Objects, Subclassing, Subtyping, and Inheritance

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

In these notes we will examine four basic concepts which play an important role in object-oriented programming. In particular, we will review classes and the relationship to types, subclassing, inheritance and subtyping.

Classes and types Classes in Java play two distinct roles, they introduce a "type" and they define code consisting of constructors, fields and methods. Consider the following example:

```
class A { ...  }

class B extends A { .... }
```

We first introduce a "type" with the name A, and then we introduce a "type" with the name B together with a relation between the type A and B[1]

Since the name of a class also denotes its type, we say that Java adopts a **nominal** notion of types. All object types are of the form C for some class C. Moreover, two object types are equivalent iff they arise from the **same** class, i.e. the type C is equal to the type D iff the class C is the same as class D. It is important to note that **distinct** classes (= distinct names) with **identical** fields and methods determine **distinct** types. This nominal view is in contrast to a **structural** view of types where the actual name may not matter, but two types are considered equivalent iff they specify the same fields and methods with the same types.

Subclassing = Inheritance A class C is a subclass of a class D, written $C \leq D$ iff C is explicitly declared to extend D. The class C may add more methods and fields, and perhaps by overloading some methods or fields which have been previously defined in D. C may also override methods with new implementations, provided that the argument types in the subclass match the argument types in the superclass exactly. The subclass relation is purely a matter of how the code is written.

In fact, the subclass may override the superclass's methods with **arbitrary** code and there is no guarantee that the objects of the subclass behave at all like the objects of the superclass! While we use in practice subclassing when the behavior of subclass instances is consistent with the behavior of superclass instances there is no mechanism which enforces this discipline.

[1] Strictly speaking the type classifies instances of the class and there is a distinction between the **class C** which denotes the actual body of code and the **type** Inst(C) of objects of the class C. In particular `new C (...)` has type Inst(C). In Java, this distinction is ellided, i.e. the class C is **identified** with the type Inst(C). We will elide here as well.

It is worth stressing that subclassing and inheritance has nothing to do with typing. It is a mechanism which allows code reuse. C^{++} for example provides inheritance separate from subtyping. In Java we have that subclassing induces subtyping (see below).

Single inheritance A class in Java can extend only one class. This restriction is called single inheritance. As a consequence, there is always precisely one parent class (with `Object` as root).

Subtyping Subtyping is a relation of **inclusion** between types.

$$B \leq A \qquad B \text{ is a subtype of } A$$

Informally, we say that a type B is a subtype of A if we can supply a value of type B whenever a value of type A is required. We began our study of subtyping by analyzing the different structure of types. For example, we say that a product $\tau_1 * \tau_2$ is a subtype of another product $\sigma_1 * \sigma_2$ if the components also have proper subtype relations, i.e $\tau_1 \leq \sigma_1$ and $\tau_2 \leq \sigma_2$. The subtyping rules we introduced worked on the structure of the types. In Java, subtyping is purely based on names. By writing

```
class B extends A { ... }
```

the programmer declares that B is a subtype of A, i.e. $B \leq A$. An A entity can be viewed as a special case of a B entity. A typical example is the following:

```
class Polygon { ... }

class Rectangle extends Polygon {... }

class Square extends Rectangle {... }
```

This code induces the following subtyping relationship:

$$\texttt{Square} \leq \texttt{Rectangle} \leq \texttt{Polygon}$$

Inheritance (= subclassing) induces subtyping in Java. But it is important to understand that they are not the same! We can have instances of subtyping without inheritance. Java for example supports subtyping independently of inheritance via interfaces.

Interfaces An interface is a declaration of a collection of method names – without method bodies. Interfaces specify the public methods of an object, but they may not (unfortunately) specify the public fields. All methods are implicitly `public abstract` so only their types are specified, not their implementation. All fields are implicitly `public static final` and hence are just constants of the interface. For example:

```
interface Plottable{
    double y (double x)
}
interface ColorPlot extends Pottable {
     int RED = 1, GREEN = 2, BLUE = 4;
}
```

The relation between classes and interfaces must be **explicitly declared**. Hence Java supports a **declarative view** of implementations. A class C implements an interface I iff it is **explicitly declared** to do so.

```
class sine implements Plottable { ... }
```

An alternative would be to adopt a **descriptive** view of an implementation. A class C implements an interface I iff the objects of the class C satisfy the requirements of the interface I. (this is what happens in SML when we check that a structure implements a signature) A structure (= class) implements an interface (signature) iff the objects (values) of the class C satisfy the requirements of I. The descriptive view is more flexible, since we are not required to anticipate the possible interfaces of interest for a class C when C is defined. Instead you can ask post hoc whether or not C implements I. Consider again the implementation of a structure and signature in SML. We can in fact implement a structure without defining a signature, and later add a signature (= interface), to describe which items we want to export.

