ITEC 1630

Yves Lespérance

Lecture Notes

Week 3 — Inheritance, Interfaces, & Polymorphism

Readings: Horstmann Ch. 13 and 11

Inheritance

Often, we need to define a class that is similar to an existing class; it just adds a few new attributes or methods, or implements existing methods differently.

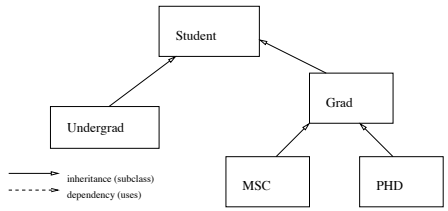
Instead of having to define the new class from scratch, OOP supports declaring the new class as a *subclass* or *specialization* of the old one.

Then the subclass *inherits* all the attributes and methods of the original class, can add new ones, as well as *override* the implementation of existing methods.

This is called *inheritance*.

Code reuse is important in software engineering. This saves time and you don't have to keep track of all the places where the same code appears so you keep them "in sync".

E.g. a hierarchy of related classes:



The class `Undergrad` is a subclass or specialization of `Student`. Inversely, we say that `Student` is a *superclass* or *generalization* of `Undergrad`.

Every instance of a class is also an instance of all its superclasses (similar to sets). So an instance of `Undergrad` is also an instance of `Student`. An instance of `MSC` is also an instance of `Grad` and `Student`.


```
public class Student
{  //constructor
   public Student(String aName, long aNumber)
   {  name = aName;
      number = aNumber;
   }
   // instance methods
   public void setName(String aName)
   {  name = aName;
   }
   public String getName()
   {  return name;
   }
   public void setNumber(long aNumber)
   {  number = aNumber;
   }
   public long getNumber()
   {  return number;
   }
   public String toString()
   {  return "Student[name=" + name +
             ",number=" + number + "]";
   }
   // instance attributes
   private String name;
   private long number;
}
```

```
public class Undergrad extends Student
{  //constructor
   public Undergrad(String aName, long aNumber,
      String aMajor, int aYear)
   {  ...
   }
   // instance methods
   // setName, getName, setNumber, & getNumbers
   // are inherited from superclass Student
   // new methods
   public void setMajor(String aMajor)
   {  major = aMajor;
   }
   public String getMajor()
   {  return major;
   }
   public void setYear(int aYear)
   {  year = aYear;
   }
   public int getYear()
   {  return year;
   }
   // need to override toString method from Student

   // instance attributes
   // name and number are inherited from Student
   private String major;
   private int year;
}
```



```
public class Eg
{  public static void main(String[] args)
   {  Student s1 = new Student("John",202123456);
      System.out.println("s1 is " + s1.getName());
      Undergrad s2 = new Undergrad(
         "Mary",201234567,"ITEC",1);
      System.out.println("s2 is "+s2.getName());//inherited
      s2.setName("Mary Ellen");//inherited
      System.out.println("s2's year is " +
                  s2.getYear());//new method
   }
}
```


Overriding Methods

Sometimes when we define a subclass, a method inherited from the superclass is inappropriate and we want to change it (e.g., `toString`). We can do this by providing a new definition for the method in the subclass. The new definition *overrides*, i.e. replaces the inherited one. E.g.

```
public class Undergrad extends Student
{  ...
   // override toString method from Student
   public String toString()
   {  return "Undergrad[name=" + getName() +
             ",number=" + getNumber() +
             ",major=" + major +
             ",year=" + year + "]";
   }
   ...
}
```



```
public class Eg
{  public static void main(String[] args)
   {  Student s1 = new Student("John",202123456);
      System.out.println(s1.toString());//calls Student's
      Undergrad s2 = new Undergrad(
         "Mary",201234567,"ITEC",1);
      System.out.println(s2.toString());//calls Undergrad's
   }
}
```

Note that the *private* attributes of the superclass, e.g. `name`, are *not visible* in the subclass; you must use a public method to retrieve their value, e.g. `getName()`.

