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:

```java
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);
```

```java
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 – Guessing Game

Consider a guessing game where someone is thinking of a number from 0-99 and two other players try to guess the number. The players are told if the guess is too high or too low if they are not correct.

Here is the main game code with most of the logic in the playRound method:

```java
public class Game {
    // Return false if the game is over
    // True if it is not
    public static boolean playRound(Player p1, Player p2, int number) {
        // Get P1's guess
        System.out.println("Player 1: " + p1.getName() + ", it is your turn.");
        int p1Guess = p1.getGuess();
        System.out.println(p1.getName() + " guessed " + p1Guess);
        if (p1Guess == number) {
            System.out.println("That is the correct number!");
            return false;
        } else if (p1Guess < number) {
            System.out.println("The guess is too low.");
        } else {
            System.out.println("The guess is too high.");
        }

        // Get P2's guess
        System.out.println("Player 2: " + p2.getName() + ", it is your turn.");
        int p2Guess = p2.getGuess();
        System.out.println(p2.getName() + " guessed " + p2Guess);
        System.out.println("Player 2, " + p2.getName() + " guessed " + p2Guess);
    }
```
if (p2Guess == number)
{
    System.out.println("That is the correct number!");
    return false;
}
else if (p2Guess < number)
    System.out.println("The guess is too low.");
else
    System.out.println("The guess is too high.");
return true;
}

public static void main(String[] args)
{
    HumanPlayer p1 = new HumanPlayer("Kenrick");
    ComputerPlayer p2 = new ComputerPlayer("Tobor");
    int numToGuess = (int) (Math.random() * 100);

    while (playRound(p1, p2, numToGuess))
    {
        System.out.println();
        System.out.println("Starting next round.");
    }
}

The key part about this method is it takes two arbitrary Player objects and then invokes the getGuess() method to get the player’s guess.

We can implement the getGuess() method different ways depending upon the player. The nice thing about polymorphism is that Java automatically uses the method defined for the Player object passed in. In our case we’ll define a Player class and a HumanPlayer and a ComputerPlayer derived from it.
public class Player {
    private String name;

    public Player() {
        name = "Unknown";
    }
    public Player(String theName) {
        this.name = theName;
    }

    public String getName() {
        return name;
    }

    public int getGuess() {
        return 0;
    }
}

import java.util.Scanner;
public class HumanPlayer extends Player {
    public HumanPlayer() {
        super();
    }
    public HumanPlayer(String name) {
        super(name);
    }

    public int getGuess() {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter your guess.");
        int guess = keyboard.nextInt();
        return guess;
    }
}
public class ComputerPlayer extends Player {
    public ComputerPlayer() {
        super();
    }
    public ComputerPlayer(String name) {
        super(name);
    }
    public int getGuess() {
        int guess = (int) (Math.random() * 100);
        return guess;
    }
}

We’ve made a really dumb implementation for `getGuess()` for the computer. It just guesses a random number. The human player just inputs a guess from the keyboard.

When this program is run it will alternate between inputting a guess from the human from the keyboard (for player 1) and randomly guessing a number for player 2. If we wanted to make two computer players we simply change `p1` to an instance of `ComputerPlayer` and re-run the program. We could make a smarter AI program as well and plug it right in (although it would need additional methods to tell the AI if the guess was too high or too low).

**Abstract Classes**

Finally, you may have noticed that in our game example, we should never be creating an instance of `Player`. This is because this class is an abstract notion of generalized players, and don’t actually exist. We should only be dealing with instances of `HumanPlayer` or `ComputerPlayer`. The situation was identical with our example using the Man/Woman classes that are instances of `Person`.

To address this issue, we could make `Player` 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 `abstract` in front of the class name:

```java
public abstract class Player {
    ...
}
```

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.

```java
public abstract int getGuess();
```
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 CS A202.

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:

```java
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:

```java
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