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.

Objects Everywhere

Pure Object Orientation

A pure object-oriented language is one in which every value is an object.

If the language is based on classes, this means that the type of each value is a class.

Is Scala a pure object-oriented language?

At first glance, there seem to be some exceptions: primitive types, functions.

But, let's look closer:

Standard Classes

Conceptually, types such as `Int` or `Boolean` do not receive special treatment in Scala. They are like the other classes, defined in the package `scala`.

For reasons of efficiency, the Scala compiler represents the values of type `scala.Int` by 32-bit integers, and the values of type `scala.Boolean` by Java's Booleans, etc.

Pure Booleans

The `Boolean` type maps to the JVM's primitive type `boolean`.

But one *could* define it as a class from first principles:

```
package idealized.scala
abstract class Boolean {
  def ifThenElse[T](t: => T, e: => T): T

  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def unary_!: Boolean            = ifThenElse(false, true)

  def == (x: Boolean): Boolean    = ifThenElse(x, x.unary_!)
  def != (x: Boolean): Boolean    = ifThenElse(x.unary_!, x)
  ...
}
```

Boolean Constants

Here are constants `true` and `false` that go with `Boolean` in the `idealized.scala`:

```
package idealized.scala

object true extends Boolean {
  def ifThenElse[T](t: => T, e: => T) = t
}

object false extends Boolean {
  def ifThenElse[T](t: => T, e: => T) = e
}
```

Exercise

Provide an implementation of the comparison operator < in class `idealized.scala.Boolean`.

Assume for this that `false` $<$ `true`.

Exercise

Provide an implementation of the comparison operator < in class `idealized.scala.Boolean`.

Assume for this that `false` $<$ `true`.

The class Int

Here is a partial specification of the class `scala.Int`.

```
class Int {
  def + (that: Double): Double
  def + (that: Float): Float
  def + (that: Long): Long
  def + (that: Int): Int         // same for -, *, /, %

  def << (cnt: Int): Int         // same for >>, >>>  */

  def & (that: Long): Long
  def & (that: Int): Int         // same for |, ^ */
```

The class Int (2)

```
    def == (that: Double): Boolean
    def == (that: Float): Boolean
    def == (that: Long): Boolean   // same for !=, <, >, <=, >=
    ...
  }
```

Can it be represented as a class from first principles (i.e. not using primitive `int`s?

Exercise

Provide an implementation of the abstract class `Nat` that represents non-negative integers.

```
abstract class Nat {
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def + (that: Nat): Nat
  def - (that: Nat): Nat
}
```

Exercise (2)

Do not use standard numerical classes in this implementation.

Rather, implement a sub-object and a sub-class:

```
object Zero extends Nat
class Succ(n: Nat) extends Nat
```

One for the number zero, the other for strictly positive numbers.

(this one is a bit more involved than previous quizzes).

Functions as Objects

Functions as Objects

We have seen that Scala's numeric types and the `Boolean` type can be implemented like normal classes.

But what about functions?

Functions as Objects

We have seen that Scala's numeric types and the `Boolean` type can be implemented like normal classes.

But what about functions?

In fact function values *are* treated as objects in Scala.

The function type `A => B` is just an abbreviation for the class `scala.Function1[A, B]`, which is defined as follows.

```
package scala
trait Function1[A, B] {
  def apply(x: A): B
}
```

So functions are objects with `apply` methods.

There are also traits `Function2`, `Function3`, ... for functions which take more parameters (currently up to 22).

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

```
{ class AnonFun extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFun
}
```

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

```
{ class AnonFun extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFun
}
```

or, shorter, using *anonymous class syntax*:

```
new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
```

Expansion of Function Calls

A function call, such as `f(a, b)`, where `f` is a value of some class type, is expanded to

```
f.apply(a, b)
```

So the OO-translation of

```
val f = (x: Int) => x * x
f(7)
```

would be

```
val f = new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
f.apply(7)
```

Functions and Methods

Note that a method such as

```
def f(x: Int): Boolean = ...
```

is not itself a function value.

