Day 3
Inheritance and Polymorphism: Sameness and Differences

Inheritance

Inheritance makes it possible to build new classes from existing classes thus facilitating the reuse of methods and data from one class in another. Moreover, inheritance allows data of one type to be treated as data of a more general type.

Example:

```
public  class Cat
{
        protected  int weight;          // notice the keyword "protected"

        public Cat()
        {
                weight = 10;
        }
        public Cat(int weight)
        {
                this.weight = weight;
        }
        public void setWeight(int w)
        {
                weight = w;
        }
        public int getWeight()
        {
                return weight;
        }
        public void eat()
        {
                System.out.println("Slurp, slurp");
        }
        public  int mealsPerDay()
        {
                return  2 + weight/50;
        }
}
```

// here is the inheritance part!

```
 public class Leopard extends Cat      // "extends" indicates inheritance
{
        protected int numSpots;
        public Leopard()
        {
                weight = 100;
                numSpots =0;  // a poor excuse for a leopard!!
        }
        public Leopard(int weight, int numSpots)
        {
                super(weight); // a call to the one argument constructor of Animal
                this.numSpots = numSpots;
        }
        public void setNumSpots(int n)
        {
                numSpots = n;
        }
        public int getNumSpots()
        {
                return numSpots;
        }
        public void eat()  //overriding the eat method of Animal
        {
                System.out.println("CRUNCH...CHOMP...CRUNCH...SLURP");
        }
    public int mealsPerDay()   //overriding the method of Animal
        {
                return super.mealsPerDay() * 2;  // note call to parent method
        }
        public void roar()   //a non-inherited method
        {
                System.out.println("GRRRRRRRRRRRRRRRRRRRR");
        }

}
```

The two classes, Cat and Leopard are related through inheritance.

- Use of the keyword **extends** signifies an inheritance relationship: Leopard extends Cat, Leopard inherits from Cat.

- Cat is called the *base class*, *super class*, or *parent class.*

- Leopard is a *derived class*, *sub class* or *child class*.

- The Leopard class inherits all data and methods of the parent base class. However, the Leopard class can also *override* any inherited methods and provide its own implementation. Further, the Leopard class may include new methods and variables that are not part of the base class.

- Constructors are not inherited.

- The derived class can call the constructor of the parent class with the keyword *super* (super() or super(x)). If a derived class calls a super class constructor then this call must be made before any other code is executed in the constructor of the derived class. If an explicit super() call is not made, the default constructor of the parent is automatically invoked. If a superclass defines constructors but not a default constructor, the subclass cannot use the default constructor of the super class because none exists.   It is a good practice to define a default constructor whenever you define any constructor.

- The access modifier **protected** is used in the parent. A protected variable or method in a public class MyClass can be accessed by any subclass of MyClass.

To prevent a class from being extended, use the modifier **final**. A final class cannot be a parent class:
Example: public final class MyClass
The class MyClass cannot be extended and is the parent of no other classes.

A Leopard *is-a* Cat

Inheritance allows the creation of a more specialized class from a parent class. The derived class extends the attributes and/or functionality of the base class. A derived class has everything that the base class has, and more.

The relationship between the base class and a derived class is often called an *is-a* relationship because every derived class *is a* (kind of) superclass. For example, A Leopard *is a* Cat in the sense that a Leopard can do everything that a Cat can do. A Leopard has all the attributes and functionality of a Cat (and more). When deciding whether or not to extend a class, you should determine whether or not an *is-a* relationship exists. If not, inheritance is probably not appropriate.

A Leopard *is-a* Cat and, as such, a Leopard object may be considered a Cat object.
For example, the following assignments are valid:

```
Cat cat  = new Leopard(200, 300);
```
or
```
Cat cat;
Leopard leopard = new Leopard(200, 300);
cat = leopard;
```

This is called *upcasting:* a base-type reference may point to an object of a derived type. Thus any type derived from class Cat (e.g., a Leopard) may be considered to be of type Cat. More specifically,

objects of a derived type may be considered objects of the base type.

This relationship between the base class and its derived classes is the cornerstone of inheritance. As mentioned at the start of this chapter: *inheritance allows data of one type to be treated as data of a more general type.* Yes, it is dandy that you can add new attributes and methods to the Cat class but it is even dandier that an object of type Leopard can be considered of type Cat. With a few more tools, you will see just how powerful this concept really is. You will see that a single sorting or searching method can work with many different types. Because objects of a derived type can be considered objects of a base type, one method, one piece of code, can handle objects of many different types.

Everything Inherits: the Object Class

The package java.lang contains a class Object, which might be considered the mother of all classes.

Every class in Java is a subclass of Object. Every class is derived from Object. Every class extends Object. Math, String, and StringBuffer all extend Object. Cat and Leopard also extend Object. Cat is-a Object; Leopard is-a Object. There is no escape.

Being a descendent of Object does come with familial privileges:

- Every class inherits the operator, instanceof.

- Every class inherits the methods of Object:
  ```
  public boolean equals(Object object)
  public String toString()
  ```

Every class can be considered of type Object, i.e., every class can be upcast to Object.

Inheriting from Object

Casting and the *instanceof* operator:

Java permits type casting among primitive types. For example, the following implicit cast that results in no loss of accuracy is acceptable:

```
int x = 5;
double y;
y = x;          // In this case, y will have the value 5.0.
```

On the other hand, the following cast (which is allowed in C++) is *illegal* in Java:

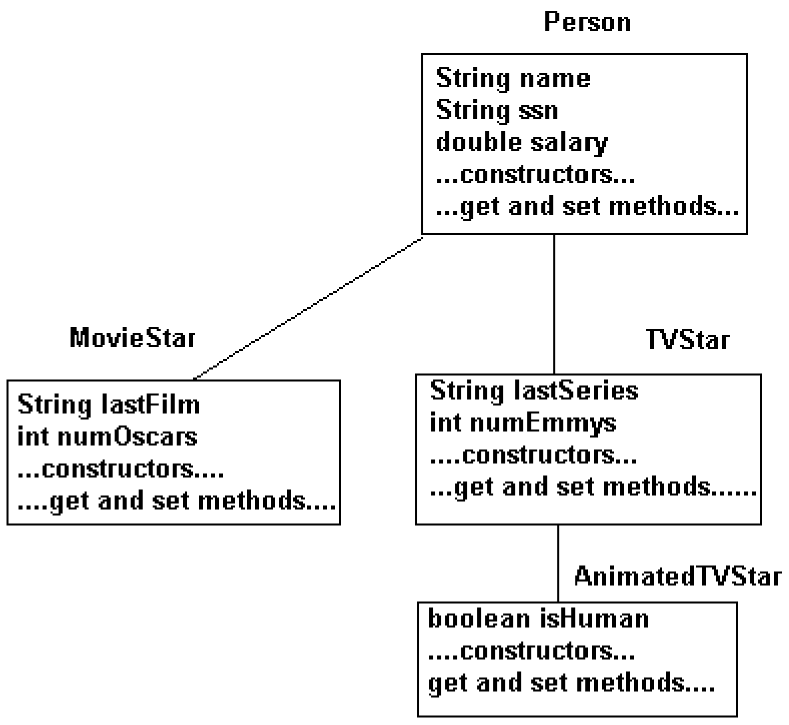
```
double x = 3.75;
int y;
y = x;
```

In C++, y would happily accept the integer value 3 but Java will not tolerate any loss of accuracy unless you force the issue with an explicit type cast:

```
double x = 3.37;
int y;
y = (int)x;
```

Similar rules hold for objects.

