalone Scala application

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

↓ then type

```
scala Hello
```

Dynamic Binding

Scala implements dynamic method dispatch

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

(Analogous to calls to higher-order functions)

Can we implement one concept in terms of the other?

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

YES.

Organization

Packages

are used to organize classes and objects

```
package progfun.example
object Hello { ... }
```

that places Hello into progfun.example

scala progfun.example.Hello

fully qualified name

Imports

We can refer to ~~objects~~ classes using FQN

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

or use import

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

```
import week3.{Rational, Hello}   // just these 2
import week3._                   // wildcard import
```

named imports

You can import from package or from object

Some entities are imported automatically:

- java.lang
- scala
- scala.Predef

Traits

In Java and Scala a class can have only one superclass

What if a class has several natural supertypes – from which it needs to inherit code

Traits can be used

```
trait Planar {                    ← same declaration as for abstract class
  def height
  def width
  def surface = height * width
}
```

But

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

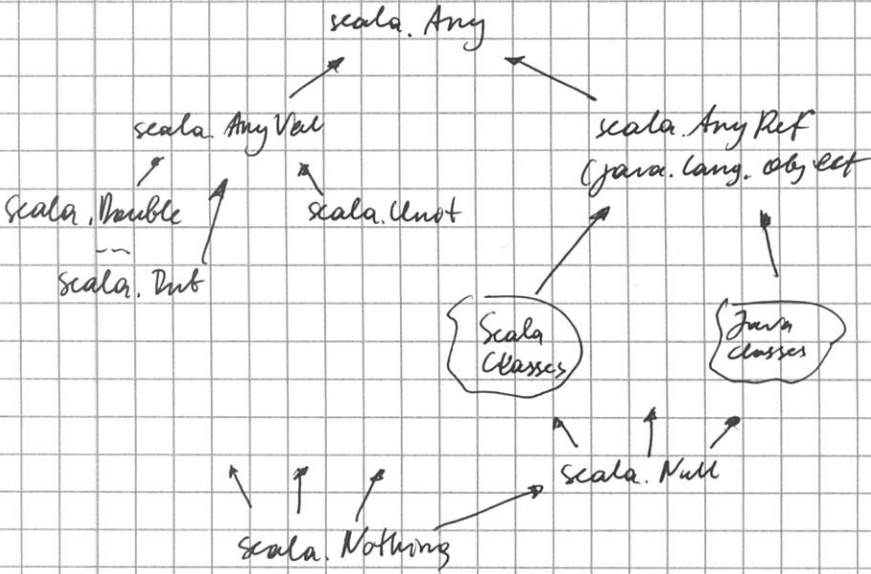
Classes and objects can inherit only from one class, but from many traits

Similar to interfaces in Java, but more powerful.

- can contain fields and
- concrete methods

But traits cannot have value parameters, only classes can

Class Hierarchy



Any – the base type of all types

methods: ==, !=, equals, hashCode, toString

AnyRef – alias of java.lang.Object

AnyVal – the base for all primitive types

The Nothing type

Nothing is at the bottom of the hierarchy. —
it's a subtype of every other type

But there is no value of this type

Can be used as an element type of empty collections

Set[Nothing]

Exceptions

similar to Java

throw Exc ← the type of this exception is Nothing

The Null Type

The type of null is Null —
a subtype of every class that inherits from Object.

```
val x = null          (x: Null)
val y: String = null  (y: String)
val z: Int = null     (error: type mismatch → only Objects)
```

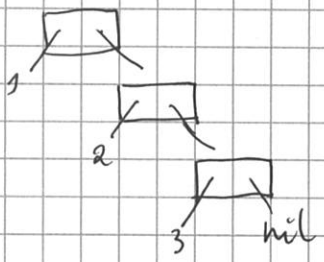
Polymorphism

Cons-lists - deep immutable linked list

two building blocks:

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

List(1, 2, 3)



Cons-lists in Scala

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

- new Nil - empty list
- new Cons(x, xs) - head x + tail xs

Value parameters

```
class Cons(val head: Int, val tail: IntList)
```


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

So it defines parameters and fields of a class

Type Parameters

too narrow to just include Ints!

But we may generalize the definition

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

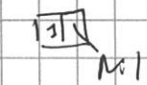
Functions, like classes, can have type parameters

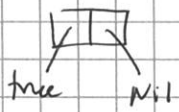
```
def singleton[T](elem: T) = new Cons[T](elem, new Nil[T])
```

↑ type parameter

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

types





but scala can infer types!

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

← parameters can be left out

(all type parameters are erased before evaluating — this is called type erasure)

Polymorphism means that a function type comes "in many forms"

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

Two principal forms:

- subtyping – instances of a subclass can be passed to a base class
- generics – instances of a function/class are created by type parametrization