But if `f` is used in a place where a Function type is expected, it is converted automatically to the function value

```
(x: Int) => f(x)
```

or, expanded:

```
new Function1[Int, Boolean] {
  def apply(x: Int) = f(x)
}
```

Exercise

In package `week4`, define an

```
object List {
  ...
}
```

with 3 functions in it so that users can create lists of lengths 0-2 using syntax

```
List()          // the empty list
List(1)         // the list with single element 1
List(2, 3)      // the list with elements 2 and 3.
```

Subtyping and Generics

Polymorphism

Two principal forms of polymorphism:

- subtyping
- generics

In this session we will look at their interactions.

Two main areas:

- bounds
- variance

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

```
def assertAllPos(s: IntSet): IntSet
```

In most situations this is fine, but can one be more precise?

Type Bounds

One might want to express that `assertAllPos` takes `Empty` sets to `Empty` sets and `NonEmpty` sets to `NonEmpty` sets.

A way to express this is:

```
def assertAllPos[S <: IntSet](r: S): S = ...
```

Here, "`<: IntSet`" is an *upper bound* of the type parameter `S`:

It means that `S` can be instantiated only to types that conform to `IntSet`.

Generally, the notation

- `S <: T` means: `S` *is a subtype of* `T`, and
- `S >: T` means: `S` *is a supertype of* `T`, or `T` *is a subtype of* `S`.

Lower Bounds

You can also use a lower bound for a type variable.

Example

```
[S >: NonEmpty]
```

introduces a type parameter `S` that can range only over *supertypes* of `NonEmpty`.

So `S` could be one of `NonEmpty`, `IntSet`, `AnyRef`, or `Any`.

We will see later on in this session where lower bounds are useful.

Mixed Bounds

Finally, it is also possible to mix a lower bound with an upper bound.

For instance,

```
    [S >: NonEmpty <: IntSet]
```

would restrict `S` any type on the interval between `NonEmpty` and `IntSet`.

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
   NonEmpty <: IntSet
```

is

```
   List[NonEmpty] <: List[IntSet]    ?
```

Intuitively, this makes sense: A list of non-empty sets is a special case of a list of arbitrary sets.

We call types for which this relationship holds *covariant* because their subtyping relationship varies with the type parameter.

Does covariance make sense for all types, not just for `List`?

Arrays

For perspective, let's look at arrays in Java (and C#).

Reminder:

- An array of `T` elements is written `T[]` in Java.
- In Scala we use parameterized type syntax `Array[T]` to refer to the same type.

Arrays in Java are covariant, so one would have:

```
NonEmpty[] <: IntSet[]
```

Array Typing Problem

But covariant array typing causes problems.

To see why, consider the Java code below.

```
NonEmpty[] a = new NonEmpty[]{new NonEmpty(1, Empty, Empty)}
IntSet[] b = a
b[0] = Empty
NonEmpty s = a[0]
```

It looks like we assigned in the last line an `Empty` set to a variable of type `NonEmpty`!

What went wrong?

The Liskov Substitution Principle

The following principle, stated by Barbara Liskov, tells us when a type can be a subtype of another.

> *If* `A <: B`*, then everything one can to do with a value of type* `B` *one should also be able to do with a value of type* `A`*.*

[The actual definition Liskov used is a bit more formal. It says:

> *Let* `q(x)` *be a property provable about objects* `x` *of type* `B`*. Then* `q(y)` *should be provable for objects* `y` *of type* `A` *where* `A <: B`*.*

]

Exercise

The problematic array example would be written as follows in Scala:

```
val a: Array[NonEmpty] = Array(new NonEmpty(1, Empty, Empty))
val b: Array[IntSet] = a
b(0) = Empty
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

O	A type error in line 1
O	A type error in line 2
O	A type error in line 3
O	A type error in line 4
O	A program that compiles and throws an exception at run-time
O	A program that compiles and runs without exception

