Classes vs. Modules

Outline

Outline

Complementary tools

Module system

The notion of *module* is taken seriously

- ⊕ Abstraction-based assembling language of structures
- ⊖ It does not help extensibility (unless it is by unrelated parts), does not love recursion

Class-based OOP

The notion of *extensibility* is taken seriously

- ⊕ Horizontally by adding new classes, vertically by inheritance
- ⊕ Value abstraction is obtained by hiding some components
- ⊖ Pretty rigid programming style, difficult to master because of late binding.

Modularity in OOP and ML

A three-layered framework

1. Interfaces
2. Classes
3. Objects

Modularity in OOP and ML

A three-layered framework

1. Interfaces
2. Classes
3. Objects

ML Modules

The intermediate layer (classes) is absent in ML module systems

Modularity in OOP and ML

A three-layered framework

1. Interfaces
2. Classes
3. Objects

ML Modules

The intermediate layer (classes) is absent in ML module systems

This intermediate layer makes it possible to

1. Bind operations to instances
2. Specialize and redefine operations for new instances

Modularity in OOP and ML

A three-layered framework

1. Interfaces
2. Classes
3. Objects

ML Modules

The intermediate layer (classes) is absent in ML module systems

This intermediate layer makes it possible to

1. Bind operations to instances
2. Specialize and redefine operations for new instances

Rationale

Objects can be seen as a generalization of “references” obtained by *tightly coupling* them with their operators

An example in Scala

```
trait Vector {
  def norm() : Double                                //declared method
  def isOrigin (): Boolean = (this.norm == 0)   // defined method
}
```

Like a Java interface but you can also give the definition of some methods.

When defining an instance of Vector I need only to specify `norm`

An example in Scala

```
trait Vector {
  def norm() : Double                                //declared method
  def isOrigin (): Boolean = (this.norm == 0)   // defined method
}
```

Like a Java interface but you can also give the definition of some methods.

When defining an instance of Vector I need only to specify `norm`:

```
class Point(a: Int, b: Int) extends Vector {
  var x: Int = a                          // mutable instance variable
  var y: Int = b                          // mutable instance variable
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))        // method
  def erase(): Point = { x = 0; y = 0; return this }    // method
  def move(dx: Int): Point = new Point(x+dx,y)          // method
}
```

An example in Scala

```
trait Vector {
  def norm() : Double                            //declared method
  def isOrigin (): Boolean = (this.norm == 0)    // defined method
}
```

Like a Java interface but you can also give the definition of some methods.
When defining an instance of Vector I need only to specify `norm`:

```
class Point(a: Int, b: Int) extends Vector {
  var x: Int = a                          // mutable instance variable
  var y: Int = b                          // mutable instance variable
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))          // method
  def erase(): Point = { x = 0; y = 0; return this }      // method
  def move(dx: Int): Point = new Point(x+dx,y)            // method
}
```

```
scala> new Point(1,1).isOrigin
res0: Boolean = false
```

Equivalently

```
class Point(a: Int, b: Int) {
  var x: Int = a                          // mutable instance variable
  var y: Int = b                          // mutable instance variable
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))        // method
  def erase(): Point = { x = 0; y = 0; return this }    // method
  def move(dx: Int): Point = new Point(x+dx,y)          // method
  def isOrigin(): Boolean = (this.norm == 0)            // method
}
```

Equivalently

```
class Point(a: Int, b: Int) {
  var x: Int = a                          // mutable instance variable
  var y: Int = b                          // mutable instance variable
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))         // method
  def erase(): Point = { x = 0; y = 0; return this }     // method
  def move(dx: Int): Point = new Point(x+dx,y)           // method
  def isOrigin(): Boolean = (this.norm == 0)             // method
}
```

Equivalently? Not really:

```
class PolarPoint(norm: Double, theta: Double) extends Vector {
  var norm: Double = norm
  var theta: Double = theta
  def norm(): Double = return norm
  def erase(): PolarPoint = { norm = 0 ; return this }
}
```

Can use instances of both `PolarPoint` and `Point` (first definition but not the second) where an object of type `Vector` is expected.

Inheritance

```
class Point(a: Int, b: Int) {
  var x: Int = a
  var y: Int = b
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))
  def erase(): Point = { x = 0; y = 0; return this }
  def move(dx: Int): Point = new Point(x+dx,y)
  def isOrigin(): Boolean = (this.norm == 0)
}

class ColPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c           // non-mutable instance variable
  def isWhite(): Boolean = c == "white"
  override def norm(): Double = {
    if (this.isWhite) return 0 else return sqrt(pow(x,2)+pow(y,2))
  }
  override def move(dx: Int): ColPoint=new ColPoint(x+dx,y,"red")
}
```

`isWhite` added; `erase`, `isOrigin` inherited; `move`, `norm` overridden. Notice the late binding of `norm` in `isOrigin`.

Late binding of `norm`

```
scala> new ColPoint( 1, 1, "white").isOrigin
res1: Boolean = true
```

the method defined in Point is executed but `norm` is dynamically bound to the definition in `ColPoint`.

Role of each construction

Traits (interfaces): Traits are similar to *recursive record types* and make it possible to range on objects with common methods with compatible types but incompatible implementations.

```
type Vector = { norm: Double ,         // actually unit -> Double
                erase: Vector ,        // actually unit -> Vector
                isOrigin: Boolean      // actually unit -> Boolean
              }
```

Both `Point` and `PolarPoint` have the type above, but only if explicitly declared in the class (name subtyping: an explicit design choice to avoid unwanted interactions).

