# Lecture 9: Inheritance and Polymorphism Using Processing

Based on *Introduction to Java Programming*, Y. Daniel Liang, Brief Version, 9/E

## Topics Covered

<table>
<thead>
<tr>
<th></th>
<th>Inheritance</th>
<th>Object Class and Its Methods</th>
<th>Polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1. Inheritance</td>
<td>2. Object Class and Its Methods</td>
<td>3. Polymorphism</td>
</tr>
<tr>
<td></td>
<td>1.1 What is Inheritance</td>
<td>3.1 Casting and <code>instanceof</code> Operator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2 <code>super</code> Keyword</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3 Superclass Methods and Data Fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4 Constructors and Inheritance</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.5 Overriding Methods of The Superclass</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1 Inheritance

1.1 What is Inheritance

Inheritance

- defining new classes from existing ones;
- defining general classes that can be extended into more specialized classes.

Inheritance models *is-a relationship*. For example

- a student *is-a* person, so a `Student` class can inherit from a `Person` class;
- an employee *is-a* person, so an `Employee` class can inherit from a `Person` class;
- a professor *is-an* employee, so a `Professor` class can inherit from an `Employee` class (which itself inherits from a `Person` class);
- a student is *NOT* a professor (even though they might share many characteristics) so the two classes should not be related by inheritance relationship.

If `C2` class extends `C1` class, `C2` is called a **subclass** and `C1` is called the **superclass**.

Defining superclasses and subclasses:

```java
public class C1 {
    // class definition
}

public class C2 extends C1 {
    // class definition
}
```

The keyword `extends` indicates that the class `C2` *extends/inherits from* the class `C1`. `C2` has access to all public and protected data fields and methods of class `C1`. It does not have access to private data fields and methods of `C1`.

**NOTE:** In Java a class can inherit from only one class, i.e., in the above example, `C2` cannot extend any other class than `C1`. This is called **single inheritance**. There are other programming languages in which multiple inheritance is allowed.

**NOTE:** Contrary to a misleading name, a subclass is not a subset of the superclass. In fact, it usually contains more than the superclass.

1.2 `super` Keyword

The keyword `super` is used to access superclass constructors, methods and data fields (the ones that are inherited, not the private ones).

**Examples:**

```
super();
```

invokes the default constructor of the superclass (assuming it exists). This call can be made only from within a constructor of a subclass.

```
super(paramList);
```
invokes the constructor with parameters of the superclass (assuming it exists). This call can be made only from within a constructor of a subclass.

    super.methodName(paramList);

invokes a method of the superclass. This call can be made from anywhere in the subclass.

    super.dataField;

accesses a dataField of the superclass. This call can be made from anywhere in the subclass.

### 1.3 Superclass Methods and Data Fields

A subclass inherits all public and protected methods and data fields of its superclass. That means that a subclass can use the methods and data fields that it inherits without defining them.

**Example:**

```java
public class C1 {
    protected int x;
    public void setX ( int x ) {
        this.x = x;
    }
    //the rest of class definition
}

public class C2 extends C1 {
    protected float c;
    public C2 ( float c, int x ) {
        this.c = c;
        super.setX(x);
    }
    //the rest of class definition
}
```

In the above code, the class C2 can access the data field `x` and the method `setX()` without defining them because they are inherited from C1.

The keyword `protected` is used to indicated data fields and methods that are accessible only from within the class and/or classes that inherit from it. Those data fields and methods are not accessible from outside of the class (they are `private` outside of subclasses).

One can use the keyword `super` to indicate the inherited methods but it is not necessary, unless the methods are overridden (see below). The class C2 above could be rewritten as

```java
public class C2 extends C1 {
    protected float c;
    public C2 ( float c, int x ) {
        this.c = c;
        super.setX(x);
    }
    //the rest of class definition
}
```

The two definitions are equivalent.
1.4 Constructors and Inheritance

Constructors of the superclass are not inherited by its subclass. They can be invoked using the super keyword from within subclass constructors.

Example:

Given the following code

```java
public class C1 {
    public C1 () { //do something }  //class definition
    public C1 ( int c ) { //do something }
    //class definition
}

public class C2 extends C1 {
    public C2 ( int c, int x ) {
        super(c);
        //do something
    }
    //class definition
}
```

an object of the C2 class can be used using two parameter constructor, BUT NOT one parameter constructor or the default constructor.

```java
C2 object1 = new C2 ( 3, 5 ); //is valid
C2 object2 = new C2 ( 3);     //is NOT valid
C2 object3 = new C2 ();        //is NOT valid
```

Constructor chaining: constructing an instance of a class invokes the constructors of ALL the superclasses along the inheritance chain.

