CS558 Programming Languages
Fall 2016
Lecture 10a
Object-oriented programs are structured in terms of objects: collections of variables ("fields") and functions ("methods").

OOP is particularly appropriate for programs that model discrete real-world processes for simulation or interaction. Idea: one program object corresponds to each real-world object. But OOP can be used for any programming task.

Typical key characteristics of OOP:

- Dynamic Dispatch
- Encapsulation
- Inheritance
- Subtyping

Important OO Languages: Simula 67, Smalltalk, C++, Java, C#, JavaScript, Python, Ruby, Scala, ...

Differences among languages: Are there static types? Are there classes? Are all values objects?
The fundamental control structure in OOP is **method invocation**, similar to function call in ordinary procedural programming, but:

- In most OO languages, there is a superficial syntactic difference: each function defined for an object takes the object itself as an implicit argument.

  ```
  s.add(x) ; OO style
  Set.add(s,x) ; procedural style
  ```

- Corresponding change in **metaphor**: instead of applying functions to values, we talk of “sending messages to objects.”
A more important difference is that in OOP, the receiving object itself controls how each message is processed. E.g., the effect of \texttt{s.add} can change depending on exactly which object is bound to \texttt{s}. This is a form of \textbf{dynamic overloading} (a certain kind of \textbf{polymorphism}).

Example:

\begin{verbatim}
    s1 = new ordered-list-set
    s2 = new balanced-tree-set
    if ... then s = s1 else s = s2
    s.add(42)
\end{verbatim}

The implementation of the \texttt{add} method is completely different in \texttt{s1} and \texttt{s2}; choice of which runs is determined at runtime.
In OOP, we typically want to create multiple objects having the same structure (field and method names).

In most OO languages this is done by defining a class, which is a kind of template from which new objects can be created.

- Different instances of the class will typically have different field values, but all will share the same method implementations.

- Classes are not essential; there are some successful OO languages (e.g. JavaScript) in which new objects are created by cloning existing prototype objects.
Classes vs. ADT’s

Class definitions are much like Abstract Data Type (ADT) definitions.

• In particular, objects often (though not always) designed so that their data fields can only be accessed by the object’s own methods. This kind of encapsulation is just what ADT’s offer.

• Using encapsulation makes it possible to change the representation or implementation of an object without affecting client code that interacts with the object only via method calls. This helps support modular development of large programs.

• However, OO programmers often violate encapsulation policies. (For example, object fields may be public, allowing them to be accessed from code outside of methods.)
SUBTYPING AND SUBCLASSES

If we think of types as representing sets of values, subtypes represent subsets of values.

- Informally, we can often determine that $B$ is a subtype of $A$ if it makes sense to say that every value of $B$ “is a” value of $A$.

Values in the subtype support all the operations of the supertype, and maybe more as well.

- Thus, we might say that $B$ is a subtype of $A$ if we can use a $B$ value wherever an $A$ value is expected.

Subtyping should obviously be a reflexive and transitive relationship; this generalizes to a sub-typing hierarchy among types. (But note that it is usually not symmetric!)

In many OO languages (including Smalltalk, C++, and Java) we declare subtypes by defining subclasses. (Java, Scala, and some other languages also offer other mechanisms; more later).
abstract class Image {
    def draw()
    def translate(dx:Int,dy:Int)
}

class Line(var x0:Int,var y0:Int,var delx:Int,var dely:Int) extends Image {
    def draw() = { Sc.moveto(x0,y0); Sc.drawto(x0+delx,y0+dely) }
    def translate(dx:Int,dy:Int) = { x0 += dx; y0 += dy }
}

class Text(var x0:Int,var y0:Int,var s:String) extends Image {
    def draw() = { Sc.moveto(x0,y0); Sc.write(s) }
    def translate(dx:Int,dy:Int) = { x0 += dx; y0 += dy }
}

class Main {
    val imgs : List[Image] = List(new Line(0,0,10,10), new Text(5,5,"hello"))
    for (img <- imgs) {
        img.translate(3,4); img.draw(); img.translate(-3,-4)
    }
}
Classes might also be related because their implementations are similar. To avoid having to write the code twice, we might like to inherit most of the implementation of one class from the other, possibly making just a few alterations.