Role of each construction

Traits (interfaces): Traits are similar to *recursive record types* and make it possible to range on objects with common methods with compatible types but incompatible implementations.

```
type Vector = { norm: Double ,          // actually unit -> Double
                erase: Vector ,         // actually unit -> Vector
                isOrigin: Boolean       // actually unit -> Boolean
              }
```

Both `Point` and `PolarPoint` have the type above, but only if explicitly declared in the class (name subtyping: an explicit design choice to avoid unwanted interactions).

Classes: Classes are object templates in which instance variables are declared and the semantics of `this` is open (late binding).

Role of each construction

Traits (interfaces): Traits are similar to *recursive record types* and make it possible to range on objects with common methods with compatible types but incompatible implementations.

```
type Vector = { norm: Double ,          // actually unit -> Double
                erase: Vector ,         // actually unit -> Vector
                isOrigin: Boolean       // actually unit -> Boolean
               }
```

Both `Point` and `PolarPoint` have the type above, but only if explicitly declared in the class (name subtyping: an explicit design choice to avoid unwanted interactions).

Classes: Classes are object templates in which instance variables are declared and the semantics of `this` is open (late binding).

Objects: Objects are instances of classes in which variables are given values and the semantic of `this` is bound to the object itself.

Late-binding and inheritance

The tight link between objects and their methods is embodied by *late-binding*

Late-binding and inheritance

The tight link between objects and their methods is embodied by *late-binding*

Example

```
class A {
   def m1() = {.... this.m2() ...}
   def m2() = {...}
}

class B extends A {
   def m3() = {... this.m2() ...}
   override def m2() = {...}                 //overriding
}
```

Late-binding and inheritance

The tight link between objects and their methods is embodied by *late-binding*

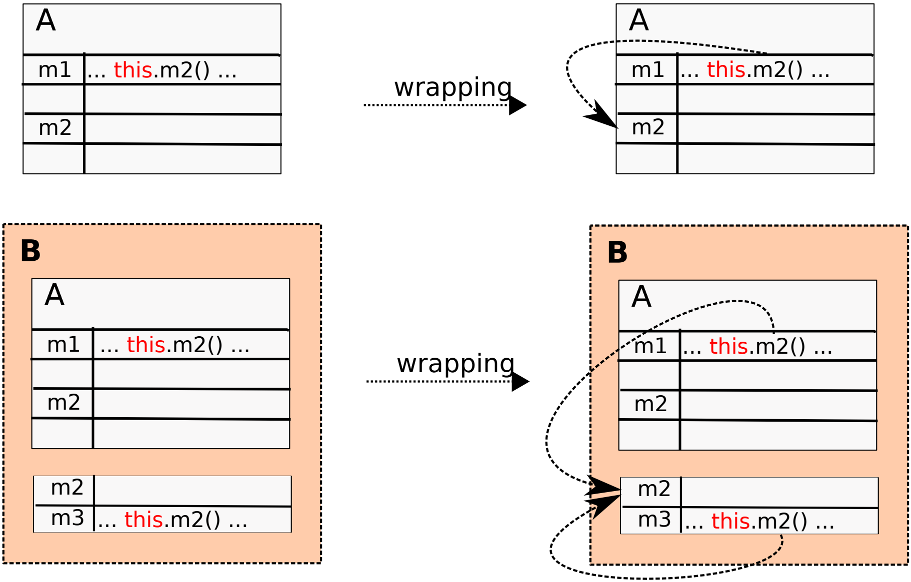
Example

```
class A {
   def m1() = {.... this.m2() ...}
   def m2() = {...}
}

class B extends A {
   def m3() = {... this.m2() ...}
   override def m2() = {...}                 //overriding
}
```

Two different behaviors according to whether late binding is used or not

Graphical representation



FP and OOP

- FP is a more operation-oriented style of programming
- OOP is a more state-oriented style of programming

FP and OOP

- FP is a more operation-oriented style of programming
- OOP is a more state-oriented style of programming
- Modules and Classes+Interfaces are the respective tools for “programming in the large” and accounting for software evolution

Software evolution

Classes and modules are not necessary for small non evolving programs (except to support separate compilation)

Software evolution

Classes and modules are not necessary for small non evolving programs (except to support separate compilation)

They are significant for software that

- should remain extensible over time
  (e.g. add support for new target processor in a compiler)
- is intended as a framework or set of components to be (re)used in larger programs
  (e.g. libraries, toolkits)

Adapted to different kinds of extensions

Instances of programmer nightmares

- Try to modify the type-checking algorithm in the Java Compiler
- Try to add a new kind of account, (e.g. an equity portfolio account) to the example given for functors (see Example Chapter 14 OReilly book).

Adapted to different kinds of extensions

Instances of programmer nightmares

- Try to modify the type-checking algorithm in the Java Compiler
- Try to add a new kind of account, (e.g. an equity portfolio account) to the example given for functors (see Example Chapter 14 OReilly book).

	FP approach	**OO approach**
Adding a new kind of things	Must edit all functions, by adding a new case to every pattern matching	Add one class (the other classes are unchanged)
Adding a new operation over things	Add one function (the other functions are unchanged)	Must edit all classes by adding or modifying methods in every class

Summary

Modules and classes play different roles:

- Modules handle type abstraction and parametric definitions of abstractions (functors)
- Classes do not provide this type abstraction possibility
- Classes provide late binding and inheritance (and message passing)

It is no shame to use both styles and combine them in order to have the possibilities of each one

Summary

Which one should I choose?

- *Any* of them when both are possible for the problem at issue
- *Classes* when you need late binding
- *Modules* if you need abstract types that share implementation (e.g. vectors and matrices)
- *Both* in several cases.