Exercise

The problematic array example would be written as follows in Scala:

```
val a: Array[NonEmpty] = Array(new NonEmpty(1, Empty, Empty))
val b: Array[IntSet] = a
b(0) = Empty
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

O	A type error in line 1
O	A type error in line 2
O	A type error in line 3
O	A type error in line 4
O	A program that compiles and throws an exception at run-time
O	A program that compiles and runs without exception

Variance

September 29, 2012

Variance

You have seen the the previous session that some types should be covariant whereas others should not.

Roughly speaking, a type that accepts mutations of its elements should not be covariant.

But immutable types can be covariant, if some conditions on methods are met.

Definition of Variance

Say `C[T]` is a parameterized type and `A`, `B` are types such that `A <: B`.

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`C[A] <: C[B]`	`C` is *covariant*
`C[A] >: C[B]`	`C` is *contravariant*
neither `C[A]` nor `C[B]` is a subtype of the other	`C` is *nonvariant*

Definition of Variance

Say `C[T]` is a parameterized type and `A`, `B` are types such that `A <: B`.

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`C[A] <: C[B]`	`C` is *covariant*
`C[A] >: C[B]`	`C` is *contravariant*
neither `C[A]` nor `C[B]` is a subtype of the other	`C` is *nonvariant*

Scala lets you declare the variance of a type by annotating the type parameter:

`class C[+A] { ... }`	`C` is *covariant*
`class C[-A] { ... }`	`C` is *contravariant*
`class C[A] { ... }`	`C` is *nonvariant*

Exercise

Say you have two function types:

```
type A = IntSet => NonEmpty
type B = NonEmpty => IntSet
```

According to the Liskov Substitution Principle, which of the following should be true?

```
O          A <: B
O          B <: A
O          A and B are unrelated.
```

Exercise

Say you have two function types:

```
type A = IntSet => NonEmpty
type B = NonEmpty => IntSet
```

According to the Liskov Substitution Principle, which of the following should be true?

```
O          A <: B
O          B <: A
O          A and B are unrelated.
```

Typing Rules for Functions

Generally, we have the following rule for subtyping between function types:

If `A2 <: A1` and `B1 <: B2`, then

```
A1 => B1  <:  A2 => B2
```

Function Trait Declaration

So functions are *contravariant* in their argument type(s) and *covariant* in their result type.

This leads to the following revised definition of the `Function1` trait:

```
package scala
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Variance Checks

We have seen in the array example that the combination of covariance with certain operations is unsound.

In this case the problematic operation was the update operation on an array.

If we turn `Array` into a class, and `update` into a method, it would look like this:

```
class Array[+T] {
  def update(x: T) ...
}
```

The problematic combination is

- the covariant type parameter `T`
- which appears in parameter position of the method `update`.

Variance Checks (2)

The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations.

Roughly,

- *covariant* type parameters can only appear in method results.
- *contravariant* type parameters can only appear in method parameters.
- *invariant* type parameters can appear anywhere.

The precise rules are a bit more involved, fortunately the Scala compiler performs them for us.

Variance-Checking the Function Trait

Let's have a look again at Function1:

```
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Here,

- `T` is contravariant and appears only as a method parameter type
- `U` is covariant and appears only as a method result type

So the method is checks out OK.

Variance and Lists

Let’s get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Variance and Lists

Let's get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Here are the essential modifications:

```
trait List[+T] { ... }
object Empty extends List[Nothing] { ... }
```

Making Classes Covariant

Sometimes, we have to put in a bit of work to make a class covariant.

Consider adding a `prepend` method to `List` which prepends a given element, yielding a new list.

A first implementation of `prepend` could look like this:

```
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

But that does not work!

Exercise

Why does the following code not type-check?

```
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

Possible answers:

O prepend turns List into a mutable class.

O prepend fails variance checking.

O prepend's right-hand side contains a type error.

Exercise

Why does the following code not type-check?

```
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

Possible answers:

O `prepend` turns `List` into a mutable class.

O `prepend` fails variance checking.

O `prepend`'s right-hand side contains a type error.

Prepend Violates LSP

Indeed, the compiler is right to throw out `List` with `prepend`, because it violates the Liskov Substitution Principle:

Here's something one can do with a list `xs` of type `List[IntSet]`:

```
xs.prepend(Empty)
```

But the same operation on a list `ys` of type `List[NonEmpty]` would lead to a type error:

```
ys.prepend(Empty)
           ^ type mismatch
           required: NonEmpty
           found: Empty
```

So, `List[NonEmpty]` cannot be a subtype of `List[IntSet]`.