In Smalltalk, C++, Java, this is again expressed by making the class that inherits the implementation a subclass of the class providing the implementation.

- This works nicely when the inheriting class is also a subtype of the providing class.
- But note: Sometimes we’d like B to inherit implementation from A even when the conceptual object represented by B is not a specialization of that represented by A; i.e. B is not really a subtype of A. (More later.)
Here we place the `translate` code in the superclass, where it can be shared.

```scala
abstract class Image(var x0:Int,var y0:Int) {
    def draw()
    def translate(dx:Int,dy:Int) = { x0 += dx; y0 += dy }
}

class Line(_x0:Int,_y0:Int,var delx:Int,var dely:Int) extends Image(_x0,_y0) {
    def draw() = { Sc.moveto(x0,y0); Sc.drawto(x0+delx,y0+dely) }
}

class Text(_x0:Int,_y0:Int,var s:String) extends Image(_x0,_y0) {
    def draw() = { Sc.moveto(x0,y0); Sc.write(s) }
}

In many cases, sharing code like this is only possible if we refactor the design.
Our ability to inherit implementation code from a super-class in enhanced by dynamic dispatch.

The key idea here is that calls are always dispatched to the original receiving object, so that superclass code can access functionality defined in the subclasses.

(In C++, this is only true for methods declared as virtual; in Java or Scala it is true for all methods by default.)

Example: Consider capturing the pattern used in Main by adding a drawShifted method to all images. We can define this at the level of the Image class, while still invoking the specific translate and draw methods from the relevant subclass.
abstract class Image(var x0:Int,var y0:Int) {
    def draw()
    def translate(dx:Int,dy:Int) = { x0 += dx; y0 += dy }
    def drawShifted(dx:Int,dy:Int) = {
        translate(dx,dy); draw(); translate(-dx,-dy)
    }
}

class Line(_x0:Int,_y0:Int,var delx:Int,var dely:Int) extends Image(_x0,_y0) {
    def draw() = { Sc.moveto(x0,y0); Sc.drawto(x0+delx,y0+dely) }
}

class Text(_x0:Int,_y0:Int,var s:String) extends Image(_x0,_y0) {
    def draw() = { Sc.moveto(x0,y0); Sc.write(s) }
}

class Main {
    val imgs : List[Image] = List(new Line(0,0,10,10), new Text(5,5,"hello"))
    for (img <- imgs)
        img.drawShifted(3,4)
}
As a program changes over time, it is easy to add new subclasses as long as they implement the methods expected of them. For example, we could add `Circle` or `Square` as subclasses of `Image` without breaking any existing code.

For added flexibility, we can also have a new subclass selectively override the implementations of some superclass functions. Again, the rule that all internal messages go to the original receiver is essential here, to make sure the most-specific version of code gets invoked.
Example: Add new `BitMap` object, with its own version of `translate` that (for some unknown and irrelevant reason) scales the argument by a factor $m$.

```
abstract class Image(var x0:Int,var y0:Int) {
  def draw()
  def translate(dx:Int,dy:Int) = { x0 += dx; y0 += dy }
}

... 