Summary

Which one should I choose?

- *Any* of them when both are possible for the problem at issue
- *Classes* when you need late binding
- *Modules* if you need abstract types that share implementation (e.g. vectors and matrices)
- ***Both* in several cases.**

Summary

Which one should I choose?

- *Any* of them when both are possible for the problem at issue
- *Classes* when you need late binding
- *Modules* if you need abstract types that share implementation (e.g. vectors and matrices)
- ***Both* in several cases.**

Trend

The frontier between modules and classes gets fussier and fuzzier

Not a clear-cut difference

- Mixin Composition
- Multiple dispatch languages
- OCaml Classes
- Haskell's type classes

Not a clear-cut difference

- Mixin Composition
- Multiple dispatch languages
- OCaml Classes
- Haskell's type classes

Let us have a look to each point

Outline

Mixin Class Composition

Reuse the new member definitions of a class (i.e., the delta in relationship to the superclass) in the definition of a new class. In Scala:

```
abstract class AbsIterator {
  type T                                   // opaque type as in OCaml Modules
  def hasNext: Boolean
  def next: T
}
```

Abstract class (as in Java we cannot instantiate it). Next define an interface (`trait` in Scala: unlike Java traits may specify the implementation of some methods; unlike abstract classes traits cannot interoperate with Java)

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) { while (hasNext) f(next) } // higher-order
}
```

Mixin Class Composition

Reuse the new member definitions of a class (i.e., the delta in relationship to the superclass) in the definition of a new class. In Scala:

```
abstract class AbsIterator {
  type T                              // opaque type as in OCaml Modules
  def hasNext: Boolean
  def next: T
}
```

Abstract class (as in Java we cannot instantiate it). Next define an interface (`trait` in Scala: unlike Java traits may specify the implementation of some methods; unlike abstract classes traits cannot interoperate with Java)

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) { while (hasNext) f(next) } // higher-order
}
```

A concrete iterator class, which returns successive characters of a string:

```
class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s charAt i; i += 1; ch }
}
```

Cannot combine the functionality of StringIterator and RichIterator into a single class by single inheritance (as both classes contain member impementations with code). Mixin-class composition (keyword `with`): reuse the delta of a class definition (i.e., all new definitions that are not inherited)

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator //mixin
    val iter = new Iter
    iter.foreach(println) }
}
```

Cannot combine the functionality of StringIterator and RichIterator into a single class by single inheritance (as both classes contain member impementations with code). Mixin-class composition (keyword `with`): reuse the delta of a class definition (i.e., all new definitions that are not inherited)

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator //mixin
    val iter = new Iter
    iter.foreach(println) }
}
```

Extends the "superclass" `StringIterator` with `RichIterator`'s methods that are not inherited from `AbsIterator`: `foreach` but not `next` or `hasNext`.

Cannot combine the functionality of StringIterator and RichIterator into a single class by single inheritance (as both classes contain member impementations with code). Mixin-class composition (keyword `with`): reuse the delta of a class definition (i.e., all new definitions that are not inherited)

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator //mixin
    val iter = new Iter
    iter.foreach(println) }
}
```

Extends the “superclass” `StringIterator` with `RichIterator`’s methods that are not inherited from `AbsIterator`: `foreach` but not `next` or `hasNext`. Note that the last application works since `println : Any => Unit`:

```
scala> def test (x : Any => Unit) = x      // works also if we replace
test: ((Any) => Unit)(Any) => Unit        // Any by a different type

scala> test(println)
res0: (Any) => Unit = <function>
```

Cannot combine the functionality of StringIterator and RichIterator into a single class by single inheritance (as both classes contain member impementations with code). Mixin-class composition (keyword `with`): reuse the delta of a class definition (i.e., all new definitions that are not inherited)

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator //mixin
    val iter = new Iter
    iter.foreach(println) }
}
```

Extends the "superclass" `StringIterator` with `RichIterator`'s methods that are not inherited from `AbsIterator`: `foreach` but not `next` or `hasNext`. Note that the last application works since `println : Any => Unit`:

```
scala> def test (x : Any => Unit) = x     // works also if we replace
test: ((Any) => Unit)(Any) => Unit        // Any by a different type

scala> test(println)
res0: (Any) => Unit = <function>
```

Rationale

Mixins are the "join" of an inheritance relation

Outline

Multiple dispatch languages

Originally used in functional languages

- The ancestor: CLOS (Common Lisp Object System)
- Cecil
- Dylan
- Now getting into mainstream languages by extensions (Ruby's `Multiple Dispatch` library, C# 4.0 dynamic or multi-method library, ...) or directly as in Perl 6.

Multiple dispatch in Perl 6

```
multi sub identify(Int $x) {
    return "$x is an integer."; }

multi sub identify(Str $x) {
    return qq<"$x" is a string.>; }   #qq stands for ‘‘double quote’’

multi sub identify(Int $x, Str $y) {
    return "You have an integer $x, and a string \"$y\"."; }

multi sub identify(Str $x, Int $y) {
    return "You have a string \"$x\", and an integer $y."; }

multi sub identify(Int $x, Int $y) {
    return "You have two integers $x and $y."; }

multi sub identify(Str $x, Str $y) {
    return "You have two strings \"$x\" and \"$y\"."; }

say identify(42);
say identify("This rules!");
say identify(42, "This rules!");
say identify("This rules!", 42);
say identify("This rules!", "I agree!");
say identify(42, 24);
```