Now, consider the following inheritance hierarchy:



As we've already mentioned, upcasting is legal:

```
Person spongeBob = new AnimatedTVStar();
TVStar homerSimpson = new AnimatedTVStar();
```

In the first statement, spongeBob is a Person reference. The assignment

```
spongeBob = new AnimatedTVStar;
```

is legal because *every* Animated TVStar **is-a** Person. The same logic holds true for the second statement.

On the other hand, the following code will cause an error.

```
Person seinfeld = new Person();
TVStar tvStar = seinfeld;
```

Every Person is *not* a TV star and you cannot arbitrarily assign an instance of Person to a TVStar reference.

However, under certain conditions, an explicit *downcast* is permissible:

```
1. Person seinfeld = new TVStar();
2. TVStar x = (TVStar)seinfeld;
3. x.numEmmys();
```

Line1:
seinfeld is a Person reference. Every TVStar *is-a* person. The assignment is legal.

Line 2:
x is a TVStar reference. seinfeld is a Person reference which points to a TVStar. The assignment is legal with an explicit downcast. As a Person reference, seinfeld is unaware of its TVStar status unless explicitly cast to a TVStar.

Line 3:
x is a TVStar reference. numEmmys() is a TVStar method. There is no problem here.

Here is another illustration. Consider the following code fragment:

```
Person people[] = new Person[3];
people[0] = new MovieStar();
people[1] = new TVStar();
people[2] = new AnimatedTVStar( );
```

Each of these assignments is legal because a MovieStar *is-a* Person (really?), a TVStar *is-a* Person (well, almost), and an AnimatedTVStar *is-a* Person (kindda).

On the other hand, the method calls
people[0].getOscars() and
people[2].isHuman()

will each cause an error.

The references people[0] and people[2] know nothing of the methods getOscars() and isHuman(). people[0] and people[2] reference **Person** objects.

Nonetheless, a downcast will produce the desired results:

((**MovieStar**)people[0]).getOscars()
((AnimatedTVStar)people[2]).ishuman()

Java will not allow the cast

```
                ((MovieStar).people[2]).getOscars()
```

since people[2] was originally assigned anAmimatedTVStar object which is not even in the same hierarchy as MovieStar.

The use of the **instanceof** operator helps to avoid casting errors.

The instanceof operator returns true if an object is an instance of a certain class or a subclass derived from that class:

```
                if ( people[2] instanceof AnimatedTVStar)
                        boolean  human = ((AnimatedTVStar)people[2]).isHuman();
                else.......
```

Example:
Notice the use of the instanceof operator in class InstanceofDemo below.

```
public class RectangleClass
{
     private int length;
     private int width;

        public RectangleClass(int x, int y)
        {
                length = x;
                width = y;
        }

        public int area()
        {
                return length*width;
        }
}
```

```
public class CubeClass
{
     private int length;
     private int width;
     private int height;

        public CubeClass(int x, int y, int z)
        {
                length = x;
                width = y;
                height = z;
        }

        public int volume()
        {
                return length*width*height;
        }
}
```

```
public class InstanceofDemo
{
        public static void main( String args[])
        {
                CubeClass x = new CubeClass(3,4,5);
                RectangleClass y = new RectangleClass(3,4);

                junk(x);
                junk (y);
        }
        public static void junk(Object z)  // notice the type Object
        {
                if (z instanceof RectangleClass)
                {
                   RectangleClass x;
                   x = (RectangleClass) z;  // notice the cast here
                   System.out.println(x.area());

                 }
                 else if(z instanceof CubeClass)
                {
                   CubeClass x = (CubeClass) z; // again notice the cast (CubeClass)
                   System.out.println(x.volume());
                }
                 else
                        System.out.println("Unknown object");

         }
}
```

The parameter of method junk() is of type Object. Since every object (lower case “o”) *is-a* Object (upper case “O”), *any* object may be passed to method junk(). junk() uses the instanceof operator to determine whether or not z is a RectangleClass or a CubeClass.

boolean equals(Object object):

The equals() method tests whether or not two objects are equal.
In the Object class equals() is implemented as:

```
        boolean equals(Object x)
        {
                return (this == x);
        }
```

So equals() is equivalent to the relational operator == for the Object class.

Remember, variables x and y are references. So, x==y returns true only if x and y both point to the exact same object. If x and y point to distinct objects with identical data, x == y returns false, as will equals().

Consider the following code fragment:

```
                String s = "doc"; // “doc” is a string literal and an object
                String t = new String("doc");
                System.out.println(s == "doc");
                System.out.println(t == "doc");
```

What is the output?
You would hope that the output is

```
        true
        true
```

but, in fact, it is

```
        true
        false
```

Remember that all strings in Java are genuine objects.
The statement:

```
String s = “doc”;
```

first creates an object for the string literal “doc” and then assigns the address of that object to reference s. The string literal “doc” is an instance of the String class.

The next statement

```
String t = new String(“doc”);
```

creates a second String object and assigns its address to reference t. Thus, s and t are referencing two different String objects, the string literal “doc” and the second String object created by new. Hence, s==”doc” is true but t==”doc” is false.

Fortunately, the String class overrides the equals method inherited from Object. The overridden version of equals() compares *characters* not references.

Using the equals() method of the String class instead of ==, the above fragment is rewritten:

```
String s = "doc";
String t = new String("doc");
System.out.println(s.equals("doc"));
System.out.println(t.equals("doc"));
```

The output is

```
true
true
```

as it should be.

Good Habit I: *To determine whether or not two objects of a class are equal based on the data of the objects, a class should override the equals() method that is inherited from Object.*

Assume that two Cube objects are equal if they have the same volume. In the following example, class Cube overrides the equals() method. The new equals() method is based on volume.

```
public class Cube
{
    private int length;
    private int width;
    private int height;

        public Cube(int x, int y, int z)
        {
                length = x;
                width = y;
                height = z;
        }

        public int volume()
        {
                return length*width*height;
        }

        public boolean equals(Object x)
        {   // notice the downcast, an Object does not have length, width and height
            //but a Cube does.
           return  length*width* height == ((Cube)x).length* ((Cube)x).width* ((Cube)x).height;

        }


        public static void main(String args[])
        {
                Cube a = new Cube(2,3,4);
                Cube b = new Cube(2,3,4);
                System.out.println(a.equals(b)); // uses overridden equals
               System.out.println( a == b);  // compares references
        }
}

The output is
            true
            false
```

You might be wondering:

Why not just write an equals method for Cube?
boolean equals(Cube x)
No downcast would be necessary. It is simpler.

Yes, such a version of equals would work. Yes, it appears simpler. However, you will shortly see the real benefit in overriding the equals method inherited from Object. Just wait a bit more.

String toString() : returns a string representation of the calling object.

