Polymorphism

Polymorphism literally means "many shapes". Inheritance allows you to define a base class and derive classes from the base class. Polymorphism allows you to make changes in the method definition for the derived classes and have those changes apply to the methods written in the base class. This all happens automatically in Java, but you must understand the process to use it correctly.

Consider the `Man` and `Woman` classes that we derived from `Person` in the inheritance lecture. Let's say that we have an array of people and we add several Man and Woman objects to it:

```
Person[] people = new Person[3];

people[0] = new Man(15, 125, 5, 9);
people[1] = new Woman(15, 125, 5, 9);
people[2] = new Man(30, 200, 6, 5);


for (int i = 0; i < people.length; i++)
{
     System.out.println("Person " + i +
          " should eat " + people[i].howManyTwix() +
          " twix bars.");
}
```

In this case we are assigning an object of a derived class (either Man or Woman) to a variable defined as an ancestor of the derived class (Person). This is valid because Person encompasses the derived classes. In other words, Man "is-a" Person and Woman "is-a" Person, so we can assign either one to a variable of type Person.

What will be the output of this program? It is logical to assume that the `howManyTwix` method defined in the `Person` object will be invoked, since the array is created of type `Person`. But that is not what happens! Instead, Java recognizes that an object of type `Man` is stored in `people[0]`. As a result, even though `people[0]` is declared to be of type `Person`, the method associated with the class used to create the object is invoked. This is called **dynamic binding** or **late binding**.

More precisely, **when an overridden method is invoked, its action is the one defined in the class used to create the object using the new operator**. It is not determined by the type of the variable naming the object. A variable of any ancestor class can reference an object of a descendant class, but the object always remembers which method actions to use for every method name. The type of the variable does not matter. What matters is the class name when the object was created.

The output in this case is:

```
Person 0 should eat 2.80555 twix bars  ← Output for a 125 lb. 5'9 man
Person 1 should eat 2.51330 twix bars  ← Output for a 125 lb. 5'9 woman
Person 2 should eat 3.49729 twix bars
```

Note that the method defined in `Person` is not invoked, otherwise the number returned would be 0.

One of the amazing things about polymorphism is it lets us invoke methods that might not even exist yet! For example, assume the program runs with only the `Person`, `Man`, and `Woman` classes defined. At some later date we could write a `Child` class that is also derived from `Person`. As long as each implements a `howManyTwix` method then we could add one of these objects to the array and its `howManyTwix` method would be invoked in the for loop. We wouldn't even need to recompile the class to invoke the new methods via dynamic binding.

Polymorphism Example

Let's start with a simple polymorphism example: calculating the area of a shape. Let's say that we have a shape class:

```
public class Shape {
    public Shape()
    {
    }
    public double getArea()
    {
        // Don't know how to compute the area, so return 0 for now
        return 0;
    }
}
```

Now, let's make a Circle and a Rectangle class that are derived from the Shape class:

```
public class Circle extends Shape {
    private double radius;
    public Circle()
    {
        radius = 0;
    }
    public Circle(double r)
    {
        radius = r;
    }
    @Override
    public double getArea()
    {
        return Math.PI * radius * radius;
    }
}
```

```
public class Rectangle extends Shape {
    private double width, height;
    public Rectangle()
    {
        width = 0;
        height = 0;
    }
    public Rectangle(double w, double h)
    {
        width = w;
        height = h;
    }
    @Override
    public double getArea()
    {
        return width * height;
    }
}
```

Here is how we might use them in a main method. Imagine we have pizzas of different shapes. We could make a method to compare the area of the pizzas to see which one is bigger.

```
    // This will compare any two shapes as long as they
    // implement the getArea method!
    public static int compareShapes(Shape s1, Shape s2)
    {
        if (s1.getArea() < s2.getArea())
            return -1;
        else if (s1.getArea() == s2.getArea())
            return 0;
        else
            return 1;
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Circle roundPizza = new Circle(6);
        Rectangle rectPizza = new Rectangle(12,9);

        if (compareShapes(roundPizza, rectPizza)>=0)
            System.out.println("The round pizza is bigger or the
same!");
        else
            System.out.println("The rectangular pizza is bigger!");
```

Abstract Classes

You may have noticed that in our Shape example, we should never be creating an instance of `Shape`. This is because this class is an abstract notions of generalized shapes, and doesn't actually exist. We should only be dealing with instances of `Circle` or `Rectangle`. The situation was identical with our example using the `Man/Woman` classes that are instances of `Person`.

To address this issue, we could make `Shape` class **abstract**. An abstract class behaves like a normal class, except we are not allowed to make instances of it. To make the class abstract, just add the keyword extract in front of the class name:

```
      public abstract class Shape
      {
            …
      }
```

We can also make a method abstract. This means that the method must be overridden and defined in any derived class. For an abstract method we don't define anything, e.g.

```
      public abstract double getArea();
```

Interfaces

Java also supports something called an **interface**. An interface is similar to an abstract class. It defines what methods a class must **implement**. A class can implement multiple interfaces. You will use interfaces quite a bit in CSCE A222 and beyond.

An interface is something like the extreme case of an abstract class but an interface is not a class. It is, however, a type that can be satisfied by any class that implements the interface. An interface is a property of a class that says what methods it must have.

An interface specifies the headings for methods that must be defined in any class that implements the interface. For example, we can make an interface that specifies the methods that a class must have if it is driveable:

```
public interface Driveable
{
  public void start();
  public void stop();
  public void turn();
}
```

This says that anything that can be Driveable must have a method to Start, Stop, and Turn. We can't implement any code for these methods in the interface.

When we define a class that is Driveable then we say that the class **implements** the Driveable interface. Here is an example:

```
public class SportsCar implements Driveable
{
  public void start()
  {
    System.out.println("Step on the gas");
  }
  public void stop()
  {
    System.out.println("Step on the brake");
  }
  public void turn()
  {
    System.out.println("Turn steering wheel");
  }
  public void engineCheck()
  {
    System.out.println("Engine OK");
  }
}
```

```
public class Bicycle implements Driveable
{
  public void start()
  {
    System.out.println("Start pedaling");
  }
  public void stop()
  {
    System.out.println("Press hand brake");
  }
  public void turn()
  {
    System.out.println("Turn handlebars");
  }
  public void ringBell()
  {
    System.out.println("Ring Ring");
  }
}
```

Here is a class that shows how we might use the interface. Let's say we need to drive some kind of vehicle forward, but we want it to work with anything that is driveable, whether it is a Bicycle or Sportscar:

```
  public static void main(String[] args)
  {
    SportsCar s = new SportsCar();
    Bicycle b = new Bicycle();
    goForward(s);
    goForward(b);
  }

  public static void goForward(Driveable vehicle)
  {
    vehicle.start();
    System.out.println("Wait a few seconds");
    vehicle.stop();
    System.out.println();
  }
```

Here the GoForward method invokes the corresponding method for either the car or the bicycle. The output is:

```
     Step on the gas
     Wait a few seconds
     Step on the brake

     Start pedaling
     Wait a few seconds
     Press hand brake
```

Exploring Interfaces

Java has a number of classes and methods built into the standard libraries that expect us to use interfaces in particular ways. This material covers a few of those interfaces.

Java has a static method named `sort` in the Arrays class that can be used to sort an array. Here is an example:

```
import java.util.Arrays;

public class SortDemo
{
	public static void main(String[] args)
	{
		int[] nums = new int[4];

		nums[0] = 3;
		nums[1] = 45;
		nums[2] = 1;
		nums[3] = 23;

		Arrays.sort(nums);

		for (int i : nums)
		{
			System.out.println(i);
		}
	}
}
```

Upon compiling and running this code it will sort the array of numbers and output 1,3, 23, 45. What if we want to sort an array of something else? For example, consider the Fruit class and the SortDemo class below. Here is an attempt to make an array of fruit and sort them.

```
public class Fruit
{
        private String fruitName;
        public Fruit()
        {
                fruitName = "";
        }
        public Fruit(String name)
        {
                fruitName = name;
        }
        public void setName(String name)
        {
                fruitName = name;
        }
        public String getName()
        {
                return fruitName;
        }
}

import java.util.Arrays;
public class SortDemo
{
        public static void main(String[] args)
        {
                Fruit[] fruits = new Fruit[4];

                fruits[0] = new Fruit("Orange");
                fruits[1] = new Fruit("Apple");
                fruits[2] = new Fruit("Kiwi");
                fruits[3] = new Fruit("Durian");

                Arrays.sort(fruits);

                // Output the sorted array of fruits
                for (Fruit f : fruits)
                {
                        System.out.println(f.getName());
                }
        }
}
```

This program should compile but give an error when you run it. This error occurs because Java doesn't know how to compare two instances of the Fruit class to each other to see if one "comes after" the other when attempting to sort the array. More precisely, the Arrays.sort method has been written with the expectation that the objects passed in the array have a **compareTo method** in accordance with the **Comparable interface**. The program worked with the array of integers because Java has defined the compareTo method for Integer objects.

The Comparable interface specifies only a single method:

```
          public int compareTo(Object other);
```

The compareTo method is to be written by the programmer and should return

- a negative number if the calling object "comes before" the parameter other,
- a zero if the calling object "equals" the parameter other, and
- a positive number if the calling object “comes after” the parameter other

Here is a modified version of the Fruit class that implements the Comparable interface and defines the compareTo method.

```
public class Fruit implements Comparable
{
        private String fruitName;
        public Fruit()
        {
                fruitName = "";
        }
        public Fruit(String name)
        {
                fruitName = name;
        }
        public void setName(String name)
        {
                fruitName = name;
        }
        public String getName()
        {
                return fruitName;
        }
        public int compareTo(Object obj)
        {
               // This makes sure the object is a fruit
                if ((obj != null) &&
                    (obj instanceof Fruit))
                {
                 Fruit otherFruit = (Fruit) obj; // Typecast object to a fruit
                 return (fruitName.compareTo(otherFruit.fruitName));
                        // Use String compareTo
                }
                return -1;   // Default if other object is not a Fruit
        }
}
```

This new version first checks if the object the Fruit is being compared to is another Fruit. If it is, the object is typecast to a Fruit. Then it uses the fruitName and compares it to the other fruit name using the String compareTo method. If the fruit name is alphabetically before the other fruit name then -1 is returned. If they are the same then 0 is returned. if the fruit name is alphabetically after the other fruit name then 1 is returned. This all happens in the fruitname.compareTo method.

Here is an alternate version of the Fruit class with a different compareTo implementation:

```
public class Fruit implements Comparable
{
        private String fruitName;
        public Fruit()
        {
                fruitName = "";
        }
        public Fruit(String name)
        {
                fruitName = name;
        }
        public void setName(String name)
        {
                fruitName = name;
        }
        public String getName()
        {
                return fruitName;
        }
        public int compareTo(Object o)
        {
                if ((o != null) &&
                    (o instanceof Fruit))
                {
                        Fruit otherFruit = (Fruit) o;
                        if (fruitName.length() >
otherFruit.fruitName.length())
                                return 1;
                        else if (fruitName.length() <
otherFruit.fruitName.length())
                                return -1;
                        else
                                return 0;

                }
                return -1;  // Default if other object is not a Fruit
        }
}
```

This new version will sort the fruits by length of the string.
Similarly, Arrays.binarySearch will perform a binary search of the array. It requires that the array be sorted and then needs compareTo to be implemented so that it can search the array for a target.

```
System.out.println(Arrays.binarySearch(fruits, new
Fruit("Orange")));
```

Returns index 3 where Orange is found.

```
System.out.println(Arrays.binarySearch(fruits, new
Fruit("Banana")));
```

Returns a negative number indicating no such fruit found.

Here is one additional example. This one doesn't require implementing an interface, but show the importance of overriding certain methods inherited from the Object class.
The first is Arrays.toString(arrayname). This method outputs the array as a string. If we try it with our fruit array:

```
      System.out.println(Arrays.toString(fruits));
```

We just get a bunch of weird pointers output. This is because the Arrays.toString method will invoke the toString method for each fruit object. If we just override the toString method we will get meaningful output:

```
      public String toString()
      {
            return fruitName;
      }
```