Lower Bounds

But `prepend` is a natural method to have on immutable lists!

Question: How can we make it variance-correct?

Lower Bounds

But `prepend` is a natural method to have on immutable lists!

Question: How can we make it variance-correct?

We can use a *lower bound*:

```
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

This passes variance checks, because:

- covariant type parameters may appear in lower bounds of method type parameters
- contravariant type parameters may appear in upper bounds of method

Exercise

Implement `prepend` as shown in trait `List`.

```
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

What is the result type of this function:

```
def f(xs: List[NonEmpty], x: Empty) = xs prepend x   ?
```

Possible answers:

O	does not type check
O	`List[NonEmpty]`
O	`List[Empty]`
O	`List[IntSet]`
O	`List[Any]`

Exercise

Implement `prepend` as shown in trait `List`.

```
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

What is the result type of this function:

```
def f(xs: List[NonEmpty], x: Empty) = xs prepend x   ?
```

Possible answers:

O	does not type check
O	`List[NonEmpty]`
O	`List[Empty]`
O	`List[IntSet]`
O	`List[Any]`

Decomposition

Decomposition

Suppose you want to write a small interpreter for arithmetic expressions.

To keep it simple, let's restrict ourselves to numbers and additions.

Expressions can be represented as a class hierarchy, with a base trait `Expr` and two subclasses, `Number` and `Sum`.

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us to the following implementation.

Expressions

```
trait Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = throw new Error(”Number.leftOp”)
  def rightOp: Expr = throw new Error(”Number.rightOp”)
}
```

Expressions (2)

```
class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = throw new Error(”Sum.numValue”)
  def leftOp: Expr = e1
  def rightOp: Expr = e2

}
```

Evaluation of Expressions

You can now write an evaluation function as follows.

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else throw new Error(”Unknown expression ” + e)
}
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

```
class Prod(e1: Expr, e2: Expr) extends Expr   // e1 * e2
class Var(x: String) extends Expr             // Variable ‘x’
```

You need to add methods for classification and access to all classes defined above.

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

- O 9
- O 10
- O 19
- O 25
- O 35
- O 40

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

- 9
- 10
- 19
- 25
- 35
- 40

Non-Solution: Type Tests and Type Casts

A “hacky” solution could use type tests and type casts.

Scala let’s you do these using methods defined in class `Any`:

```
def isInstanceOf[T]: Boolean  // checks whether this object’s type conforms to ‘T‘
def asInstanceOf[T]: T        // treats this object as an instance of type ‘T‘
                              // throws ‘ClassCastException‘ if it isn’t.
```

These correspond to Java’s type tests and casts

```
Scala                   Java

x.isInstanceOf[T]       x instanceof T
x.asInstanceOf[T]       (T) x
```

But their use in Scala is discouraged, because there are better alternatives.

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if (e.isInstanceOf[Number])
    e.asInstanceOf[Number].numValue
  else if (e.isInstanceOf[Sum])
    eval(e.asInstanceOf[Sum].leftOp) +
    eval(e.asInstanceOf[Sum].rightOp)
  else throw new Error(”Unknown expression ” + e)
```

Assessment of this solution:

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if (e.isInstanceOf[Number])
    e.asInstanceOf[Number].numValue
  else if (e.isInstanceOf[Sum])
    eval(e.asInstanceOf[Sum].leftOp) +
    eval(e.asInstanceOf[Sum].rightOp)
  else throw new Error(”Unknown expression ” + e)
```

Assessment of this solution:

+ no need for classification methods, access methods only for classes where the value is defined.
– low-level and potentially unsafe.

Solution 1: Object-Oriented Decomposition

For example, suppose that all you want to do is *evaluate* expressions.

You could then define:

```
trait Expr {
  def eval: Int
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

But what happens if you'd like to display expressions now?

You have to define new methods in all the subclasses.

Limitations of OO Decomposition

And what if you want to simplify the expressions, say using the rule:

```
a * b + a * c   ->   a * (b + c)
```

Problem: This is a non-local simplification. It cannot be encapsulated in the method of a single object.

