Introduction to Computer Programming

Using the Java Programming Language

8. Inheritance

Inheritance is a major component of a powerful and popular programming technique known as object-oriented-programming. Inheritance is a technique that will allow you to define a very general class and then later define more specialized classes by simply adding some new details to the older more general class definitions. This saves work because the more specialized class inherits all the properties of the general class and you, the programmer, need only program the new features.

Key Topics

After reading this section, you should be familiar with the following topics:

- Single Inheritance
- Class Hierarchies
- Using Inheritance
- Method Overriding

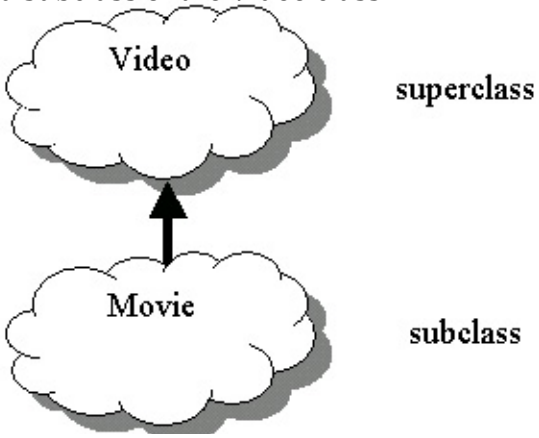
Single Inheritance

1.Inheritance enables you to define a new class based on a class that already exists. The new class will be similar to the existing class, but will have some new characteristics. This makes programming easier, because you can build upon your previous work instead of starting out from scratch.

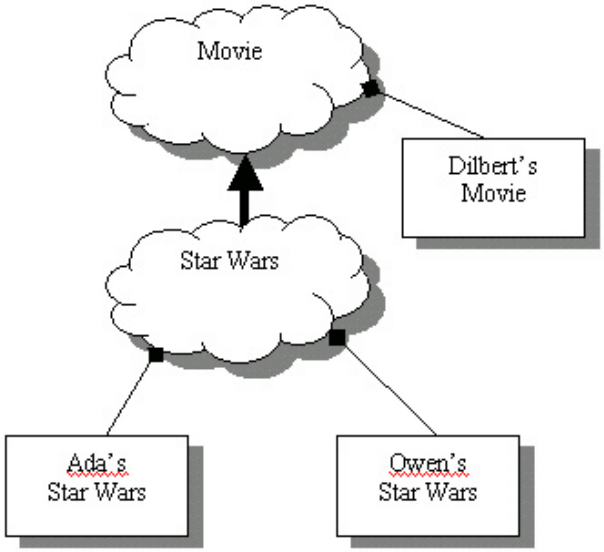
2. The class that is used as a basis for defining a new class is called a superclass (or *parent class* or *base class*). The new class based on the superclass is called a subclass (or *child class* or *derived class*.)

3. In Java, (unlike with humans) children inherit characteristics from just one parent. This is called single inheritance. Some languages allow a child to inherit from more than one parent. This is called multiple inheritance. With multiple inheritance, it is sometimes hard to tell which parent will contribute what characteristics to the child (as with humans.) Java avoids these problems by using single inheritance.

4. The figure on the next page shows a superclass and a subclass. The line between them shows the "is a kind of" relationship. Notice that the arrow points to the superclass. The picture can be read as "a Movie is a kind of Video." The clouds represent classes. That is, the picture does not show any particular Video or any particular Movie, but shows that the class Movie is a subclass of the Video class.



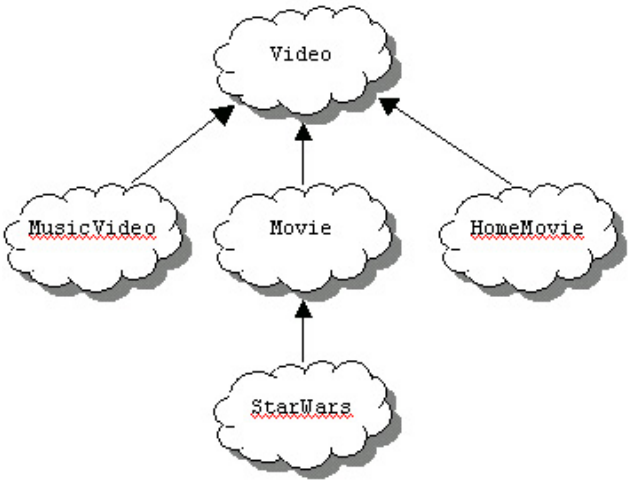
5. Inheritance is between classes, not between objects. A superclass is a blueprint that is followed when an object is constructed. A subclass of the superclass is another blueprint (that looks much like the original), but with added features. The subclass is used to construct objects that look like the superclass ' objects, but with added features.



6. The figure shows a superclass and a subclass, and some objects that have been constructed from each. These objects are shown as rectangles (to convey the idea that they are more real than the classes, which are only designs.) In the picture, "Dilbert's Movie," "Ada's Star Wars," and "Owen's Star Wars" represent actual objects. The cloudy classes represent designs that were used to construct the objects.