Single inheritance, multiple subtyping A class can implement more than one interface, but it can only extend one class (directly). Therefore Java supports multiple subtyping via interfaces, but allows only single inheritance.

Here is an example of interfaces and multiple classes which implement the specified interface. Suppose we have written code to draw the graph of a mathematical function. Our intention is that we can plot **any** mathematical function (sine, cosine, exp, log etc.) The actual function we want to plot should be a parameter.

```
interface Plottable{
    double y (double x)
}

class sine implements Plottable { ... }

class exp implements Plottable { ... }

class log implements Plottable { ... }
```

All the classes implement Plottable, and therefore must provide an implementation for `y`.

Dynamic lookup and dispatch So far, we have encountered static binding in functional languages such as SML. The idea is that once the function is defined, the function definition is fixed and cannot change. For example, if we want to add two integers we define add as follows:

```
val add = fn (x,y) => x + y
```

We statically bind the variable `add` to the function definition `fn (x,y) => x + y`. If we later use the variable `add` we will exactly use this definition. The name of the function is primary, while the arguments are subsidiary.

In OO programming, the object is primary and the code is packaged with its methods; If we write `a.inc(5)` in Java, then the meaning depends dynamically on the type of the object `a` during runtime. It does not only depend on the *declared type* of the object, but each object has a method lookup table which is consulted during runtime to decide which method must be used. This is necessary, because we have inheritance and we can in fact have two different methods with the same name but which operate on different types. For example, assume we have defined a class `myInt` and a class `gaussInt` which extends `myInt`.

```
class myInt {
  ....
    public myInt add(myInt N){...}
    ...
}
class gaussInt extends myInt {
    ...
    public gaussInt add(gaussInt z){
      .....
    }
}
```

Both classes provide a method `add`. Which method is actually picked is determined during runtime, by looking in the method table. As a consequence, *this is expensive*, because we actually figure out during runtime what we need to do. We will come back to this issue later.

Comparing OO-style vs datatypes in functional programming Let us assume we want to write an application for human resources department at a hospital. There are two functionalities we want: 1) we want to be able to display the name of a doctor, nurse and an orderly. 2) we want to be able to print their salary. In a language like SML, we start by defining a datatype of `personnel` as follows:

```
datatype personnel = Doctor of string | Nurse of string | Orderly of string
```

Next, we can write the two functions we required:

```
fun display(Doctor n) = print (``Doctor's name : `` ^ n)
  | display(Nurse n) = print (``Nurse's name : `` ^ n)
  | display(Orderly n) = print (``Orderly's name : `` ^ n)
```

If we want a function which returns their salary we just write another function:

```
fun salary(Doctor _ ) = 500 000
  | salary(Nurse _ ) = 50 000
  | salary(Orderly _ ) = 25 000
```

The code for defining the salary or displaying the name is all in one place. How would one approach this problem in OO? In OO programming, we define a class for each category.

```
class Doctor{
  private N string;
  public void display() { ... }
  public void salary() { ... }
}

class Nurse{
  private N string;
  public void display() { ... }
  public void salary() { ... }
}

class Orderly{
  private N string;
  public void display() { ... }
  public void salary() { ... }
}
```

The data forms the cluster, but the code which defines the functionality is broken into pieces! It is scattered over the different classes.

The lesson to learn: Each programming language paradigm supports a very different style. The functional programming style makes it easy to add new operations and functions on the data. However, when we want to add a new element to the datatype (for example if we want to add a new category `surgeon`), we need to modify each function which operates on the datatype `personell`. This is a bit awkward. On the other hand typed languages such as SML actually warn the user if not all cases have been covered, and hence provide actually a way to easily debug existing code.

Object-oriented organization makes it easy to add a new category for surgeons. We just need to define a new class together with the functionality needed for surgeons. However, OO organization makes it hard to modify or add operations for all personell, since it requires us to visit each class and add the appropriate functionality. To easily add functionality in OO

programs and keep track of where we need to modify the code once we want to change the some functionality, several techniques have been proposed. Aspect-oriented programming is one such technique.