The keyword `super` refers to the object as an instance of the superclass. It can be used to invoke overridden methods. E.g. another way to define `toString` in `Undergrad`:

```
public class Undergrad extends Student
{  ...
   // override toString method from Student
   public String toString()
   {  // call superclass's method
      String s = super.toString();
      // make appropriate changes & return
      s = s.substring(s.indexOf("["),s.length()-1);
      return "Undergrad" + s +
             ",major=" + major +
             ",year=" + year + "]";
   }
   ...
}
```


Overloaded vs Overridden Methods

It is important to distinguish between overloaded and overridden methods. As we have seen earlier, we can define several versions of a method that work with different types of arguments. This is called *overloading* the methods. E.g. the overloaded constructors:

```
public Person(){...}
public Person(String n, int a){...}
```


Another e.g.: we already have

```
public void setYear(int aYear)
{  year = aYear;
}
```

We could add:

```
public void setYear(){
  setYear(1); // default value
}
```

Then the user can write:

```
s1.setYear(); // sets year to 1
s1.setYear(n); // sets year to n
```


The *signature* of a method is the number and types of its parameters and their ordering. E.g.

1st `Person` constructor:	`()`
2nd `Person` constructor:	`(String,int)`
original `setYear`:	`(int)`
2nd `setYear`:	`()`
perhaps a 3rd `setYear`:	`(String)`

When overloading methods, you are defining several methods with the same name that will be available *in the same class*. This is only possible when the signatures of the methods are *different*.

To decide which overloaded method to call, the compiler looks at the number and types of the arguments. If the signatures are the same, it cannot determine which method is called.

In contrast, *overridden* methods have the *same signature*. The method in the subclass replaces that in the superclass (even though the superclass's version is still available using `super`). E.g.

```
public class Parent
{  public void meth() // #1
   {  ...
   }
   public void meth(int n) // #2, overloads #1
   {  ...
   }
   ...
}
public class Offspring extends Parent
{  public void meth(int n) // #3, overrides #2
   {  ...
   }
   ...
}
...
// in main
Parent o1 = new Parent();
Offspring o2 = new Offspring();
o1.meth();   // calls #1
o1.meth(31); // calls #2
o2.meth();   // calls #1
o2.meth(29); // calls #3
```


Inheritance and Attributes

As we saw previously, attributes defined in a superclass are inherited by the subclass, but are not directly accessible (assuming they are `private`). They can only be accessed through inherited `public` methods.

You can also redefine an attribute in the subclass, but unlike for methods, the new attribute does not replace the original one. Instead, you will have two attributes with the same name, with the one defined in the subclass *shadowing* the one defined in the superclass. E.g.


```
public class Parent
{  ...
   public setAtt(int n)
   {  att = n;
   }
   private int att;
}
public class Offspring extends Parent
{  public void meth()
   {  att = 4;// sets Offspring's att
      setAtt(3);// sets Parent's att
   }
   private int att;
}
// in main
Offspring o = new Offspring();
o.meth();
```

If the superclass's attribute is not private, you can refer to it as `super.att`.

Constructors and Inheritance

An instance of a class C in a hierarchy is also an instance of all of C's superclasses (all the way to the top). This fits with the view of classes as sets and the specialization relation as subset.

If an object is to be an instance of a class, then the class's constructor should be called when the object is created. For an instance of a class C in a hierarchy, the constructor of C *and the constructors of all of C's superclasses should be called*. Java *requires* this.

You can call a constructor of C's superclass by putting `super(...)` in C's constructor. This must be the first statement in C's constructor. E.g.

```
public class Student
{  public Student()
   {  name = "UNKNOWN";
      number = -1;
   }
   public Student(String aName, long aNumber)
   {  name = aName;
      number = aNumber;
   } ...
   private String name;
   private long number;
}
public class Undergrad extends Student
{  public Undergrad()
   {  super();// calls Student's 0 args constructor
      major = "general";
      year = 1;
   }
   public Undergrad(String aName, long aNumber, String aMajor, int aYear)
   {  super(aName, aNumber);// calls Student's
                            // 2 args constructor
      major = aMajor;
      year = aYear;
   } ...
   private String major;
   private int year;
}
// in main
Undergrad u = new Undergrad("John", 201234567, "ITEC", 1);
```


If you leave out the call to the superclass's constructor, Java automatically inserts `super()` at the beginning of the subclass's constructor, i.e. a call to the superclass's 0 arguments constructor.

If there are more than one superclass, each ancestor class's constructor is called in turn. E.g.

```
public class PartTimeUndergrad extends Undergrad
{  //constructors
   public PartTimeUndergrad()
   {  super();// calls Undergrad's 0 args constructor
      courseLoad = 2.5;
   }
   public PartTimeUndergrad(String aName, long
      aNumber, String aMajor, int aYear, double aLoad)
   {  super(aName, aNumber, aMajor, aYear);
         // calls Undergrad's 4 args constructor
      courseLoad = aLoad;
   }
   ...
   private double courseLoad;
}
// in main
PartTimeUndergrad p = new PartTimeUndergrad(
   "John", 201234567, "ITEC", 1, 3.5);
```


Constructors are never inherited.