Multiple dispatch in Perl 6

Embedded in classes

```
class Test {
    multi method identify(Int $x) {
      return "$x is an integer."; }
    }
    multi method identify(Str $x) {
      return qq<"$x" is a string.>;
    }
}
my Test $t .= new();
$t.identify(42);                # 42 is an integer
$t.identify("weasel");          # "weasel" is a string
```

Multiple dispatch in Perl 6

Embedded in classes

```
class Test {
    multi method identify(Int $x) {
      return "$x is an integer."; }
    }
    multi method identify(Str $x) {
      return qq<"$x" is a string.>;
    }
}
my Test $t .= new();
$t.identify(42);              # 42 is an integer
$t.identify("weasel");        # "weasel" is a string
```

Partial dispatch

```
multi sub write_to_file(str $filename , Int $mode ;; Str $text) {
  ...
}

multi sub write_to_file(str $filename ;; Str $text) {
  ...
}
```

Class methods as special case of partial dispatch

```
class Point {
    has $.x is rw;
    has $.y is rw;

    method set_coordinates($x, $y) {
        $.x = $x;
        $.y = $y;
    }
};

class Point3D is Point {
    has $.z is rw;

    method set_coordinates($x, $y) {
        $.x = $x;
        $.y = $y;
        $.z = 0;
    }
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
say $a.x;                                          # 23
say $a.z;                                          # 12
$a.set_coordinates(10, 20);
say $a.z;                                          #  0
```

Equivalently with multi subroutines

```
class Point {
    has $.x is rw;
    has $.y is rw;
};

class Point3D is Point {
    has $.z is rw;
};

multi sub set_coordinates(Point $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
};

multi sub set_coordinates(Point3D $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
    $p.z = 0;
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
say $a.x;                                        # 23
say $a.z;                                        # 12
set_coordinates($a, 10, 20);
say $a.z;                                        #  0
```

Nota Bene

There is no encapsulation here.

```
class Point {
    has $.x is rw;
    has $.y is rw;
};

class Point3D is Point {
    has $.z is rw;
};

multi sub set_coordinates(Point $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
};

multi sub set_coordinates(Point3D $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
    $p.z = 0;
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
say $a.x;                                       # 23
say $a.z;                                       # 12
set_coordinates($a, 10, 20);
say $a.z;                                       #  0
```

Note this for the future (of the course)

```
class Point {
    has $.x is rw;
    has $.y is rw;
};

class Point3D is Point {
    has $.z is rw;
};

multi sub fancy(Point $p, Point3D $q) {
    say "first was called";
};

multi sub fancy(Point3D $p, Point $q) {
    say "second was called";
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
fancy($a,$a)};
```

Note this for the future (of the course)

```
class Point {
    has $.x is rw;
    has $.y is rw;
};

class Point3D is Point {
    has $.z is rw;
};

multi sub fancy(Point $p, Point3D $q) {
    say "first was called";
};

multi sub fancy(Point3D $p, Point $q) {
    say "second was called";
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
fancy($a,$a)};
```

```
Ambiguous dispatch to multi 'fancy'. Ambiguous candidates had signatures:
:(Point $p, Point3D $q)
:(Point3D $p, Point $q)
in Main (file <unknown>, line <unknown>)
```

Outline

Some compromises are needed

- No polymorphic objects
- Need of explicit coercions
- *No overloading*

Some compromises are needed

- No polymorphic objects
- Need of explicit coercions
- *No overloading*

A brief parenthesis

A scratch course on OCaml classes and objects by Didier Remy (just click here) `http://gallium.inria.fr/~remy/poly/mot/2/index.html`

OCaml Classes

Some compromises are needed

- No polymorphic objects
- Need of explicit coercions
- *No overloading*

A brief parenthesis

A scratch course on OCaml classes and objects by Didier Remy (just click here) `http://gallium.inria.fr/~remy/poly/mot/2/index.html`

Programming is in general less liberal than in “pure” object-oriented languages, because of the constraints due to type inference.

OCaml Classes

Some compromises are needed

- No polymorphic objects
- Need of explicit coercions
- ***No overloading***

A brief parenthesis

A scratch course on OCaml classes and objects by Didier Remy (just click here) `http://gallium.inria.fr/~remy/poly/mot/2/index.html`

Programming is in general less liberal than in “pure” object-oriented languages, because of the constraints due to type inference.

OCaml Classes

Some compromises are needed

- No polymorphic objects
- Need of explicit coercions
- ***No overloading*** **... Haskell makes exactly the opposite choice ...**

A brief parenthesis

A scratch course on OCaml classes and objects by Didier Remy (just click here) `http://gallium.inria.fr/~remy/poly/mot/2/index.html`

Programming is in general less liberal than in “pure” object-oriented languages, because of the constraints due to type inference.

Outline

Haskell's Typeclasses

Typeclasses define a set of functions that can have different implementations depending on the type of data they are given.

```
class BasicEq a where
    isEqual :: a -> a -> Bool
```

An instance type of this typeclass is any type that implements the functions defined in the typeclass.