Hierarchies

1. In a hierarchy, each class has at most one superclass, but might have several subclasses. There is one class, at the "top" of the hierarchy that has no superclass. This is sometimes called the root of the hierarchy.



The figure shows a hierarchy of classes. It shows that "MusicVideo is a kind of Video," "Movie is a kind of Video," and that "HomeMovie is a kind of Video." It also shows that "StarWars is a kind of Movie."

2. A *derived* class is a class defined by adding instance variables and methods to an existing class. The existing class that the derived class is built upon is called the *base class*. In our example, the class Video is the base class and the classes MusicVideo, Movie, HomeMovie, and StarWars are derived classes.

3. The terms "superclass" and "subclass" are not absolute. Any class may be the superclass for a subclass derived from it. Just as with human relationships, an individual is a child to some relatives and a parent to others.

4. In Java, the syntax for deriving a child class from a parent class is:

```
class subclass extends superclass
{
    // new characteristics of the subclass go here
}
```

5. Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors - namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram above, classes `MusicVideo, Movie`, and `HomeMovie` are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class `StarWars` is a subclass of class Movie which is itself a subclass of class Video. In this case, class `StarWars` is considered to be a sub-subclass of class Video, as it is not a direct subclass. We could also say that Video is a super-superclass of `StarWars`

Using Inheritance

1. Here is a program that uses a class `Video` to represent movies available at a video rental store. The `Video` class has basic information in it, and would be OK for documentaries and instructional tapes. But movies need more information. An additional class, `Movie`, is created that is similar to `Video`, but has the name of the director and a rating.

```
class Video
{
	protected String  myTitle;     // name of the item
	protected int     myLength;    // number of minutes
	protected boolean myIsAvail;   // is the tape in the store?

	public Video(String title, int len)
	{
		myTitle = title; myLength = len ; myIsAvail = true;
	}

	public String toString()
	{
		return myTitle + ", " + myLength +
				" min. available: " + myIsAvail;
	}
}

class Movie extends Video
{
	protected String  myDirector;     // name of the director
	protected String  myRating;       // G, PG, R, or X

	// constructor
	public Movie(String title, int len, String dir, String rating)
	{
		// use the super class' constructor
		super(title, len);

		// initialize what's new to Movie
		myDirector = dir;  myRating = rating;
	}
}

class VideoStore
{

	public static void main (String args[])
	{
		Video item1 = new Video("Juiced on Java", 90 );
		Movie item2 = new Movie("Just Java", 120, "Gosling", "PG" );
		System.out.println(item1.toString());
		System.out.println(item2.toString());
	}
}
```

2. The class `Movie` is a derived class (subclass) of `Video`. An object of type `Movie` has the following members in it:

Members	
`MyTitle`	Inherited from `Video`
`myLength`	Inherited from `Video`
`myIsAvail`	Inherited from `Video`
`toString()`	Inherited from `Video`
`myDirector`	Defined in `Movie`
`myRating`	Defined in `Movie`

Both classes are defined: the Video class can be used to construct objects of that type, and now the Movie class can be used to construct objects of the Movie type.

3. The class definition for Video has a constructor that initializes the member data of Video objects. The class Movie has a constructor that initializes the data of Movie objects. The constructor for class Movie looks like this:

```
// constructor
public Movie(String title, int len, String dir, String rating)
{
      // use the super class' constructor
      super(title, len);

      // initialize what's new to Movie
      myDirector = dir;
      myRating = rating;
}
```

The statement super(title, len) invokes the base class' constructor to initialize some of the data. Then the next two statements (on one line) initialize the members that only Movie has. When super is used in this way, it must be the first statement in the subclass' constructor.

4. It is not necessary to use super; the following would also work as a constructor for Movie:

```
// constructor
public Movie(String title, int len, String dir, String rating)
{
      // initialize the inherited members
      myTitle = title; myLength = len ; myIsAvail = true;

      // initialize members unique to Movie
      myDirector = dir;  myRating = rating;
}
```

In this constructor, each variable of the newly created Movie object is set to an initial value.

5. So far, we have only seen the **public** and **private** access modifiers. There is a third access modifier that can be applied to an instance variable or method. If it is declared to be **protected**, then it can be used in the class where it is defined and in any subclass of that class. This is obviously less restrictive than **private** and more restrictive than **public**. Classes that are written specifically to be used as a basis for making subclasses often have protected members. The protected members are there to provide a foundation for the subclasses to build on. But they are still invisible to the public at large.