However, if you don't define a constructor for a class, Java automatically provides a default 0 arguments constructor that initializes the attributes to default values, 0 for numbers, `false` for `boolean`, and `null` for objects.

Problem: define a method

```
public double calculateFees(
     double courseload)
```

for `Student` and `Undergrad` which returns the fees to be paid by the student depending on on his/her courseload.

Suppose that for students generally, the fees are $800 per course and that for undergraduates, there is an additional incidental charge of $100 for first year students and $150 for students in later years.

```
public class Student
{  ...
   public double calculateFees(double courseload)
   {  final double FEE_PER_COURSE = 800;
      return FEE_PER_COURSE * courseload;
   }
   ...
}
public class Undergrad extends Student
{  ...
   // override Student's calculateFees method
   public double calculateFees(double courseload)
   {  final double INCIDENTAL_FEE_Y1 = 100;
      final double INCIDENTAL_FEE_Y_GT_1 = 150;
      double fee = super.calculateFees(courseload);
      if (year == 1)
         fee = fee + INCIDENTAL_FEE_Y1;
      else
         fee = fee + INCIDENTAL_FEE_Y_GT_1;
      return fee;
   }
   ...
}
// in main
Student s = new Student("Mary", 202345678);
System.out.println(s + " fees: " + s.calculateFees(4.5));
Undergrad u = new Undergrad("John", 201234567, "ITEC", 1);
System.out.println(u + " fees: " + u.calculateFees(4.5));
```


Polymorphism

In OOP, classes correspond to types. When a class `Co` is a subclass of a class `Cp`, then the type `Co` is a subtype of `Cp`. If you have a reference to an instance of `Co`, it can be assigned to a variable of type `Cp` and manipulated as if it were an instance of the class `Cp`. E.g.

```
Student s1;
s1 = new Undergrad("Mary",201234567,"ITEC",1);
```

The ability to use a value of a more specific type where one of a more general type is expected is called *polymorphism*, since the general type comes in "many forms".

You can also assign a reference to an instance of superclass (a supertype) to a variable of a subclass (a subtype) provided you perform a type cast. E.g.


```
Student s1;
s1 = new Undergrad("Mary",201234567,"ITEC",1);
Undergrad u1;
u1 = (Undergrad) s1;
```

Note that if the object being cast does not belong to the type given (e.g. `s1` is not an instance of `Undergrad` or one of it's subclasses) an exception will be thrown when the program is run.

To be safe, you can check whether the object belongs to the right class using the `instanceof` operator before you perform the cast, e.g.

```
if (s1 instanceof Undergrad)
   u1 = (Undergrad) s1;
```


Dynamic Method Binding

When you call an overridden method on a reference of polymorphic type (e.g. `s1.toString()`), the version of the method that gets called depends on the actual type of the object referred to at run time; it is not necessarily that of the declared type. E.g.

```
Student s1;
s1 = new Student("John",202123456);
System.out.println(s1.toString());// calls Student's
s1 = new Undergrad("Mary",201234567,"ITEC",1);
System.out.println(s1.toString());// calls Undergrad's
```

This approach to selecting which version of an overridden method to call is named *dynamic* or *late method binding*. The term polymorphism is often used to refer to this specific feature of polymorphic method calls.

In this e.g. the compiler could have determined the actual type of the object, but in general, this is not the case, e.g.

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in);
Student s1;
System.out.print("Enter student's name: ");
String name = in.nextLine();
System.out.print("Enter student's number: ");
long number = in.nextLong();
System.out.print("Is student an undergrad (y/n)? ");
String ans = in.nextLine();
if (ans.equals("y"))
{  System.out.print("Enter student's major: ");
   String major = in.nextLine();
   System.out.print("Enter student's year: ");
   int year = in.nextInt();
   s1 = new Undergrad(name,number,major,year);
} else
   s1 = new Student(name,number);
System.out.println(s1.toString());// polymorphic call
```


In C++, the programmer specifies whether calls to a method should be interpreted statically or dynamically. Methods that are selected dynamically are called *virtual* methods.

Note that even though Java uses dynamic method binding, the compiler requires that for every method call, the method invoked must be defined for the reference's declared type. E.g.