Like equals(), every object inherits toString() from Object. However, the inherited version of toString() is not particularly useful. As inherited from Object, toString() returns the class name of the calling object along with a system number.

```
public static void main(String args[])
{
        Cube r = new Cube(2,3,4);
        System.out.println(r.toString());
}
```

The output of this is:

```
                        Cube@310d42
```

Good Habit II: *Overriding toString() makes good sense. A class should override the toString() method by including all the relevant information about an object. Often such information is helpful when debugging.*

Example:
For class Cube, you might override toString() as follows:

```
 public String toString()
{
        String s = "length = "+length+" width = "+width+" height = "+height;
        return s;
}
```

Now, when toString() is invoked by a Cube object the result is a bit more meaningful than "Cube@310d42."

```
        public static void main(String args[])
        {
                        Cube r = new Cube(2,3,4);
                        System.out.println(r.toString());  ←————————
        }

        Output :
                length = 2 width = 3 height = 4
```

One final note:
The toString() method is automatically called when an object is passed to println. Consequently

```
                System.out.println(r.toString());  and
                System.out.println(r));
```

produce the same output

Inheritance via Abstract classes

Consider the following three classes that encapsulate three geometrical shapes. Notice that each class implements toString() and equals(), both inherited from Object.

```
public class Square
{
    private int rows;
    private char character;

//constructors
    public Square()
   {

        rows = 0;
        char character = ' ';
    }

public Square(int x, char ch)
{
    base =height = x;
    character = ch;
}


public int getRows()
{
    return rows;
}

public char getCharacter()
{
    return character;
}

public void setRows(int y)
{
    rows = y;
}

public void setCharacter(char ch)
{
    character = ch;
}

public boolean equals(Object x)
{
    return  numChars() ==
     ((Square)x).numChars();
}

public String toString()
{
  String s = “rows = “ +rows;
   return s;
}

public int  numChars()
{
    return rows * rows;
}
```

```
public class RightTriangle
{
   private int rows;
   private char character;


public RightTriangle()
   {

         rows = 0;
         char character = ' ';
    }

public RightTriangle(int x, char ch)
 {
    rows = x;
    character = ch;
}


public int getRows ()
{
    return rows;
}

public char getCharacter()
{
    return character;
}

public void setRows(int y)
{
    rows = y;
}

public void setCharacter(char ch)
{
    character = ch;
}

public boolean equals(Object x)
{
    return  numChars() ==
     ((RightTriangle)x).numChars();
}

public String toString()
{
  String s = "rows = " +rows;
   return s;
}

public int numChars()
{
              return ((rows)*(rows+1))/2;
}
```

```
public class Isosceles
{
   private int rows;
   private char character;


public Isosceles ()
   {

         rows = 0;
         char character = ' ';
    }

public Isosceles (int x, char ch)
{
    rows = x;
    character = ch;
}


public int getRows()
{
    return rows;
}

public char getCharacter()
{
    return character;
}

public void setRows(int y)
{
    rows = y;
}

public void setCharacter(char ch)
{
    character = ch;
}

public boolean equals(Object x)
{
    return  numChars() ==
     ((Isosceles)x).numChars();
}

public String toString()
{
  String s = "rows = " +rows;
   return s;
}

public int  numChars()
{
              return ((rows)*(rows+1))/2;
}
public void   draw(int x, int y)
{
   for ( int i = 1; i <= y; i++)
          System.out.println();
   for (int len = 1; len<= rows; len++)
   {
      for (int i = 1; i <= x; i++)
          System.out.print(' ');
        for (int j = 1; j <= rows; j++)
           System.out.print(character);
         System.out.println();
    }
}
}
```

```
public void   draw(int x, int y)
{
   for ( int i = 1; i <= y; i++)
        System.out.println();
   for (int len = 1; len<= rows;
len++)
   {
     for (int i = 1; i <= x; i++)
      System.out.print(' ');
      for (int j = 1; j <= len; j++)
      System.out.print(character);

         System.out.println();
    }
}
}
```

```
public void   draw(int x, int y)
{
   for ( int i = 1; i <= y; i++)
        System.out.println();
   for(int j=0; j<rows; j++)
   {
      for(int i=0; i < rows-j+x; i++)
      System.out.print(" ");
      for(int i =0; i<j+1; i++)
      System.out.print(character
                              +" " );
      System.out.println();
    }
  }
}
```

Note: The numChars() method returns the number of characters used to draw a figure.
For example if rows = 5, the three figures are drawn as:

```
*****          *                         *
*****          **                       * *
*****          ***                     * * *
*****          ****                   * * * *
*****          *****                 * * * * *
Square         RightTriangle           Isosceles
```

using 25, 15, and 15 characters respectively. Each triangle, though drawn differently, has 1+2+3+4+5 characters.

There is much the same about the three classes. In fact, they are more similar than different. If several classes have data and methods in common, you may be able to *factor out* the commonality of the classes into one super class.

Example

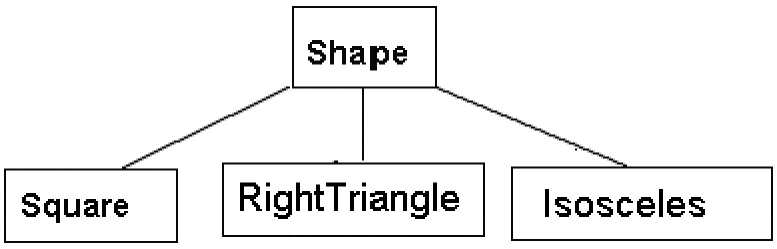
```
public class Shape  // has methods common to Square, RightTriangle, and Isosceles
{
        protected int rows;
        protected char character;

        public Shape()
        {
                rows  = 0;
                char character = ' ';
        }

        public Shape(int x, char ch)
        {
                rows = x;
                character = ch;
        }
        public int getRows()
        {
                return rows;
        }

        public char getCharacter()
        {
                return character;
        }

        public void setRows(int y)
        {
                rows = y;
        }

        public void setCharacter(char ch)
        {
                character = ch;
        }

        public boolean equals(Object x)
        {
          return  numChars() ==
              ((Shape)x).numChars();
        }


        public String toString()
        {
                String s = “rows = “ +rows;
                return s;
        }

        public int numChars()
        {
                System.out.println("Not applicable");
                return -1;
        }

        public void   draw(int x, int y)
        {
                System.out.println("Draw method not applicable.");
        }

}
```

Square, RightTriangle and Isosceles can now be redefined to extend Shape. Shape is the parent. Shape is the base class.

```
class Square extends Shape
{
public Square()
{
       super();
}

public Square(int x, char ch)
{
            super(x,ch);

}


public void draw(int x, int y)
{

    for ( int i = 1; i <= y; i++)
            System.out.println();
    for (int len = 1; len<= rows;
len++)
    {
        for (int i = 1; i <= x; i++)
            System.out.print(' ');
       for (int j = 1; j <= rows; j++)
            System.out.print(character);
        System.out.println();
     }
}
public boolean equals(Object x)
{
    return  numChars() ==
     ((Square)x).numChars();
}

public int numChars()
{
      return rows * rows;
}
}
```

```
class RightTriangle extends Shape
{
public RightTriangle()
{
       super();
}

public RightTriangle(int x, char ch)
{
            super(x,ch);

}


public void draw(int x, int y)
{

   for ( int i = 1; i <= y; i++)
            System.out.println();
    for (int len = 1; len<= rows;
len++)
    {
        for (int i = 1; i <= x; i++)
            System.out.print(' ');
        for (int j = 1; j <= len; j++)
            System.out.print(character);
         System.out.println();
     }
}

public boolean equals(Object x)
{
    return  numChars() ==
     ((RightTriangle)x).numChars();
}

public int numChars()
{
      return ((rows)*(rows+1))/2;
}
}
```

```
class Isosceles extends Shape
{
public Isosceles ()
{
       super();
}

public Isosceles (int x, char ch)
{
            super(x,ch);

}

public void   draw(int x, int y)
{
    for ( int i = 1; i <= y; i++)
    System.out.println();
    for(int j=0; j<rows; j++)
    {
       for(int i=0; i < rows-j+x; i++)
       System.out.print(" ");
       for(int i =0; i<j+1; i++)
       System.out.print(character
                                       +" " );
       System.out.println();
     }
  }


public boolean equals(Object x)
{
    return  numChars() ==
     ((Isosceles)x).numChars();
}

public int numChars()
{
        return ((rows)*(rows+1))/2;
}
}
```



Of course, the base class *Shape* is not a class that one would ordinarily instantiate. What does a shape object look like?   Yes, a Shape object can be created, but such an object probably would not make much sense. Further a call to the draw() or numChars() methods of Shape would do no more than produce a message. In fact, you might say that Shape is a somewhat theoretical, intangible, abstract, and even ethereal class.