Haskell's Typeclasses

Typeclasses define a set of functions that can have different implementations depending on the type of data they are given.

```
class BasicEq a where
    isEqual :: a -> a -> Bool
```

An instance type of this typeclass is any type that implements the functions defined in the typeclass.

```
ghci> :type isEqual
isEqual :: (BasicEq a) => a -> a -> Bool
```

« For all types `a`, so long as `a` is an instance of `BasicEq`, `isEqual` takes two parameters of type `a` and returns a `Bool` »

To define an instance:

```
instance BasicEq Bool where
    isEqual True  True  = True
    isEqual False False = True
    isEqual _     _     = False
```

To define an instance:

```
instance BasicEq Bool where
    isEqual True  True  = True
    isEqual False False = True
    isEqual _     _     = False
```

We can now use `isEqual` on `Bool`s, but not on any other type:

```
ghci> isEqual False False
True
ghci> isEqual False True
False
ghci> isEqual "Hi" "Hi"

<interactive>:1:0:
    No instance for (BasicEq [Char])
      arising from a use of ‘isEqual’ at <interactive>:1:0-16
    Possible fix: add an instance declaration for (BasicEq [Char])
    In the expression: isEqual "Hi" "Hi"
    In the definition of ‘it’: it = isEqual "Hi" "Hi"
```

As suggested we should add an instance for strings

```
instance BasicEq String where ....
```

A not-equal-to function might be useful. Here's what we might say to define a typeclass with two functions:

```
class BasicEq2 a where
    isEqual2 :: a -> a -> Bool
    isEqual2 x y = not (isNotEqual2 x y)

    isNotEqual2 :: a -> a -> Bool
    isNotEqual2 x y = not (isEqual2 x y)
```

People implementing this class must provide an implementation of at least one function. They can implement both if they wish, but they will not be required to.

Type-classes vs OOP

Type classes are like traits/interfaces/abstract classes, not classes itself (no *proper* inheritance and data fields).

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- let's just implement one function in terms of the other
  x /= y = not (x == y)
```

is, in a Java-like language:

```
interface Eq<A> {
  boolean equal(A x);
  boolean notEqual(A x) {        // default, can be overridden
     return !equal(x);
  }
}
```

Type-classes vs OOP

Type classes are like traits/interfaces/abstract classes, not classes itself (no *proper* inheritance and data fields).

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- let's just implement one function in terms of the other
  x /= y = not (x == y)
```

is, in a Java-like language:

```
interface Eq<A> {
  boolean equal(A x);
  boolean notEqual(A x) {        // default, can be overridden
     return !equal(x);
  }
}
```

Haskell typeclasses concern more overloading than inheritance. They are closer to multi-methods (overloading and no access control such as `private` fields), but only with *static dispatching*.

A flavor of inheritance

They provide a very limited form of inheritance (but without overriding and late binding!):

```
class  Eq a => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

Type-classes vs OOP

A flavor of inheritance

They provide a very limited form of inheritance (but without overriding and late binding!):

```
class  Eq a => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

The *subclass* `Ord` "inherits" the operations from its *superclass* `Eq`. In particular, "methods" for subclass operations can assume the existence of "methods" for superclass operations:

```
class  Eq a => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
  x < y =  x <= y && x /= y
```

Inheritance thus is not on instances but rather on types (a Haskell class is not a type but a template for a type).

Type-classes vs OOP

A flavor of inheritance

They provide a very limited form of inheritance (but without overriding and late binding!):

```
class  Eq a => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

The *subclass* `Ord` "inherits" the operations from its *superclass* `Eq`. In particular, "methods" for subclass operations can assume the existence of "methods" for superclass operations:

```
class  Eq a => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
  x < y =  x <= y && x /= y
```

Inheritance thus is not on instances but rather on types (a Haskell class is not a type but a template for a type). *Multiple inheritance* is possible:

```
class (Real a, Fractional a) => RealFrac a where ...
```

Hybrid solutions

- Mixins raised in FP area (Common Lisp) and are used in OOP to allow minimal module composition (as functors do very well). On the other hand they could endow ML module system with inheritance and overriding
- Multi-methods are an operation centric version of OOP. They look much as a functional approach to objects
- OCaml and Haskell classes are an example of how functional language try to obtain the same kind of modularity as in OOP.

Something missing in OOP

What about Functors?

Outline

Generics in C#

Why in C# and not in Java?

Direct support in the CLR and IL (intermediate language)

The CLR implementation pushes support for generics into almost all feature areas, including serialization, remoting, reflection, reflection emit, profiling, debugging, and pre-compilation.

Generics in C#

Why in C# and not in Java?

Direct support in the CLR and IL (intermediate language)

The CLR implementation pushes support for generics into almost all feature areas, including serialization, remoting, reflection, reflection emit, profiling, debugging, and pre-compilation.

Java Generics based on GJ

Rather than extend the JVM with support for generics, the feature is "compiled away" by the Java compiler

Consequences:

- generic types can be instantiated only with reference types (e.g. string or object) and not with primitive types
- type information is not preserved at runtime, so objects with distinct source types such as List<string> and List<object> cannot be distinguished by run-time
- Clearer syntax

Generics Problem Statement

```
public class Stack
{
   object[] m_Items;
   public void Push(object item)
   {...}
   public object Pop()
   {...}
}
```

- runtime cost (boxing/unboxing, garbage collection)
- type safety

```
Stack stack = new Stack();
stack.Push(1);
stack.Push(2);
int number = (int)stack.Pop();

Stack stack = new Stack();
stack.Push(1);
string number = (string)stack.Pop();        // exception thrown
```

Heterogenous translation

You can overcome these two problems by writing type-specific stacks. For integers:

```
public class IntStack
{
   int[] m_Items;
   public void Push(int item){...}
   public int Pop(){...}
}
IntStack stack = new IntStack();
stack.Push(1);
int number = stack.Pop();
```

For strings:

```
public class StringStack
{
   string[] m_Items;
   public void Push(string item){...}
   public string Pop(){...}
}
StringStack stack = new StringStack();
stack.Push("1");
string number = stack.Pop();
```

Problem

Writing type-specific data structures is a tedious, repetitive, and error-prone task.

Problem

Writing type-specific data structures is a tedious, repetitive, and error-prone task.

Solution

Generics

```
public class Stack<T>
{
   T[] m_Items;
   public void Push(T item)
   {...}
   public T Pop()
   {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

Problem

Writing type-specific data structures is a tedious, repetitive, and error-prone task.