```
Student s1;
s1 = new Undergrad(
  "Mary",201234567,"ITEC",1);
String m = s1.getMajor(); // error
// getMajor is not defined for Student
```


Polymorphism and Heterogenous Collections

Polymorphism is particularly useful for dealing with heterogenous collections of objects, i.e., collections that contain objects of different types. Here is an e.g. that uses an array to store an heterogenous collection of students:

```
Student[] stuGroup = new Student[5];
stuGroup[0] = new Student("John",201234567));
stuGroup[1] = new Undergrad("Mary",202123456,"ITEC",1));
stuGroup[2] = new Student("Ellen",203456789));
stuGroup[3] = new Undergrad("Paul",202123456,"ITEC",2));
for (int i = 0, i < 4, i++)
   System.out.println(stuGroup[i].toString());
   // calls Student or Undergrad toString method
   // according to type of each object
```

This works even if the data is read in and the types of the objects created depend on the input — the method selection is done at run time.

Abstract Classes

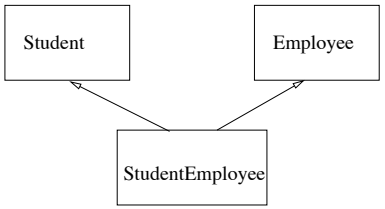
Abstract classes are "incompletely defined" classes. They contain a partial definition, a sort of template, that will be completed when we define concrete subclases.

Abstract classes can contain `abstract` methods, i.e. methods for which we only provide the signature, not the code. The code will be provided by the subclasses. Abstract classes can also contain normal method definitions and attribute declarations like ordinary classes. If a class contains even one abstract methods, the class must be declared `abstract`.

Because they are "incomplete", abstract classes cannot have any direct instances, i.e. instances that are not also instances of some concrete subclass.

Multiple Inheritance and Java Interfaces

Sometimes, it would be useful to *inherit from several classes*. E.g.



But it is mot exactly clear what should be inherited when several superclasses define methods or attributes of the same name.

Languages like C++ allow *multiple inheritance* and have complex rules to resolve inheritance conflicts.

To avoid this complexity, Java only allows a very limited form of multiple inheritance through the `interface` mechanism. A Java `interface` is like a purely abstract class. It can only contain abstract method declarations (i.e. signature only, no implementation) and constant declarations; these are all automatically public. E.g.

```
public interface Comparable
{  int compareTo(Object other);
}

public interface Clonable
{  Object clone();// returns copy of this instance
}

public interface Enumeration // for enumerating
{  boolean hasMoreElements();// a collection
   Object nextElement();
}
```


A class that `implements` an `interface` must supply implementations for the methods defined in the `interface`. E.g.

```
public class Student implements Comparable
{  ...
   public int compareTo(Object other)
   {  Student otherStud = (Student)other;
      if (number < otherStud.number) return -1;
      if (number > otherStud.number) return 1;
      return 0;
   }
   ...
}
```

A class can `implement` several `interfaces`, e.g.

```
public class Student implements Comparable, Clonable
{...}
```

`interfaces` can also be organized in a hierarchy; an interface can `extend` several parent interfaces.

Interfaces are often useful for designing classes that can work with many different types of objects. The `DataSet` class of Horstmann Ch. 11 is a good example of this. It represents a collection to which objects can be added and can return the average and maximum of the collection. This can work for any collection of objects that are “measurable”. We would prefer not to have to commit to a particular type of objects, e.g. bank accounts, coins, etc.

To do this the version in Sec. 1 defines an interface:

```
public interface Measurable
{  double getMeasure();
}
```

The `DataSet` class can then use this interface as the type of objects that can be added to the collection or saved as the maximum. Any class that implements the interface can be added to the collection. `DataSet` uses the `getMeasure()` method to obtain a value that can be used to compute the average and maximum.

A class using an interface type is another case of polymorphism.

A second version of `DataSet` in Sec. 4 is even more general, allowing multiple ways of measuring a given thing.

The Class `Object`

There is a root to the class hierarchy in Java, a class called `Object`. Every class that is defined without a superclass being specified is automatically made to be a subclass of `Object`.

Because it is so general, the class `Object` does not provide much. Its most useful methods are discussed below. It is generally a good idea to override these when you define a class.

`String toString()` generates a string representing the object, by default *ClassOfObject@HashCodeOfObject*, e.g. `Student@123f56`. This is not very meaningful and we have seen how to override it in `Student`.

`boolean equals(Object other)` tests whether this instance is equal to `other`. By default it returns true iff `this` and `other` are the same reference. We generally override it to make a comparison based on the objects' attributes, e.g.