In Java, the notion of an *abstract* class is very precise:

> An abstract class is a class that cannot be instantiated.

To be of *any* use, an abstract class must be extended. The following example re-defines Shape as an **abstract** class.

Example

```
public abstract class Shape  // notice the keyword abstract
{
        protected int rows;
        protected char character;

        public Shape()
        {
                rows = 0;
                char character = ' ';
        }

        public Shape(int x, char ch)
        {
                rows = x;
                character = ch;
        }

        public int getRows()
        {
                return rows;
        }

        public char getCharacter()
        {
                return character;
        }

        public void setRows(int y)
        {
                rows = y;
        }

        public void setCharacter(char ch)
        {
                character = ch;
        }
        public String toString()
        {
                String s = “rows = “ +rows;
                return s;
        }
        public abstract boolean equals(Object x);
        public abstract int numChars();
        public abstract void   draw(int x, int y);
}
```

Shape contains all the variables and methods common to the Square, RightTriangle and Isosceles classes. Because Shape is abstract, however, Shape cannot be instantiated, i.e., there can be no Shape objects. Indeed, methods numChars() and draw() and equals() have no implementations whatsoever.

In general, an abstract class has the following properties:

- The word *abstract* denotes an abstract class.
- An abstract class is a template that can be inherited by subclasses.
- An abstract class cannot be instantiated. You cannot create an object of an abstract class.
- An abstract class *may* contain abstract methods**.** If a class is declared abstract and does not have any abstract methods, the class cannot be instantiated.
- If an abstract class contains abstract methods, those methods *must* be overridden in the subclass. If this is not done, then the subclass will also have to be declared abstract.
- All abstract classes and methods are public.

Although Shape cannot be instantiated, Shape can still serve as a base class with the following proviso:

> Any class that extends shape *must* implement the draw(), equals(), and numChars() methods;   otherwise the derived class must also be abstract.

That’s the contract – any class that extends Shape must implement these methods (or else remain in abstract-land). Thus, any non-abstract sub-class of Shape is guaranteed to have a draw() , equals() and numChars() method – the contract must be fulfilled.
Furthermore, Square, RightTriangle, and Isosceles can all be upcast to Shape.

Interfaces

We have used the term *interface* in conjunction with everything from the buttons on a TV to the public methods provided by a class. In Java, the term *interface* has a very specific meaning:

An *interface* is a named collection of static constants and abstract methods. An interface specifies certain actions or behavior of a class but not implementations.

Example:

The following interface, ThreeDShape, consists of one static constant and three abstract methods.

```
public interface ThreeDShape
{
        public static final double pi = 3.14159;
        public abstract double area();
        public abstract double volume();
}
```

(The modifier *abstract* is optional, since by definition of an interface, all methods are abstract.)

Notice that unlike a class:

- All methods are public.
- All methods are abstract -- no implementations at all.
- There are no instance variables.

Like an abstract class, an interface cannot be instantiated. In fact, a class cannot even *extend* an interface. Instead, a class *implements* an interface.

Example:
The following three classes each implement the ThreeDShape interface.

```
public class Cube implements
                    ThreeDShape
{
   private double length, width,
                  height;
   public Cube()
   {
         length = 1;
         width = 1;
        height = 1;

   }
   public Cube(double l, double w,
                  double h)
   {
         length = l;
         width = w;
          height = h;
   }
   public double area()
   {
         return 2*length*width+
          2*length*height+
          2*width*height;
   }
  public double volume()
  {
    return  length*width*height;
  }

}
```

```
public class Sphere   implements
ThreeDShape
{
   private double radius;

  public Sphere()
   {
          radius = 1;

   }


 public Sphere(double r)
   {
          radius = r;
   }



public double area()
   {
      return
      4*pi*radius*radius;
   }

public double volume()
  {
       return
      (4.0/3.0)*
      pi*radius*radius*radius;
  }
}
```

```
public class Cylinder implements
ThreeDShape
{
   private double radius,height;

  public Cylinder()
   {
          radius = 1;
         height = 1;

   }

 public Cylinder(double r,
                  double h)
   {
          radius = r;
         height = h;
   }

 public double area()
   {
      return  2*pi*radius*height+
      2*pi*radius*radius;
   }

public double volume()
  {
      return
      pi*radius*radius*height;
  }
}
```

An interface is a contract. An interface specifies a set of responsibilities, actions, or behaviors for any class that implements it. Any class that implements an interface must implement *all* the methods of the interface. Because Cube, Sphere and Cylinder all implement ThreeDShape, all three classes, by contract, *must* implement the volume() and area() methods as declared in the interface. Moreover, since Cube implements ThreeDShape, any client of Cube is guaranteed area() and volume() methods.

But isn't this idea of a *contract* true of an abstract class? Doesn't every (non abstract) class that extends an abstract class have an obligation to implement the abstract methods? Why confuse things with interfaces? Why not simply define an abstract class where every method is abstract? Wouldn't such a class accomplish the same thing as an interface? The answer to the last question is yes and no.

Some object oriented languages like C++ allow *multiple inheritance.* A subclass can inherit from multiple super classes. The unrestricted use of multiple inheritance is a controversial feature with many complexities and pitfalls. Many programmers do not use the feature on principle. Nonetheless, there are many advantages and conveniences of multiple inheritance. Java avoids such complexities, but does not throw the baby out with the bathwater. Java does not allow multiple inheritance - a subclass cannot inherit from two different base classes. On the other hand, a class may implement any number of interfaces. Thus, a class may extend one class as well as implement an interface or two. In fact, a derived class can be upcast to any one of its interfaces. Therefore, one difference between interfaces and abstract classes, is that interfaces allows the Java Programmer some of the flexibility of multiple inheritance, without the associated pitfalls.

Interface Example 1:

In a previous example, we designed a hierarchy of classes with methods used for drawing various squares and triangles. The following version of Shape does not contain numChar() and draw().

```
public abstract class Shape  // cannot be instantiated
{
        protected int rows;
        protected char character;

        public Shape()
        {
                rows = 0;
                char character = ' ';
        }

        public Shape(int x, char ch)
        {
                rows = x;
                character = ch;
        }

        public int getRows()
        {
                return rows;
        }

        public char getCharacter()
        {
                return character;
        }
        public void setRows(int y)
        {
                rows = y;
        }

        public void setCharacter(char ch)
        {
                character = ch;
        }

        abstract public boolean equals(Object x);

        public String toString()
        {
                String s = “rows = “ +rows;
                return s;
        }

}
```

Below is an interface, Drawable, specifying a contract for any implementations.

```
public interface Drawable
{
        public int numChars();
        public void   draw(int x, int y);
}
```