Solution

Generics

```
public class Stack<T>
{
   T[] m_Items;
   public void Push(T item)
   {...}
   public T Pop()
   {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

You have to instruct the compiler which type to use instead of the generic type parameter T, both when declaring the variable and when instantiating it:

`Stack<int> stack = new Stack<int>();`

```
public class Stack<T>{
   readonly int m_Size;
   int m_StackPointer = 0;
   T[] m_Items;
   public Stack():this(100){
   }
   public Stack(int size){
      m_Size = size;
      m_Items = new T[m_Size];
   }
   public void Push(T item){
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
   public T Pop(){
      m_StackPointer--;
      if(m_StackPointer >= 0) {
         return m_Items[m_StackPointer]; }
      else {
         m_StackPointer = 0;
         throw new InvalidOperationException("Cannot pop an empty stack");
      }
   }
}
```

Recap

Two different styles to implement generics (when not provided by the VM):

1. *Homogenous:* replace occurrences of the type parameter by the type `Object`. This is done in GJ and, thus, in Java (>1.5).
2. *Heterogeneous:* make one copy of the class for each instantiation of the type parameter. This is done by C++ and Ada.

The right solution is to support generics directly in the VM

Recap

Two different styles to implement generics (when not provided by the VM):

1. *Homogenous:* replace occurrences of the type parameter by the type `Object`. This is done in GJ and, thus, in Java (>1.5).
2. *Heterogeneous:* make one copy of the class for each instantiation of the type parameter. This is done by C++ and Ada.

The right solution is to support generics directly in the VM

Unfortunately, Javasoft marketing people did not let Javasoft researchers to change the JVM.

Multiple Generic Type Parameters

```
class Node<K,T> {
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node() {
      Key      = default(K);            // the "default" value of type K
      Item     = default(T);            // the "default" value of type T
      NextNode = null;
   }
   public Node(K key,T item,Node<K,T> nextNode) {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}

public class LinkedList<K,T> {
   Node<K,T> m_Head;
   public LinkedList() {
      m_Head = new Node<K,T>();
   }
   public void AddHead(K key,T item){
      Node<K,T> newNode = new Node<K,T>(key,item,m_Head);
      m_Head = newNode;
   }
}
```

Generic Type Constraints

Suppose you would like to add searching by key to the linked list class

```
public class LinkedList<K,T> {

   public T Find(K key) {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key == key)                    //Will not compile
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   // rest of the implementation
}
```

The compiler will refuse to compile this line

`if(current.Key == key)`

because the compiler does not know whether K (or the actual type supplied by the client) supports the == operator.

We must ensure that K implements the following interface

```
public interface IComparable  {
   int CompareTo(Object other);
   bool Equals(Object other);
}
```

We must ensure that K implements the following interface

```
public interface IComparable  {
   int CompareTo(Object other);
   bool Equals(Object other);
}
```

This can be done by specifying a constraint:

```
public class LinkedList<K,T> where K : IComparable {
   public T Find(K key) {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key.CompareTo(key) == 0)
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   //Rest of the implementation
}
```

We must ensure that K implements the following interface

```
public interface IComparable  {
   int CompareTo(Object other);
   bool Equals(Object other);
}
```

This can be done by specifying a constraint:

```
public class LinkedList<K,T> where K : IComparable {
   public T Find(K key) {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key.CompareTo(key) == 0)
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   //Rest of the implementation
}
```

Problems

1. `key` is boxed/unboxed when it is a value (i.e. not an object)
2. The static information that `key` is of type `K` is not used (`CompareTo` requires a parameter just of type `Object`).

F-bounded polymorphism

In order to enhance type-safety (in particular, enforce the argument of `K.CompareTo` to have type `K` rather than `Object`) and avoid boxing/unboxing when the key is a value, we can use a generic version of IComparable.

```
public interface IComparable<T> {
   int CompareTo(T other);
   bool Equals(T other);
}
```

F-bounded polymorphism

In order to enhance type-safety (in particular, enforce the argument of `K.CompareTo` to have type `K` rather than `Object`) and avoid boxing/unboxing when the key is a value, we can use a generic version of IComparable.

```
public interface IComparable<T> {
   int CompareTo(T other);
   bool Equals(T other);
}
```

This can be done by specifying a constraint:

```
public class LinkedList<K,T> where K : IComparable<K> {
   public T Find(K key) {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key.CompareTo(key) == 0)
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   //Rest of the implementation
}
```

Generic methods

You can define method-specific (possibly constrained) generic type parameters even if the containing class does not use generics at all:

```
public class MyClass
{
   public void MyMethod<T>(T t) where T : IComparable<T>

   {...}
}
```

Generic methods

You can define method-specific (possibly constrained) generic type parameters even if the containing class does not use generics at all:

```
public class MyClass
{
   public void MyMethod<T>(T t) where T : IComparable<T>

   {...}
}
```

When calling a method that defines generic type parameters, you can provide the type to use at the call site:

```
MyClass obj = new MyClass();
obj.MyMethod<int>(3)
```

Subtyping

Generics are *invariant*:

```
List<string> ls = new List<string>();
ls.Add("test");
List<object> lo = ls;   // Can’t do this in C#
object o1 = lo[0];      // ok - converting string to object
lo[0] = new object();   // ERROR - can’t convert object to string
```

Subtyping

Generics are *invariant*:

