Scribe: Mark Sandler, Stevan Malesevic Lecturer: Andrew Myers

In the last ten years object oriented programming (OOP) has become leading programming paradigm. Majority of programmers are using OOP as primary programming tool. However, number of computer scientific does not share happiness about OOP with programmers. In this lecture we will try to explore some of the important OOP features and try to use our machinery to formally describe those features.

Important features

First let's list some of the features. Some of them have been explored in previous lectures and we are already familiar with them. Important features:

- Modules/encapsulation
- Subtyping
- Inheritance
- Static typing
- Parametric polymorphism

Encapsulation is also known as information hiding. The reason is that encapsulation allows hiding of real implementation of the object. Instead an interface is provided and the client can use it to communicate with the object. Subtyping can also be used to accomplish information hiding; the implementation is a a subtype of the interface, and the coercion to the interface type hides the private and protected fields. These two ideas are introduced in Smalltalk. Inheritance, which provides mechanism of code reuse, along with encapsulation, subtyping and static typing are included in Java and C++. It should be noticed that implementation of these two thins differs in these two languages. Currently there is no programming language that has all of these five OOP ideas. We will review in detail, some of the Java features. Roughly, a Java program is a set of *packages*, where each *package* consists of *classes*. Class can be seen as on the following picture.

$$\begin{array}{ll} \text{class object} & \left\{ \begin{array}{l} static\ fields \\ static\ methods \end{array} \right. \\ \text{type object/template} & \left\{ \begin{array}{l} fields \\ methods \end{array} \right. \\ & constructors \end{array}$$

So as we can see, a Java class contains static and objects methods. We declare a class we simultaneously create two different types: a *class object* type and an *object type*. The first one corresponds to the class and we can have only one object of this type (and which is created once we have defined the class) and the second one is actually a template and we can have different objects having this type. Constructors are similar to static procedures which allow us to create new instances of the type objects (with probably different properties).

To be very precise, every non-static method can be one of three types *public*, *private* and *protected*. The first type corresponds to method visible from anywhere, second only inside other class methods, and the last one is visible from the descendants of the class (and also from package which contains the class). Therefore actually there are even four different types created by the class definition (class type, type corresponding to the external view of the object, type corresponding to the view of object within the package and type corresponding to the internal view of the object).

Inheritance

Let's look at the one example of inheritance.

```
class A {                          class B extends A {
    int f() {return g()}               int g() {return 1}
    int g() {return 0}             }
}
```

Here we defined to class A and B, where class B is a descendant of class A. Also in the class B we override method g from class A. Let's now look at the part of the code where we use these classes.

$$B\, b = \textbf{new}\ B();$$
$$b.f();$$

What we will get as a result of calling method f? Note, that we have not overriden it in B. However we will get 1. Reason for this is that although we use method f from definition of A it calls method g not from A, but from the definition of *actual* object type (i.e. selection is made dynamically), so in our case b is object of type B therefore it will call $B.g()$.

There is one more useful thing that we can do, to make code more intuitive. Namely, we can define some methods as *abstract* (i.e. without definition of what they should do), obviously these methods must be overriden in descendants with some suitable code. But however we can call this methods even in the base class. Base class which contains *abstract* methods is also called *abstract*.

How can we use inheritance? Here is one more example. Suppose we want to implement graphic library. We will start with definition of point:

$$\begin{array}{l}\textbf{interface}\ Point\ \{\\ \quad \textbf{int}\ x()\\ \quad \textbf{int}\ y()\\ \}\end{array} \quad \Leftrightarrow \quad Point = \{x : 1 \to \textbf{int},\ y : 1 \to \textbf{int}\}$$

Here we see that we can think of that interface as a special kind of class, where all methods are made abstract (so it is just a definition of the type).

Now we want to continue our definitions, and would like to define the interface *Shape*. It will be our base for further objects (like *Box Circle*, etc), so in this interface we want to define some common properties. For example, each shape has an area, and some rectangle covering it, so it is useful to add corresponding methods in definition of *Shape*, also we might want to know if given *Shape* intersect with some other *Shape*. Here is a definition:

$$\begin{array}{l}\textbf{interface}\ Shape\ \{\\ \quad PointLT();\\ \quad PointRB();\\ \quad \textbf{float}\ Area();\\ \quad \textbf{boolean}\ Intersects(Shape\ L);\\ \}\end{array}$$

Corresponds to the following type:

$$Shape = \mu S.\{LT : 1 \to Point,\ RB : 1 \to Point,\ Area : 1 \to \textbf{float},\ Intersects : S \to \textbf{boolean}\}$$

Now we want to got further and define an abstract class *CommonShape* which will contain implementation of *Intersect* method.

$$\begin{array}{l}\textbf{abstract class}\ CommonShape\ \textbf{implements}\ Shape\ \{\\ \quad PointLT();\\ \quad PointRB();\\ \quad \textbf{abstract float}\ Area();\\ \quad \textbf{abstract boolean}\ Intersects_specific(Shape\ L);\\ \quad \textbf{boolean}\ Intersects(Shape\ L)\{\\ \qquad \text{Check whether covering rectangles of this object}\\ \qquad \text{and } L \text{ intersects or not, if they are intersect}\\ \qquad \text{call } IntersectSpific(L) \text{ otherwise return } false\\ \quad \}\\ \}\end{array}$$

Now we can define the simplest shape Box:

$$
\begin{array}{l}
\textbf{class}\ Box\ \textbf{extends}\ CommonShape\ \{ \\
\quad Pointlt, rb; PointLT(); \\
\quad PointRB(); \\
\quad \textbf{float}\ Area(); \{\ldots\} \\
\quad \textbf{boolean}\ IntersectSpecific(Shape\ L); \{\textbf{return}\ true\} \\
\quad Box(Point\ ll, Point\ rr)\{lt = ll, ur = rr\}//Constructor! \\
\}
\end{array}
$$

Here we got a constructor, when program calling it implicitly pass as argument pointer to just created class called *this*. (Actually *this* is passed in every method, however if would define it as explicit parameter we can lose type safety (because function in subtype will have a covariant parameter). It is still possible to model this object using recursive types.

But let's look to the next example:

$$
\begin{array}{l}
\textbf{class}\ RoundedBox\ \textbf{extends}\ Box\ \{ \\
\quad \textbf{float}\ R; \\
\quad RoundedBox(\textbf{float}\ rd, Point\ ll, ur)\ \{Box(ll, ur); R = rd; \} \\
\}
\end{array}
$$

Here we have called a constructor of superclass Box. It turns out that it is difficult to model such behaviour using recursive records. Let's view the following example:

$$
\begin{array}{l}
\textbf{class}\ C\ \textbf{extends}\ C'\ \{ \\
\quad C(x : \tau_C)\{\ldots\} \\
\quad \textbf{protected}\ l_j = \lambda x : \tau_{1_j} e'_j : \tau'_{1_j} \\
\quad \textbf{public}\ m_j = \lambda x : \tau_j . e_j : \tau'_j \\
\}
\end{array}
$$

Here we have defined three types

$$
\begin{array}{l}
T_{class} = \{c : \tau_c \to T_{object}\} \\
T_{protected} = \mu C.\{l_j : \tau_{1_j} \to \tau'_{1_j}, m_j : \tau_j \to \tau'_j\} \\
T_{object} = \mu C.\{m_j : \tau_j \to \tau'_j\}
\end{array}
$$

We actually need two different views of even the class object here because the constructor C can be called both from outside the class and from the constructor of a subclass of the class C. The signature above corresponds to only the former usage; a "protected" view of the class object exposes the other way to invoke the constructor. We continue with this issue in the next lecture.