Finally, we have the three concrete classes: Square, RightTriangle, and Isosceles. Each extends Shape and implements Drawable. Each can be considered of type Shape as well as type Drawable.

```
class Square extends Shape
             implements Drawable
{
public Square()
{
      super();
}

public Square(int x, char ch)
{
          super(x,ch);
}
public boolean equals(Object x)
{
   return  numChars() ==
((Square)x).numChars();
}

public void draw(int x, int y)
{
   for ( int i = 1; i <= y; i++)
          System.out.println();
   for (int len = 1; len<= rows;
len++)
   {
       for (int i = 1; i <= x; i++)
          System.out.print(' ');
      for (int j = 1; j <= rows; j++)
          System.out.print(character);
       System.out.println();
    }
}

public int numChars()
{
     return rows * rows;
}
}
```

```
class RightTriangle extends Shape
                implements Drawable
{
public RightTriangle()
{
      super();
}

public RightTriangle(int x, char ch)
{
          super(x,ch);
}
public boolean equals(Object x)
{
   return  numChars() ==
((RightTriangle)x).numChars();
}

public void draw(int x, int y)
{
  for ( int i = 1; i <= y; i++)
          System.out.println();
   for (int len = 1; len<= rows;
len++)
   {
      for (int i = 1; i <= x; i++)
          System.out.print(' ');
      for (int j = 1; j <= len; j++)
          System.out.print(character);
       System.out.println();
    }
}

public int numChars()
{
     return ((rows)*(rows+1))/2;
}
}
```

```
class Isosceles extends Shape
            implements Drawable
{
public Isosceles ()
{
      super();
}

public Isosceles (int x, char ch)
{
          super(x,ch);
}
public boolean equals(Object x)
{
           return  numChars() ==
((Isosceles)x).numChars();
}

public void   draw(int x, int y)
{
   for ( int i = 1; i <= y; i++)
   System.out.println();
   for(int j=0; j<rows; j++)
   {
      for(int i=0; i < rows-j+x; i++)
      System.out.print(" ");
      for(int i =0; i<j+1; i++)
      System.out.print(character
                             +" " );
      System.out.println();
    }
  }

public int numChars()
{
       return ((rows)*(rows+1))/2;
}
}
```

The Comparable Interface

So far, class Cube has overridden both the equals() and toString() methods inherited from Object. The equals() method allows us to determine whether or not two Cubes are equal. Now, by implementing the Comparable interface, any two Cubes can be compared.

The Comparable interface is an interface with just one (abstract) method, compareTo():

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

Notice that compareTo() returns an integer. A class which implements the Comparable interface *usually* implements compareTo so that

```
                        a.CompareTo(b)  =  –1 if  a is less than b
                        a.CompareTo(b)  =  0 if  a equals b
                        a.CompareTo(b)  =  1 if  a is greater than b
```

Interface Example II:

```
public class Cube implements Comparable
{
          private int length;
          private int width;
          private int height;

        public Cube(int x, int y, int z)
        {
                length = x;
                width = y;
                height = z;
        }

        public int volume()
        {
                return length*width*height;
        }

        public boolean equals(Cube x)
        {
            return  length*width*height == x.length*x.width*x.height;
        }

        public String toString()
        {
                String s = "length = "+length+" width = "+width+" height = "+height;
                return s;
        }

        public int compareTo(Object o)  // compare based on volume
        {
                // because the parameter o is of type Object a downcast to Cube is essential
                if (volume() < ((Cube)o).volume())
                        return -1;
                else if (volume() > ((Cube)o).volume())
                        return 1;
                else
                        return 0;
        }
}
```

The Flexibility of Interfaces

Implementation of the Comparable interface highlights another more subtle distinction between interfaces and abstract classes. A Cube class can implement Comparable - so can a Car class, a Person class, or a Vampire class. However, the classes that implement Comparable are not necessarily related. On the other hand, because an abstract class contains some implementations, derived classes all share these implementations and are logically coupled.

We return to the abstract Shape class. In the world of Shape objects, comparisons are based on numChars. So, although it may be illogical to compare apples and oranges, comparing triangles and squares makes perfect sense.

Example:

```
 public abstract class Shape  implements Comparable
{
    protected int rows;
    protected char character;

    public Shape()
    {
            rows = 0;
            char character = ' ';
    }

    public Shape(int x, char ch)
    {
            rows = x;
            character = ch;
    }
    public int getRows()
    {
            return rows;
    }
    public char getCharacter()
    {
            return character;
    }
    public void setRowst(int y)
    {
            rows = y;
    }
    public void setCharacter(char ch)
    {
            character = ch;
    }
    public abstract int numChars();
    public abstract void   draw(int x, int y);
    public boolean equals(Object x)
    {
             return  numChars() == ((Shape)x).numChars();
    }
```

```
        public String toString()
        {
                String s = “rows = “ +rows;
                return s;
        }

        public int compareTo(Object o)
        {
             if (numChars() < ((Shape)o).numChars())
                        return -1;
                else if (numChars() > ((Shape)o).numChars())
                        return 1;
                else
                        return 0;
         }
}
```

The next example demonstrates the power of inheritance with a general routine capable of sorting an array of objects of any class C, *provided C implements the Comparable interface.*

Example:

```
 public class Sort // selection sort
{
  public static void sort(Comparable[] x)  // any type can be upcast to Object
  {
           Comparable max;
            int maxIndex;

           for (int i=x.length-1; i>=1; i--)
           {
             // Find the maximum in the x[0..i]
             max = x[i];
             maxIndex = i;

             for (int j=i-1; j>=0; j--)
             {
                 //Since max is of type Object, max must be downcast
               if (max.compareTo(x[j]) < 0)
               {
                 max = x[j];
                 maxIndex = j;
               }
             }
             if (maxIndex != i)
             {
                 x[maxIndex] = x[i];
                 x[i] = max;
             }
           }
   }

}
```

The following fragment illustrates the generic sort method:

```
int length, width, height;
Cube x[] = new Cube[10];
 System.out.println("enter the length, width, and height for 10 cubes");
 for (int i = 0; i <10; i++)
 {
             length = MyInput.readInt();
              width = MyInput.readInt();
              height = MyInput.readInt();
              x[i] = new Cube(length,width,height);
 }
 Sort.sort(x);
 System.out.println("The cubes  in order of magnitude are: ");
 for(int i = 0; i < 10; i++)
 System.out.println(x[i].volume());
```

If inheritance merely provided new functionality for existing classes, it would still be a useful technique. However, the real muscle lies in the fact that a derived type object can be considered an object of a base type, i.e., the ability to upcast.

Composition and the *has-a* relationship

Inheritance is characterized by an *is-a* relationship:

a Square is-a Shape,
a RightTriangle is-a Shape,
a MovieStar is-a Person,
a Dog is-an Animal,
a Terrier is-a Dog.

Often times, however, classes are related but not via an is-a relationship, and upcasting is not of any apparent value. Consider for example the two (partial) classes Person and BankAccount:

public class Person
{
   private String name:
   private String address;
   //etc.
}
	public Class BankAccount
	{
	     private String accountNumber;
	     private double balance;
	     ....
	     public double balance()
	     //etc.
	}

Suppose that every Person possesses a BankAccount. Certainly it is *possible* to derive BankAccount from Person or Person from BankAccount, but the relationship is not natural. A person is-*not* a BankAccount and a BankAccount is-*not* a Person. Also, there is no apparent or logical reason to consider a Person a type of BankAccount or vice versa. Inheritance is not a good fit.

We have already seen that one object may contain objects of another class. Indeed, string object have been included in many of our previous classes. Thus, a BankAccount object might be an instance variable of the Person class. We might say that a Person *has-a* BankAccount.

```
public Person
{
        private String name:
        private String address;
        private BankAccount account;
        // etc.
}
```

The relationship between the Person and the BankAccount classes is an example of *composition* -- a relationship where one object is composed of other objects. A class in which some instance variables are objects is often called an *aggregate* class.   As an **is-a** relationship indicates inheritance, a **has-a** relationship signals composition. Inheritance implies an extension of functionality and the ability to upcast; composition indicates ownership. The two should not be confused.

Polymorphism

Polymorphism literally means "many shapes" or "many forms." Method overloading is one form of polymorphism: several methods with the same name may behave differently.

Square	Rectangle	Cube

int area(int x)
{
        return x*x;
}
	int area(int x, int y)
	{
	  return x*y;
	}
		int area(int x, int y, int z)
		{
		     return 2*x*y + 2*x*z + 2*y*z;
		}

The area method has many forms, well at least three.

Inheritance provides another form of polymorphism: an object of a derived type can also be considered an object of a base type:

```
        Person joe;
        joe = new MovieStar();
        joe = new TVStar();
        joe = new AnimatedTVStar();
```

The above code emphasizes that a MovieStar is-a Person, a TVStar is-a Person, and an AnimatedTVStar is-a Person. That is, a Movie Star object is upcast to a Person type as is a TVStar and an AnimatedTVStar. Since joe is of type person, the reference joe is *polymorphic*, i.e., joe has "many forms."   Upcasting is a form of polymorphism.

While inheritance exploits the sameness of types in a hierarchy, a third form of polymorphism, provided via *dynamic or late binding,* emphasizes the behavioral differences among types in the same hierarchy.

This third form of polymorphism is illustrated in the following example.

Example:

Consider the following test class that utilizes the Shape hierarchy:

```
public class TestDrawShape
{
        public static void drawShape(Shape s)
        {
                int xcoord, ycoord;
                System.out.print( "x coordinate: ");
                xcoord = MyInput.readInt();
                System.out.print( "y coordinate: ");
                ycoord = MyInput.readInt();

                s.draw(xcoord,ycoord);
        }

        public static void main(String args[])
        {
           Shape shape = null;
           int rows;
           int shapeNum; //id for each shape
           char ch;

                System.out.println("Enter 1: Square, 2: Right Triangle, 3 : Isosceles Triangle");
                shapeNum = MyInput.readInt();
                System.out.print( "Rows: ");
                rows = MyInput.readInt();
                System.out.print("Character: ");
                ch = MyInput.readChar();

                switch (shapeNum)
                {
                        case  1 : shape = new Square(rows,ch);
                                        break;
                        case  2 : shape = new RightTriangle(rows,ch);
                                     break;
                        case  3 : shape = new Isosceles(rows,ch);
                                  break;
                 }

                System.out.println(shape);
                TestDrawShape.drawShape(shape);

         }
}
```

Method drawShape() takes a **Shape** parameter. However, because Square *is-a* Shape, RightTriangle *is-a* Shape and Isosceles *is-a* Shape, the parameter *s* of the method drawShape(Shape *s*) can refer to any type of Shape, i.e., upcasting is applicable.

Method drawShape() calls draw():

```
s.draw(xcoord,ycoord)
```

Which draw method is called? The choice of the draw method (there are three) is determined not when the program is compiled but at runtime. At run time Java determines which form of the draw method is applicable.

> **The Java Virtual Machine picks the method as determined by the type of the object which was created by the *new* operator.**

There is no way the *compiler* could determine which shape must be drawn. In the code, the method call is simply s.draw(x,y) and s is of type *Shape*. Regardless of the particular type of object passed to drawShape the method s.draw(x,y) works. The draw function is polymorphic i.e. it has many forms.

Without polymorphism a sequence of if-else statements would be necessary to draw the correct figure:

```
if (s instanceof Square)
        ((Square)s).draw(xcoord, ycoord);
else if (s instanceof RightTriangle)
        ((RightTriangle)s).draw(xcoord, ycoord);
else if (s instanceof Isosceles)
        ((Isosceles)s).draw(xcoord, ycoord);
```

Postponing a choice until runtime is called *late binding* or *dynamic binding*.

In designing a programming language, there are many features in which a choice of binding time must be made. For example, if *type* binding is early then the compiler will know that x==y is an error if x is double and y is an integer. If type binding is late, then x and y can change types dynamically during the running of the program, and we would never know whether x==y was an error until we ran the program. Early versus late (or static versus dynamic) binding is a design choice that affects the whole philosophy of a programming language. In general, dynamic or late binding offers flexibility to the programmer, at the cost of efficiency and complexity.

Now, at the risk of gross simplification, let's see how the draw function is chosen.
Notice that the variable shape is declared of type Shape:

```
Shape shape
```

Shape is often called the *apparent type* of shape.

On the other hand, the *real type* of variable shape is the type of the object that was created by the call to *new.* Thus the real type of Shape is either Square, RightTriangle, or Isosceles, depending on user input.

Parameter *s* in method drawShape has apparent type Shape and real type of either Square, RightTriangle or Isosceles. Let's arbitrarily decide that the real type of *s* is RightTriangle.

When the method draw is invoked, Java begins with the real type (RightTriangle). If RightTriangle has a draw method then that method is called. If no draw method is defined in RightTriangle, then the parent of RightTriangle is searched, etc., all the way up the hierarchy.

If instead of draw(), suppose that the getRows() method had been called

```
s.getRows()
```

Again, Java starts searching the RightTriangle class. Since RightTriangle does not implement a getRows() method, Java continues the search in the parent class (Shape) where such a method does exist.

But wait! Polymorphism gets even better. Consider a subclass of RightTriangle, say InvertedRightTriangle, which draws a triangle “up-side-down:”

```
* * * * *
* * * *
* * *
* *
*
```

InvertedRightTriangle necessarily overrides draw() and implements its own constructors:

```
public class InvertedRightTriangle  extends RightTriangle
{
        public InvertedRightTriangle ()
        {
                super();
        }

        public InvertedRightTriangle (int x, char ch)
        {
                super(x,ch);
        }
        public void  draw(int x, int y)
        {
                for ( int i = 1; i <= y; i++)
                                System.out.println();
                for (int len = rows; len>= 1; len--)
                {
                for (int i = 1; i <= x; i++)
                        System.out.print(' ');

                for (int j = 1; j <= len; j++)
                        System.out.print(character);

                System.out.println();
                }
         }
  }
```

The hierarchy has been easily extended, and amazingly, the *only* necessary change occurs in the test program (below in **bold**). Just two lines!

```
public class TestDrawShape2
{
        public static void drawShape(Shape s)
        {
                int xcoord, ycoord;
                System.out.print( "x coordinate: ");
                xcoord = MyInput.readInt();
                System.out.print( "y coordinate: ");
                ycoord = MyInput.readInt();

                s.draw(xcoord,ycoord);
        }

    public static void main(String args[])
    {
        Shape shape = null;
        int rows,shapeNum;
        char ch;

        System.out.println("Enter 1 :Square, 2: RightTriangle, 3: Isosceles, 4 Inverted Triangle ");
        shapeNum = MyInput.readInt();
        System.out.print( "Rows: ");
        rows = MyInput.readInt();
        System.out.print("Character: ");
        ch = MyInput.readChar();

        switch (shapeNum)
        {
                case  1 : shape = new Square(rows,ch);
                                    break;
                case  2 : shape = new RightTriangle(rows,ch);
                                  break;
                case  3 : shape = new Isosceles(rows,ch);
                                break;
                case  4 : shape = new InvertedRightTriangle(rows,ch);
                                 break;
         }

         System.out.println(shape);
         TestDrawShape2.drawShape(shape);

     }
}
```

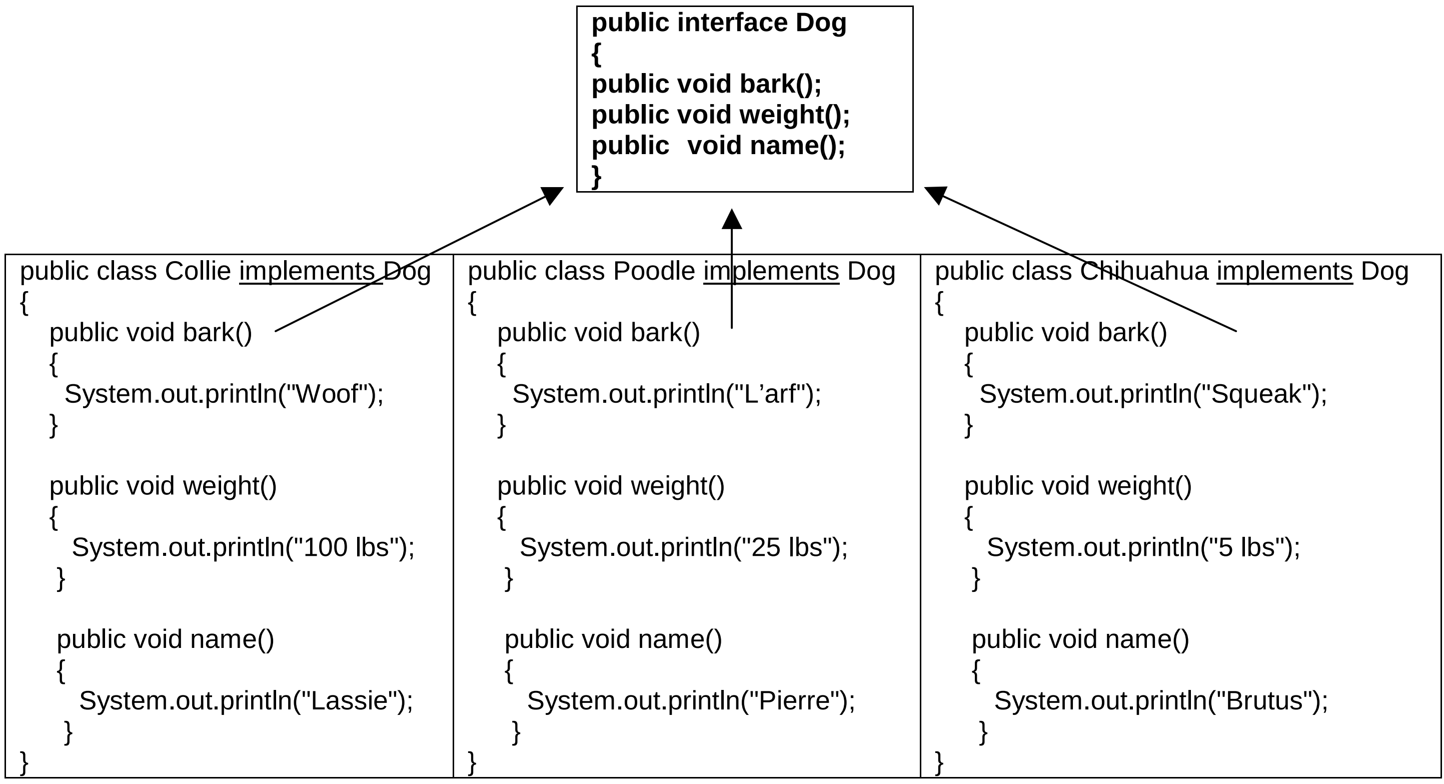
Nothing in the Shape hierarchy needed alteration. In fact the old classes (Square, RightTriangle, Isosceles) do not even have to be recompiled**.**

Polymorphism Makes Programs Extensible.

Without polymorphism, the demo program would ramble on with a bunch of switch or if-else statements. Adding a new class would mean changing existing classes. Moreover, everything would need to be recompiled. We had to do none of this; we just had to write a new class and voilà, it all works -- plug and play. Moreover, polymorphism facilitates code reuse. We can add as many shapes as we'd like and no code in our current system will need to be changed.

Example:

Consider the following hierarchy:



```
public interface Dog
{
public void bark();
public void weight();
public  void name();
}
```

```
public class Collie implements Dog
{
   public void bark()
   {
     System.out.println("Woof");
   }

   public void weight()
   {
      System.out.println("100 lbs");
    }

    public void name()
    {
       System.out.println("Lassie");
     }
}
```

```
public class Poodle implements Dog
{
   public void bark()
   {
     System.out.println("L'arf");
   }

   public void weight()
   {
      System.out.println("25 lbs");
    }

    public void name()
    {
       System.out.println("Pierre");
     }
}
```

```
public class Chihuahua implements Dog
{
   public void bark()
   {
     System.out.println("Squeak");
   }

   public void weight()
   {
      System.out.println("5 lbs");
    }

    public void name()
    {
       System.out.println("Brutus");
     }
}
```

Below is a public class with two methods. The first method greeting() takes one parameter of type Dog which means the reference passed may refer to any subtype. The second method allDogs() receives an array of type Dog.

```
public class Doggie
{
   public static void greeting(Dog d)  // note d is Dog
   {

                  System.out.println(); //blank line
                  System.out.print("My name is ");
                  d.name();                        // which name()????????????????????
                  System.out.print("I weigh ");
                  d.weight();                      //which weight??????????????????????
                  d.bark();                        //which bark()?????????????????????
                  System.out.println();
    }
    public static void allDogs(Dog[] dogList) // note type Dog
    {
          for (int i = 0; i < dogList.length; i++)
                  greeting(dogList[i]);
    }
}
```

Below is a simple main() method which utilizes class Doggie:

```
public static void main(String args[])
{
            int dogNum, count;
            Dog[]  dogList;

             System.out.println("How many dogs?");
             count = MyInput.readInt();
             dogList = new Dog[count];

             System.out.println("Enter 1,2,or 3 for each type of dog");
             for (int i = 0; i < count; i++)
             {
                    System.out.print("? ");
                    dogNum = MyInput.readInt();
                    switch (dogNum)
                    {
                            case  1 : dogList[i] = new Collie();break;
                            case  2 : dogList[i] = new Poodle();break;
                            case  3 : dogList[i] = new Chihuahua ();break;
                    }
              }
             Doggie.allDogs(dogList);
}
```

First run of the program:

```
How many dogs?
3
Enter 1,2,or 3 for each type of dog
? 2
? 1
? 3

My name is Pierre
I weigh 25 lbs
L'arf


My name is Lassie
I weigh 100 lbs
Woof


My name is Brutus
I weigh 5 lbs
Squeak

Press any key to continue . . .
```

A second run of the program:

```
How many dogs?
2
Enter 1,2,or 3 for each type of dog
? 3
? 1

My name is Brutus
I weigh 5 lbs
Squeak


My name is Lassie
I weigh 100 lbs
Woof

Press any key to continue . . .
```

So what is happening?
The call **Doggie.allDogs(dogList);**

passes to allDogs the array dogList which is an array of Dog. Such an array can refer to objects of any of the classes which implement Dog, i.e., dogList[i] can refer to a Collie, Poodle, or Chihuahua object.

The choice of the various implementations of name(), bark() and weight() is determined not at compile time but at runtime. That's right – late binding.

On one hand, inheritance allows that Collie, Poodle and Chihuahua objects can all be considered objects of type Dog. All belong to the same hierarchy.   All can be considered the same (parent) type. All belong to the same Dog family. On the other hand, polymorphism untwines the differences among these types. Inheritance underscores similarities; polymorphism accentuates the differences within a family.