Method Overriding

1. A derived class can *override* a method from its base class by defining a replacement method with the same signature. For example in our `Movie` subclass, the `toString()` method contained in the `Video` superclass can not reference the new variables that have been added to objects of type `Movie`, so nothing new is printed out. We need a new `toString()` method in the class `Movie`:

```
// overrides the toString method in the parent class
public String toString()
{
      return myTitle + ", " + myLength +
                  " min. available: " + myIsAvail +
                  ", dir: " + myDirector + ", " + myRating;
}
```

2. Even though the base class has a `toString()` method, the new definition of `toString()` in the derived class will override the base class' version . The base class has its method, and the derived has its own method with the same name. With the change in the class `Movie` the following program will print out the full information for both items.

```
class VideoStore
{
      public static void main (String args[])
      {
            Video item1 = new Video("Samurai Jack", 90 );
            Movie  item2 = new Movie("Star Wars", 120, "Lucas",
"PG");
            System.out.println(item1.toString());
            System.out.println(item2.toString());
      }
}
```

Output:

```
Samurai Jack, 90 min. available: true
Star Wars, 120 min. available: true, dir: Lucas, PG
```

The line `item1.toString()` calls the `toString()` method defined in `Video`, and the line `item2.toString()` calls the `toString()` method defined in `Movie`.

3. Sometimes (as in the example) you want a derived class to have its own method, but that method includes everything the derived class' method does. You can use the super reference in this situation to first invoke the original `toString()` method in the base class as follows:

```
public String toString()
{
   return super.toString()+ ", dir: " + myDirector + ", " + myRating;
}
```

Unlike the case when super is used in a constructor, inside a method super does not have to be used in the first statement.

Summary

Inheritance represents the "is a kind of" relationship between types of objects. In practice it may be used to a new features to an existing class. It is the primary tool for reusing your own and standard library classes. Inheritance allows a programmer to derive a new class (called a derived class or a subclass) from another class (called a base class or superclass). A derived class inherits all the data fields and methods (but not constructors) from the base class and can add its own

Example: Calculation Class

The following example illustrates how to extend a class and add a method while retaining the methods in the original class. There are two class files:

- Calculation.java
- MyCalculation.java

`MyCalculation` extends the `Calculation` class and adds one method – multiplication.

To try this out, create two class files, one for `Calculation` and one for `MyCalculation`.

Calculation.java

```
class Calculation{
   int z;

   public void addition(int x, int y){
      z = x+y;
      System.out.println("The sum of the given numbers:"+z);
   }

   public void subtraction(int x,int y){
      z = x-y;
      System.out.println("The difference between the given
numbers:"+z);
   }
}
```

MyCalculation.java

```
public class MyCalculation extends Calculation{

   public void multiplication(int x, int y){
      z = x*y;
      System.out.println("The product of the given numbers:"+z);
   }

   public static void main(String args[]){
      int a = 20, b = 10;
      MyCalculation demo = new MyCalculation();
      demo.addition(a, b);
      demo.subtraction(a, b);
      demo.multiplication(a, b);
   }
}
```

Program Execution

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

Example: Use of Super

The following example illustrates how to utilize the `super` keyword to reference the super class. In Java the `super` keyword can be used to:

- Differentiate the members of the superclass from the subclass if they have the same names.
- To invoke the superclass constructor from the subclass.

If a class is inheriting properties from another class, you can use the `super` keyword as follows:

- `super.variable`
- `super.method();`

This example includes two classes:

- Super_class.java
- Sub_class.java

Super_class.java

```
class Super_class{

   int num = 20;

   //display method of superclass
   public void display(){
      System.out.println("This is the display method of
superclass");
   }

}
```

Sub_class.java

```
public class Sub_class extends Super_class {

   int num = 10;

   //display method of sub class
   public void display(){
      System.out.println("This is the display method of
subclass");
   }

   public void my_method(){

      //Instantiating subclass
      Sub_class sub = new Sub_class();

      //Invoking the display() method of sub class
      sub.display();

      //Invoking the display() method of superclass
      super.display();

      //printing the value of variable num of subclass
      System.out.println("value of the variable named num in sub
class:"+                    sub.num);

      //printing the value of variable num of superclass
      System.out.println("value of the variable named num in
super class:"+                     super.num);
   }

   public static void main(String args[]){
      Sub_class obj = new Sub_class();
      obj.my_method();

   }
}
```

Program Execution

```
This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20
```

Example: Calling a Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the super class. But if you want to call a parameterized constructor (one that takes arguments) of the super class, you need to use the `super` keyword. An example of this is given here.

This example includes two classes:

- Superclass.java
- Subclass.java

Superclass.java

```
class Superclass{

   int age;

   Superclass(int age){
      this.age = age;
   }

   public void getAge(){
      System.out.println("The value of age in super class is: "
+age);
   }
}
```

Subclass.java

```
public class Subclass extends Superclass {

   Subclass(int age){
      super(age);
   }

   public static void main(String args[]){
      Subclass s = new Subclass(24);
      s.getAge();
   }

}
```

Program Execution

```
The value of age in super class is: 24
```

Types of Inheritance

For programming languages that allow inheritance, it is important to understand the various types of inheritance that can occur. These types are outlined in this chart shown below: