Polymorphism

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism can be demonstrated with a minor modification to the `Bicycle` class. For example, a `printDescription` method could be added to the class that displays all the data currently stored in an instance.

```
public void printDescription(){
    System.out.println("\nBike is " + "in gear " + this.gear
        + " with a cadence of " + this.cadence +
        " and travelling at a speed of " + this.speed + ". ");
}
```

To demonstrate polymorphic features in the Java language, extend the `Bicycle` class with a `MountainBike` and a `RoadBike` class. For `MountainBike`, add a field for `suspension`, which is a `String` value that indicates if the bike has a front shock absorber, `Front`. Or, the bike has a front and back shock absorber, `Dual`.

Here is the updated class:

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
               int startCadence,
               int startSpeed,
               int startGear,
               String suspensionType){
        super(startCadence,
              startSpeed,
              startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
      return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }

    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

Note the overridden `printDescription` method. In addition to the information provided before, additional data about the suspension is included to the output.

Next, create the `RoadBike` class. Because road or racing bikes have skinny tires, add an attribute to track the tire width. Here is the `RoadBike` class:

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
```

```
                    int newTireWidth){
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
      return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}
```

Note that once again, the `printDescription` method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: `Bicycle`, `MountainBike`, and `RoadBike`. The two subclasses override the `printDescription` method and print unique information.

Here is a test program that creates three `Bicycle` variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;

    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);

    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}
```

The following is the output from the test program:

```
Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.
```

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as *virtual method invocation* and demonstrates an aspect of the important polymorphism features in the Java language.

http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html