```
List<string> ls = new List<string>();
ls.Add("test");
List<object> lo = ls;    // Can't do this in C#
object o1 = lo[0];       // ok - converting string to object
lo[0] = new object();    // ERROR - can't convert object to string
```

This is the right decision as the example above shows.

Subtyping

Generics are *invariant*:

```
List<string> ls = new List<string>();
ls.Add("test");
List<object> lo = ls;   // Can't do this in C#
object o1 = lo[0];      // ok - converting string to object
lo[0] = new object();   // ERROR - can't convert object to string
```

This is the right decision as the example above shows.
Thus

`S` is a subtype of `T` *does not imply* `Class<S>` is a subtype of `Class<T>`.

If this (covariance) were allowed, the last line would have to result in an exception (eg. InvalidCastException).

Beware of self-proclaimed type-safety

Since `S` is a subtype of `T` *implies* `S[]` is subtype of `T[]`. (*covariance*)

Do not we have the same problem with arrays?

Beware of self-proclaimed type-safety

Since `S` is a subtype of `T` *implies* `S[]` is subtype of `T[]`. (*covariance*)

Do not we have the same problem with arrays? **Yes**

Beware of self-proclaimed type-safety

Since `S` is a subtype of `T` *implies* `S[]` is subtype of `T[]`. (*covariance*)

Do not we have the same problem with arrays? **Yes**

From Jim Miller CLI book

> *The decision to support covariant arrays was primarily to allow Java to run on the VES (Virtual Execution System). The covariant design is not thought to be the best design in general, but it was chosen in the interest of broad reach.*

(yes, it is not a typo, Microsoft decided to break type safety and did so in order to run Java in `.net`)

Beware of self-proclaimed type-safety

Since S is a subtype of T *implies* S[] is subtype of T[]. (*covariance*)

Do not we have the same problem with arrays? **Yes**

From Jim Miller CLI book

> *The decision to support covariant arrays was primarily to allow Java to run on the VES (Virtual Execution System). The covariant design is not thought to be the best design in general, but it was chosen in the interest of broad reach.*

(yes, it is not a typo, Microsoft decided to break type safety and did so in order to run Java in .net*)*

Regretful (and regretted) decision:

```
class Test {
   static void Fill(object[] array, int index, int count, object val) {
      for (int i = index; i < index + count; i++) array[i] = val;
   }
   static void Main() {
      string[] strings = new string[100];
      Fill(strings, 0, 100, "Undefined");
      Fill(strings, 0, 10, null);
      Fill(strings, 90, 10, 0);    //→System.ArrayTypeMismatchException
   }
}
```

Variant annotations

Add variants (C# 4.0)

```
// Covariant parameters can be used as result types
interface IEnumerator<out T> {
      T Current { get; }
      bool MoveNext();
}
// Covariant parameters can be used in covariant result types
interface IEnumerable<out T> {
      IEnumerator<T> GetEnumerator();
}
// Contravariant parameters can be used as argument types
interface IComparer<in T> {
      bool Compare(T x, T y);
}
```

Variant annotations

Add variants (C# 4.0)

```
// Covariant parameters can be used as result types
interface IEnumerator<out T> {
      T Current { get; }
      bool MoveNext();
}
// Covariant parameters can be used in covariant result types
interface IEnumerable<out T> {
      IEnumerator<T> GetEnumerator();
}
// Contravariant parameters can be used as argument types
interface IComparer<in T> {
      bool Compare(T x, T y);
}
```

This means we can write code like the following:

```
IEnumerable<string> stringCollection = ...;             //smaller type
IEnumerable<object> objectCollection = stringCollection; //larger type
foreach( object o in objectCollection ) { ... }

IComparer<object> objectComparer = ...;                 //smaller type
IComparer<string> stringComparer = objectComparer;      //larger type
bool b = stringComparer.Compare( "x", "y" );
```

Features becoming standard in modern OOLs ...

In Scala we have generics classes and methods with annotations and bounds

```
class ListNode[+T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend[U >: T](elem: U): ListNode[U] =
    ListNode(elem, this)
}
```

Features becoming standard in modern OOLs . . .

In Scala we have generics classes and methods with annotations and bounds

```
class ListNode[+T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend[U >: T](elem: U): ListNode[U] =
    ListNode(elem, this)
}
```

and F-bounded polymorphism as well:

```
class GenCell[T](init: T) {
  private var value: T = init
  def get: T = value
  def set(x: T): unit = { value = x }
}

trait Ordered[T] {
  def < (x: T): boolean
}

def updateMax[T <: Ordered[T]](c: GenCell[T], x: T) =
  if (c.get < x) c.set(x)
```

. . . but also in FP.

All these characteristics are present in different flavours in OCaml

. . . but also in FP.

All these characteristics are present in different flavours in OCaml

Generics are close to parametrized classes:

```
# exception Empty;;

class ['a] stack =
  object
   val mutable p : 'a list =  []
   method push x = p <- x :: p
   method pop =
     match p with
     | [] -> raise Empty
     | x::t -> p <- t; x
  end;;
class ['a] stack :
object val mutable p : 'a list method pop : 'a method push : 'a -> unit en
# new stack # push 3;;
- : unit = ()
# let x = new stack;;
val x : '_a stack = <obj>
# x # push 3;;
- : unit = ()
# x;;
- : int stack = <obj>
```

Constraints can be deduced by the type-checker

```
#class ['a] circle (c : 'a) =
   object
     val mutable center = c
     method center = center
     method set_center c = center <- c
     method move = (center#move : int -> unit)
   end;;
class ['a] circle :
  'a ->
  object
    constraint 'a = < move : int -> unit; .. >
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