class BitMap(_x0:Int,_y0:Int,var m:Int,var b:Array[Boolean]) extends Image(_x0*m,_y0*m) {
  override def translate(dx:Int,dy:Int) = { x0 += dx*m; y0 += dy*m }
  def draw() = { Sc.moveto(x0,y0); Sc.blit(m,b) }
}
```
Often we’d like to use both subtyping and inheritance, but the subclassing structure we want for these purposes may be different.

For example, suppose we want to define a class `ImageGroup` whose objects are collections of images that can be translated or drawn as a unit. We want to be able to insert and retrieve the elements of a group just as for objects of the Scala library class `Set`, using add, remove, iterate, etc.

For subtyping purposes, our group class should clearly be a subclass of `Image`, but for code inheritance purposes, it would be very convenient to make it a subclass of `Set`.

There are at least two common solutions to this problem:

- One approach is to give `ImageGroup` a field containing a `Set`, and reimplement all the set-like operations by delegating them to that set. This causes code duplication, but it also lets us craft a more precise interface containing just the operations we want for `ImageGroups`.
• Another approach is to use **multiple inheritance**. Scala has only single inheritance from classes, but it also has a notion of **traits**; these are like abstract class descriptions and are just the thing for describing **subtypes**.

So in Scala, we could define a trait `Displayable` rather than the abstract class `Image`, and make `ImageGroup` a subclass of `Set` that “**mixes in**” `Displayable`.

In Java, we could do something similar by making `Displayable` an **interface** that `ImageGroup` “**implements**”.

**Scala Traits Example...**
trait Displayable {
  def draw()
  def translate(dx:Int,dy:Int)
}

class Line(var x0:Int,var y0:Int,var delx:Int,var dely:Int) extends Displayable {
  ...
}
class Text(var x0:Int,var y0:Int,var s:String) extends Displayable {
  ...
}
class ImageGroup extends scala.collection.mutable.HashSet[Displayable] with Displayable {
  def translate(dx:Int,dy:Int) = {
    for (i <- this) // works because ImageGroup is a HashSet
      i.translate(dx,dy)
  }
def draw() {
    for (i <- this) // ditto
      i.draw()
  }
}

class Main {
  val img : ImageGroup = new ImageGroup()
  img += new Line(0,0,10,10); // works because ImageGroup is a HashSet
  img += new Line(1,1,200,500); // ditto
  img.translate(3,4); img.draw(); img.translate(-3,-4)
}
**Representation of Objects**

In a naive interpreted implementation, each object is represented by a heap-allocated record, containing

- Name and values of each instance field.
- Pointer to class description record.

Each class is represented by a (essentially static) record with:

- Name and code pointer for each class method.
- Pointer to super-class’s record.
EXAMPLE

(based on the code from slides 10,14)
INTERPRETED IMPLEMENTATION OF OPERATIONS

To perform a message send (function call) at runtime, the interpreter does a method lookup, starting from the receiver object, as follows:

- Use class pointer to find class description record.
- Search for method in class record. If found, invoke it; otherwise, continue search in superclass record.
- If no method found, issue “Message Not Understood” or similar error. (Can’t happen if language is statically typed.)

Instance fields are accessed in the object record; this always points to the receiver object record.

We can obviously improve on this naive scheme by caching results of searches; this works well when the same methods are called repeatedly.
How about **compiling** OO languages?

Dynamic binding makes compilation difficult:

- method code doesn’t know the precise class to which the object it is manipulating belongs,
- nor the precise method that will execute when it sends a message.

Instance fields are not so hard.

- Code that refers to instance fields of a given class will actually operate on objects of that class or of a subclass.

- Since a subclass always **extends** the set of instance variables defined in its superclass, compiler can consistently assign each instance variable a fixed (static) offset in the object record; this offset will be the same in every object record for that class and any of its subclasses.

- Compiled methods can then reference variables by offset rather than by name, just like ordinary record field offsets.

(But multiple inheritance, interfaces, or traits require more work.)
Handling message sends is harder, because methods can be overridden by subclasses.

Simple approach: keep a per-class static method table (or vtable) and “compile” message sends into indirect jumps through fixed offsets in this table.

Example: Classes in our example code all have this vtable structure:

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>draw code ptr.</td>
</tr>
<tr>
<td>1</td>
<td>translate code ptr.</td>
</tr>
</tbody>
</table>
```

These tables can get large, and much of their contents will be duplicated between a class and its superclasses. Still, this approach is used by many compiled languages including C++, Java. (Again, multiple inheritance, interfaces, traits, etc. cause complications.)