A constructor may

- invoke an overloaded constructor (of its own class), or
- invoke its superclass constructor (this has to be done in the first line of the constructor).

If neither of these happens explicitly, the Java compiler automatically adds super() as the first statement in the constructor. (This may cause problems if the superclass does not have a default constructor.)

See ConstructorChaining.java

1.5 Overriding Methods of The Superclass

Overriding is redefining a method of a superclass in a subclass. The overridden method has to have the same name, parameter list and return type as the method in the superclass.

Example:

```java
public class C1 {
    public void sayHi() {
        System.out.println("C1 says hi");
    }
}
```
The class \texttt{C2} overrides the inherited method \texttt{sayHi()} and uses the \texttt{super} keyword to access the overridden method (in this case the keyword \texttt{super} is not optional).

- An overridden method of a superclass can be accessed using the \texttt{super} keyword. (This only works over one level of inheritance, i.e., there is not \texttt{super.super.____}.

- Static methods of the superclass are not overridden (well, depends on how you look at it). If a subclass defines a static method whose name matches the name of the static method in the superclass, one needs to use the name of the superclass, not the keyword \texttt{super} to access it.

\begin{verbatim}
SuperClassName.staticMethodName(...);
not, super.staticMethodName();
\end{verbatim}

- Use the override annotation, @Override, when declaring methods that override methods of the superclass. This way the Java compiler double checks that your method truly overrides the method of the superclass.

**Overriding vs. overloading**

If the method in a subclass does not match the method of its superclass exactly in its parameter list, then it only overloads the method of the superclass (both still can be accessed).

**Example:**

```java
public class Test {
    public static void main( String [] args ) {
        C2 c = new C2();
        c.foo( 10 );
        c.foo( 10.0 );
    }
}

class C1 {
    public void foo (double x ) {
        System.out.println("C1.foo() called ");
    }
}

class C2 extends C1 {
    //this is overriding
    public void foo (double x ) {
        System.out.println("C2.foo() called ");
    }
}

Output:
C2.foo() called
C2.foo() called
```

```java
public class Test {
    public static void main( String [] args ) {
        C2 c = new C2();
        c.foo( 10 );
        c.foo( 10.0 );
    }
}

class C1 {
    public void foo (double x ) {
        System.out.println("C1.foo() called ");
    }
}

class C2 extends C1 {
    //this is overloading
    public void foo (int x ) {
        System.out.println("C2.foo() called ");
    }
}

Output:
C2.foo() called
C1.foo() called
```
2 Object Class and Its Methods

Every class in Java inherits automatically from the Object class (you do not need to do anything in order for your class to inherit from Object).

The Object class has several methods that every other class inherits. Most of the times, unless the subclass overrides these methods they are not very interesting. See the documentation for the Object class at http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

The toString() method that we have been using for some time now, is inherited from the Object class. Whenever you write your own version of the toString() method, you are overriding the one in the Object class.

3 Polymorphism

Polymorphism - a variable of a superclass / supertype / base type can refer to a variable of a subclass / subtype / derived type.

Dynamic binding - a method can be implemented in several classes along the inheritance chain. JVM decides which method should run at runtime.

A variable has two (possibly different) types associated with it:

- **declared type** - the type listed in variable declaration,
- **actual type** - the type of object that variable references.

The method that is invoked by a variable at runtime is determined by its actual type, but the compiler can only determine appropriateness of method calls based on the declared type.

3.1 Casting and instanceof Operator

Person p = new Student ( ... );

is referred to as **implicit casting**. It is done automatically (you do not need to tell the compiler to do it) because an instance of the class Student is an instance of the class Person (student is a person).

On the other hand, the statements

```java
Person s = new Student ( ... );
Student s = p;
```

will cause a compiler error because the compiler only uses the declared type of variable p and since person is not a student it cannot perform the assignment. In this particular case p references a Student object, but it might not always be the case (and the compiler certainly cannot know that). The way around it is explicit casting:

```java
Person s = new Student ( ... );
Student s = (Student) p;
```

The instanceof operator allows you to make sure that the particular variable references the object that is of a particular type.

```java
refVariable instanceof ClassName
```

evaluates to true if refVariable references an instance of class ClassName, and false otherwise.