Constraints can be imposed by the programmer

```
#class point x_init =
   object
     val mutable x = x_init
     method get_x = x
     method move d = x <- x + d
   end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit e

#class ['a] circle (c : 'a) =
   object
     constraint 'a = #point    (* = < get_x : int; move : int->unit; .. > *)
     val mutable center = c
     method center = center
     method set_center c = center <- c
     method move = center#move
   end;;
class ['a] circle :
  'a ->
  object
    constraint 'a = #point
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

Explicit instantiation is done just for inheritance

```
#class colored_point x (c : string) =
   object
     inherit point x
     val c = c
     method color = c
   end;;
class colored_point :
  int ->
  string ->
  object
   :
  end

#class colored_circle c =
   object
     inherit [colored_point] circle c
     method color = center#color
   end;;
class colored_circle :
  colored_point ->
  object
    val mutable center : colored_point
    method center : colored_point
    method color : string
    method move : int -> unit
    method set_center : colored_point -> unit
  end
```

Variance constraints

- Variance constraint are meaningful only with subtyping (i.e. objects, polymorphic variants, . . .).

Variance constraints

- Variance constraint are meaningful only with subtyping (i.e. objects, polymorphic variants, ...).
- They can be used in OCaml (not well documented): useful on abstract types to describe the expected behaviour of the type with respect to subtyping.

Variance constraints

- Variance constraint are meaningful only with subtyping (i.e. objects, polymorphic variants, . . .).
- They can be used in OCaml (not well documented): useful on abstract types to describe the expected behaviour of the type with respect to subtyping.
- For instance, an immutable container type (like lists) will have a covariant type:

  ```
  type (+'a) container
  ```

  meaning that if s is a subtype of t then s container is a subtype of t container. On the other hand an acceptor will have a contravariant type:

  ```
  type (-'a) acceptor
  ```

  meaning that if s is a subtype of t then t acceptor is a subtype s acceptor.

Variance constraints

- Variance constraint are meaningful only with subtyping (i.e. objects, polymorphic variants, . . .).
- They can be used in OCaml (not well documented): useful on abstract types to describe the expected behaviour of the type with respect to subtyping.
- For instance, an immutable container type (like lists) will have a covariant type:

  ```
  type (+'a) container
  ```

  meaning that if s is a subtype of t then s container is a subtype of t container. On the other hand an acceptor will have a contravariant type:

  ```
  type (-'a) acceptor
  ```

  meaning that if s is a subtype of t then t acceptor is a subtype s acceptor.

Variance constraints

- Variance constraint are meaningful only with subtyping (i.e. objects, polymorphic variants, ...).
- They can be used in OCaml (not well documented): useful on abstract types to describe the expected behaviour of the type with respect to subtyping.
- For instance, an immutable container type (like lists) will have a covariant type:

```
type (+'a) container
```

  meaning that if s is a subtype of t then s container is a subtype of t container. On the other hand an acceptor will have a contravariant type:

```
type (-'a) acceptor
```

  meaning that if s is a subtype of t then t acceptor is a subtype s acceptor.

see also `https://ocaml.janestreet.com/?q=node/99`

Summary for generics . . .

Generics endow OOP with features from the FP universe

Generics endow OOP with features from the FP universe

Generics on classes (in particular combined with Bounded Polymorphism) look close to functors.

Generics endow OOP with features from the FP universe

Generics on classes (in particular combined with Bounded Polymorphism) look close to functors.

Compare the Scala program in two slides with the `Set` functor with signature:

```
module Set :
functor (Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

where

```
type comparison = Less | Equal | Greater;;

module type ORDERED_TYPE =
    sig
     type t
     val compare: t -> t -> comparison
    end;;
```

and that is defined as:

```
module Set (Elt: ORDERED_TYPE) =
  struct
    type element = Elt.t
    type set = element list
    let empty = []
    let rec add x s =
      match s with
        [] -> [x]
      | hd::tl ->
         match Elt.compare x hd with
           Equal   -> s          (* x is already in s *)
         | Less    -> x :: s     (* x is smaller than all elmts of s *)
         | Greater -> hd :: add x tl
    let rec member x s =
      match s with
        [] -> false
      | hd::tl ->
          match Elt.compare x hd with
            Equal   -> true      (* x belongs to s *)
          | Less    -> false     (* x is smaller than all elmts of s *)
          | Greater -> member x tl
  end;;
```

```
trait Ordered[A] {
  def compare(that: A): Int
  def < (that: A): Boolean =  (this compare that) <  0
  def > (that: A): Boolean =  (this compare that) >  0
}

trait Set[A <: Ordered[A]] {
  def add(x: A): Set[A]
  def member(x: A): Boolean
}

class EmptySet[A <: Ordered[A]] extends Set[A] {
  def member(x: A): Boolean = false
  def add(x: A): Set[A] =
           new NonEmptySet(x, new EmptySet[A], new EmptySet[A])
}

class NonEmptySet[A <: Ordered[A]]
      (elem: A, left: Set[A], right: Set[A]) extends Set[A] {
  def member(x: A): Boolean =
     if (x < elem) left member x
     else if (x > elem) right member x
     else true
  def add(x: A): Set[A] =
     if (x < elem) new NonEmptySet(elem, left add x, right)
     else if (x > elem) new NonEmptySet(elem, left, right add x)
     else this
}
```

Generics endow OOP with features from the FP universe

Generics on methods bring the advantages of parametric polymorphism

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}

val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

Generics endow OOP with features from the FP universe

Generics on methods bring the advantages of parametric polymorphism

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}

val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

Local Type Inference

It is possible to deduce the type parameter from `s1` and `s2`. Scala does it for us.

```
val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix(s1, s2))
```