

## CSE 1030

Yves Lespérance

### Lecture Notes

#### Week 6 — Implementing Inheritance

Recommended Readings:

Van Breugel & Roumani Ch. 5 and Savitch Ch. 7, 8, and 9 and Sec. 13.1

*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. `RewardCard` *is a* `CreditCard`.

Can represent is-a/subclass relationships in UML class diagrams.

## 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.

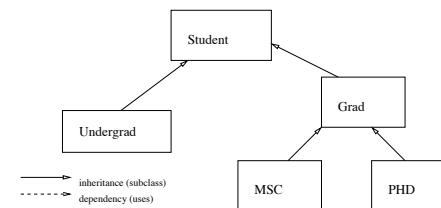
Instead of having to define the new class from scratch, OOP supports declaring the new class as a *subclass* or *specialization* of the old one.

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*.

We say that the subclass *is a* the superclass.

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

```

5

```

public class Undergrad extends Student
{ // instance attributes
  // name and number are inherited from Student
  private String major;
  private int year;
  //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)
  { this.major = aMajor;
  }
  public String getMajor()
  { return this.major;
  }
  public void setYear(int aYear)
  { this.year = aYear;
  }
  public int getYear()
  { return this.year;
  }
  // need to override toString method from Student
}

```

6

## 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 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,"compsci",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
  }
}

```

7

```

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

```

8

```

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,"compsci",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()`.

9

## 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)
{
    this.year = aYear;
}

```

11

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=" + this.getMajor() +
            ",year=" + this.getYear() + "]";
    }
    ...
}

```

10

We could add:

```

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

```

Then the user can write:

```

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

```

12

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

```
1st Person constructor: Person()
2nd Person constructor: Person(String,int)

original setYear:      setYear(int)
2nd setYear:          setYear()
perhaps a 3rd setYear: 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.

13

## 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.

15

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
```

14

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

16

## 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.

17

```
public class Undergrad extends Student
{ private String major;
  private int year;

  public Undergrad()
  { super();// calls Student's 0 args constructor
    this.setMajor("general");
    this.setYear(1);
  }
  public Undergrad(String aName, long aNumber,
    String aMajor, int aYear)
  { super(aName, aNumber);// calls Student's
    // 2 args constructor
    this.setMajor(aMajor);
    this.setYear(aYear);
  } ...
}

// in main
Undergrad u = new Undergrad(
  "John", 201234567, "compsci", 1);
```

19

```
public class Student
{
  private String name;
  private long number;

  public Student()
  { this("UNKNOWN",-1);
  }
  public Student(String aName, long aNumber)
  { this.setName(aName);
    this.setNumber(aNumber);
  }
  ...
}
```

18

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.

20

```

public class PartTimeUndergrad extends Undergrad
{ private double courseLoad;
  //constructors
  public PartTimeUndergrad()
  { super();// calls Undergrad's 0 args constructor
    this.setCourseLoad(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
    this.setCourseLoad(aLoad);
  }
  ...
}
// in main
PartTimeUndergrad p = new PartTimeUndergrad(
  "John", 201234567, "compsci", 1, 3.5);

```

21

**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.

23

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.

22

```

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, "compsci", 1);
System.out.println(u + " fees: " + u.calculateFees(4.5));

```

24

We can use delegation in defining equals. E.g. in Rectangle, we have:

```
public boolean equals(Object object)
{
    boolean equal;
    if (object != null && this.getClass() == object.getClass())
    {
        Rectangle other = (Rectangle) object;
        equal = (this.getWidth() == other.getWidth()) &&
            (this.getHeight() == other.getHeight());
    }
    else
    {
        equal = false;
    }
    return equal;
}
```

25

## Polymorphism

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

```
Student s1;
s1 = new Undergrad(
    "Mary", 201234567, "compsci", 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 super-type) to a variable of a subclass (a subtype) provided you perform a type cast. E.g.

27

In GoldenRectangle, we can override this with:

```
public boolean equals(Object object)
{
    boolean equal;
    if (object != null && this.getClass() == object.getClass())
    {
        GoldenRectangle other = (GoldenRectangle) object;
        equal = super.equals(other) &&
            this.getWeight() == other.getWeight();
    }
    else
    {
        equal = false;
    }
    return equal;
}
```

26

```
Student s1;
s1 = new Undergrad(
    "Mary", 201234567, "compsci", 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 its 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;
```

But note that using `instanceof` in defining `equals` destroys symmetry; see lecture notes p. 107.

28

## 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,"compsci",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.

29

## 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 subclasses. This is often useful to avoid code duplication.

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.

31

But note that dynamic method binding searches for method signature determined at compilation time, so it may not get the expected version of a method. E.g. from the `type` library:

`CreditCard` provides `isSimilar(CreditCard)`

`RewardCard` overloads this with `isSimilar(RewardCard)`

```
CreditCard c1 = new RewardCard(9, "adam");
CreditCard c2 = new RewardCard(9, "adam");
c1.charge(100);
c1.pay(100);
System.out.println(c1.isSimilar(c2)); // CreditCard version
System.out.println(c1.isSimilar((RewardCard)c2)); // CreditCard version!!
System.out.println(((RewardCard)c1).isSimilar(c2)); // CreditCard version
System.out.println(((RewardCard)c1).isSimilar((RewardCard)c2)); // RewardCard versi
```

See Java By Abstraction p. 341-343.

30

See for example the abstract class `RightPrism` class from lecture notes.

It does not make sense to create instances of `RightPrism` because it does not specify what the shape is really like.

But we define it as an abstract class to collect code that is common between its concrete subclasses `RightRectangularPrism` and `RightTriangularPrism`.

We can include a method declaration

```
public abstract int getBaseArea();
```

that needs to be defined by the concrete subclasses.

This method can even be used by another method `getVolume()`.

32



## **Inheritance and Preconditions/Postconditions**

In a subclass, if we override a method from the superclass, we can weaken its precondition, but not strengthen it. The method of the subclass should work whenever the superclass's version would because an instance of the subclass is an instance of the superclass.

For a postcondition, we have the opposite: if in a subclass we override a method from the superclass, we can strengthen its postcondition, but not weaken it. The new version should achieve all the effects guaranteed by the superclass's version and possibly more.