You are back to square one; you need test and access methods for all the different subclasses.

Pattern Matching

Reminder: Decomposition

The task we are trying to solve is find a general and convenient way to access objects in a extensible class hierarchy.

Attempts seen previously:

- *Classification and access methods*: quadratic explosion
- *Type tests and casts*: unsafe, low-level
- *Object-oriented decomposition*: does not always work, need to touch all classes to add a new method.

Solution 2: Functional Decomposition with Pattern Matching

Observation: the sole purpose of test and accessor functions is to *reverse* the construction process:

- Which subclass was used?
- What were the arguments of the constructor?

This situation is so common that many functional languages, Scala included, automate it.

Case Classes

A *case class* definition is similar to a normal class definition, except that it is preceded by the modifier `case`. For example:

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Like before, this defines a trait `Expr`, and two concrete subclasses `Number` and `Sum`.

Case Classes (2)

It also implicitly defines companion objects with `apply` methods.

```
object Number {
  def apply(n: Int) = new Number(n)
}
object Sum {
  def apply(e1: Expr, e2: Expr) = new Sum(e1, e2)
}
```

so you can write `Number(1)` instead of `new Number(1)`.

However, these classes are now empty. So how can we access the members?

Pattern Matching

Pattern matching is a generalization of `switch` from C/Java to class hierarchies.

It's expressed in Scala using the keyword `match`.

Example

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

Match Syntax

Rules:

- `match` is followed by a sequence of *cases*, `pat => expr`.
- Each case associates an *expression* `expr` with a *pattern* `pat`.
- A `MatchError` exception is thrown if no pattern matches the value of the selector.

Forms of Patterns

Patterns are constructed from:

- *constructors*, e.g. `Number`, `Sum`,
- *variables*, e.g. `n`, `e1`, `e2`,
- *wildcard patterns* `_`,
- *constants*, e.g. `1`, `true`.

Variables always begin with a lowercase letter.

The same variable name can only appear once in a pattern. So, `Sum(x, x)` is not a legal pattern.

Names of constants begin with a capital letter, with the exception of the reserved words `null`, `true`, `false`.

Evaluating Match Expressions

An expression of the form

$$\texttt{e match } \{ \texttt{ case } p_1 => e_1 \ldots \texttt{ case } p_n => e_n \}$$

matches the value of the selector `e` with the patterns $p_1, \ldots, p_n$ in the order in which they are written.

The whole match expression is rewritten to the right-hand side of the first case where the pattern matches the selector *e*.

References to pattern variables are replaced by the corresponding parts in the selector.

What Do Patterns Match?

- A constructor pattern `C`$(p_1, \ldots, p_n)$ matches all the values of type `C` (or a subtype) that have been constructed with arguments matching the patterns $p_1, \ldots, p_n$.
- A variable pattern `x` matches any value, and *binds* the name of the variable to this value.
- A constant pattern `c` matches values that are equal to `c` (in the sense of ==)

Example

Example

```
eval(Sum(Number(1), Number(2)))
```

$\rightarrow$

```
Sum(Number(1), Number(2)) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

$\rightarrow$

```
eval(Number(1)) + eval(Number(2))
```

Example (2)

$\rightarrow$

```
Number(1) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
} + eval(Number(2))
```

$\rightarrow$

```
1 + eval(Number(2))
```

$\twoheadrightarrow$

```
3
```

Pattern Matching and Methods

Of course, it's also possible to define the evaluation function as a method of the base trait.

Example

```
trait Expr {
  def eval: Int = this match {
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
  }
}
```

Exercise

Write a function `show` that uses pattern matching to return the representation of a given expressions as a string.

```
def show(e: Expr): String = ???
```

Exercise (Optional, Harder)

Add case classes `Var` for variables `x` and `Prod` for products `x * y` as discussed previously.

Change your `show` function so that it also deals with products.

Pay attention you get operator precedence right but to use as few parentheses as possible.

Example

```
Sum(Prod(2, Var("x")), Var("y"))
```

should print as "`2 * x + y`". But

```
Prod(Sum(2, Var("x")), Var("y"))
```

should print as "`(2 + x) * y`".