Now add a new class, StandardPoodle:

```
public class StandardPoodle extends Poodle
{
        public void weight()
        {
                System.out.println("80 lbs");
        }
        public void name()
        {
                System.out.println("Fifi");
        }
}
```

As in the previous example, only the test class needs to be changed:

```
public class Test1
{
   public static void main(String args[])
   {
                Dog[]  dogList;
                System.out.println("How many dogs?");
                int dogNum, count = MyInput.readInt();
                dogList = new Dog[count];
                System.out.println("Enter 1,2,3, or 4 for each dog");
                for (int i = 0; i < count; i++)
                {
                        System.out.print("? ");
                        dogNum = MyInput.readInt();
                        switch (dogNum)
                        {
                                case  1 : dogList[i] = new Collie();break;
                                case  2 : dogList[i] = new Poodle();break;
                                case  3 : dogList[i] = new Chihuahua ();break;
                                case  4 : dogList[i] = new StandardPoodle();break;
                        }
                 }
                Doggie.allDogs(dogList);
         }
}
```

Output:

```
How many dogs?
4
Enter 1,2,3, or 4 for each dog
? 4
? 3
? 2
? 1

My name is Fifi
I weigh 80 lbs
L'arf

My name is Brutus
I weigh 5 lbs
Squeak

My name is Pierre
I weigh 25 lbs
L'arf

My name is Lassie
I weigh 100 lbs
Woof
```

Example:
This example demonstrates the necessity of casting:

```
public class Parent
{
   public hello()
   {
       System.out.println(“Hi”);
    }

}
```

```
public class Child1 extends Parent
{

            public void goodbye()
            {
               System.out.println("Goodbye");
             }
            public void  hello()
          {
                System.out.println(“Hi”);
            }
}
```

```
public class Child2 extends Parent
{
            public void hello()
            {
               System.out.println("Bonjour");
            }
            public void goodbye()
            {
              System.out.println("Au revoir");
             }
}
```

Since every object of a subclass **is-a** object of the superclass the following code is acceptable

```
            Parent x;
            x = new Child1();
            x.hello();
```

This is an example of *polymorphism* (“many forms”). Java will pick the correct hello method.

However, the following code will not even **compile.**

```
Parent x;
x = new Child1();
x.hello();
x.goodbye();
// To the compiler, x is a Parent reference and Parent has no goodbye() method.
```

A cast fixes the problem:

```
((Child1).x).goodbye();
```

Finally, consider the fragment:

```
Parent x;
x = new Child2();
x.hello();
```

The output is

```
Bonjour
```

Although x is declared of type Parent, Java uses the hello() method of Child2. This is another example of *late (dynamic) binding*. At run time, Java decides which version of hello() to use. Parent.hello() or Child2.hello().

One final example, illustrates inheritance, polymorphism and the advantages and necessity of overriding methods of the Object class.

Example:
The following class Student

- Overrides the toString() function inherited from Object.
- Overrides the equals() function inherited from Object with equality based on the id field.
- Implements Comparable (using the id field as a basis of comparison).

The rest of the class is fairly straightforward.

```
public class Student implements Comparable
{
        private String name;
        private String id;
        private double gpa;

        public Student()
        {
                name = "";
                id = "";
                gpa = 0.0;
        }
        public Student( String name, String id, double gpa)
        {
                this.name = name;
                this.id = id;
                this.gpa = gpa;
        }
```

```
	public boolean equals(Object x)
	{
		return id.equals(((Student)x).id);
	}

	public int compareTo(Object x)
	{
		return id.compareTo(((Student)x).id);
	}

	public String toString()
	{
		return id+"  "+ name + "  "+ gpa;
	}

	public void setName(String name)
	{
		this.name = name;
	}

	public String getName ()
	{
		return name;
	}
	public void setId(String id)
	{
		this.id = id;
	}

	public String getId()
	{
		return id;
	}
	public void setGpa(double gpa)
	{
		this.gpa = gpa;
	}

	public double getGpa()
	{
		return gpa;
	}
}
```

The class Search contains a static method that performs a binary search on an array of Object, based on a key of type Object. Remember that to perform a binary search, the array must be sorted which mandates that the array type must implement Comparable.

```
public class Search
{
	public static int search(Object [] x, Object key, int size)
	{
		int lo = 0;
		int hi = size -1;
		int mid = (lo+hi)/2;
```

```
                while ( lo <= hi)
                {
                        if (key.equals(x[mid]))
                                return mid;
                        else if (((Comparable)key).compareTo(x[mid]) < 0)

                                hi = mid -1;
                        else
                                lo = mid + 1;
                        mid = (lo+hi)/2;
                }
                return -1;
        }
}
```

The following test class builds and then searches an array of Student:

```
public class TestStudent
{
        public static void main(String args[])
        {
                Student[] s = new Student[5];
                Student key = new Student();
                String name, id;
                double gpa;
                int place;

                //populate the array
                for (int i = 0; i < 5; i++)
                {
                        System.out.print("Name: ");
                        name = MyInput.readString();
                        System.out.print("id: ");
                        id = MyInput.readString();
                        System.out.print("GPA: ");
                        gpa = MyInput.readDouble();
                        s[i] = new Student(name, id, gpa);
                }

                Sort.sort(s);
                // search based on id
                do
                {
                        System.out.print("ID: ");
                        id = MyInput.readString();
                        if (id.equals(""))
                                break;
                        key.setId(id);  // wrap id in a Student object
                        place = Search.search(s, key,5); // key is a Student object
                        if (place >= 0  &&  place < 5)
                                System.out.println(s[place]);
                        else
                                System.out.println("not found");
                 } while(true);
          }
}
```

Output:

```
// first build an array
Name: Rob Petri
id: 333
GPA: 3.1
Name: Laura Petri
id: 111
GPA: 3.5
Name: Sally Rogers
id: 444
GPA: 3.0
Name: Buddy Sorell
id: 222
GPA: 2.5
Name: Mel Cooley
id: 555
GPA: 4.0

//Search the array.  key is the id
ID: 111
111  Laura Petri  3.5
ID: 222
222  Buddy Sorell  2.5
ID:
Press any key to continue . . .
```

The above classes demonstrate *both* inheritance and polymorphism:

- Inheritance permits upcasting. An array of any *object* type can be passed to the search function since every class extends Object and every object type can be upcast to an Object. Consequently, a Student array can be passed to search(). Inheritance asserts that all classes are the same in some way; all classes are Objects.

- Polymorphism exploits differences within a hierarchy. Notice that search() method makes a call to equals:
  ```
  if (key.equals(x[mid]))
      return mid;
  ```
  The apparent type of key is Object. The real type of key (in this example) is Student. Java chooses the correct implementation of equals at run time – polymorphism in action.

Recall that earlier we mentioned that it is advisable to override the equals method inherited from Object rather than defining a new equals method in a class. Suppose that Student implements equals not by overriding the equals(Object o) method inherited from Object but as:

```
boolean equals(Student x)
{
        return id.equals(x.id);
}
```

The method certainly performs correctly under most circumstances. However, take another look at the search method. In the expression

```
key.equals(x[mid])
```

the apparent type of key is Object and the real type of key is Student. x[mid] is of type Object and because Student did not override equals(Object o) the version of equals defined in Object is chosen. There is no other choice. Therefore, Equality is not based on the id field but on reference equality, and none of our searches would return true. Polymorphism is broken.

The above example punctuates the close relationship between polymorphism and inheritance. Moreover, the example underscores the benefits of inheriting from Object.