```
public class Student
{  ...
   public boolean equals(Object other)
   {  if (other instanceof Student)
         return number == ((Student)other).number;
      else
         return false;
   }
   ...
}
```


`Object clone()` returns a new object which is a copy of this instance. By default this is a *shallow copy*, i.e. attributes which are themselves objects are not cloned, only their references are copied. If an object has attributes which are mutable objects, it is generally better to make a *deep copy*.

E.g. suppose that `Student` had an additional attribute `nextofKin` of type `Person`. To make `clone` return a deep copy, we override it as follows:

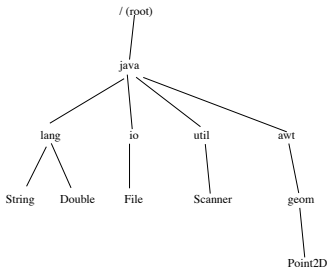
```
public class Student implements Clonable
{  ...
   public Object clone()
   {  try
      {  Student copy = (Student)super.clone();
         // or = new Student(name,number);
         copy.nextOfKin = (Person)nextOfKin.clone();
         // assumes Person is Cloneble
         return copy;
      }
      catch (CloneNotSupportedException e)
      {  return null; // can't really happen
      }
   }
}
```

If a class uses `clone` from Object, it must implement the `Clonable` interface and catch the `CloneNotSupportedException`.

The `Object` class can also be used for defining very general data structures, e.g. an array whose elements are of type `Object`, or an `ArrayList` of `Object`. If you have such a data structure, then you can put any kind of object into it, `Student`s, `Name`s, `Integer`s (wrapper class), etc. In many cases though, it is better to use generic classes that take a type parameter, e.g `ArrayList<BankAccount>`.

Packages

In Java, classes are grouped into *packages*. Package names correspond to the directory hierarchy into which classes are organized.



We've already seen how to use the `import` statement to avoid having to refer to a class using its full name.

Java looks for classes starting in the directories that are on the `CLASSPATH`, e.g.

```
CLASSPATH=/cs/dept/www/java/bin/york/york.jar:\
/cs/home/fac1/lesperan/java:.;
```

To create a package, you put a `package` declaration at the beginning of each file containing the definition of a class in the package, and you install the files in the appropriate directories. E.g. for the class `Student`, we put the following in `/cs/home/fac1/lesperan/java/yves/Student.java`:

```
package yves;
public class Student{
...
}
```

Classes that are not explicitly put in a package go into an unnamed package.

The Java API provides a set of standard classes grouped in several packages, e.g.

```
java.util
java.io
java.lang
```

The `java.lang` package contains the core classes (e.g. `String`, `System`, `Math`, etc.) and is automatically imported.

Visibility/Scope of Class Attributes and Methods

Besides `public` and `private`, there are other access modifiers that can be used to control the visibility of class constants, variables, and methods:

- `public`: visible everywhere,
- `protected`: visible in all classes of the same package and in subclasses in other packages,
- default: visible in all classes of the same package,
- `private`: visible only in the class itself.

`protected` can be used on attributes to give subclasses access to the implementation of a class. But this is generally a bad idea because you cannot prevent anyone from defining a subclass of one of your classes. If you give them access to your implementation, then you can't change it without affecting them.

If you want define classes that work closely together, it is often better to use *inner classes* (see below).

As well, access to classes can be controlled. `public` classes are visible everywhere. Classes that are not labeled `public` are only visible in their package.

Also, classes that are declared `final` cannot be extended, i.e. no one can create subclasses.

You can also prevent someone from overriding a method in a subclass that they define by declaring the method `final`.

Inner Classes

An *inner class* is a class that is defined inside another class (or a method of another class). This is usually done when the inner class is only meant to be used by the enclosing class, i.e. you don't what to make it available to others (as for a private method).

The third version of Horstmann's `DataSet` example in Ch. 11 Sec. 5 defines the class that implements the `Measurer` interface as an inner class, as it is not meant to be used in other contexts.

Some graphical shapes are defined as inner classes, e.g. `Ellipse2D.Double`.

There are restrictions on what variables inner classes can access (see Horstmann Ch. 11 Sec. 7).