# CS225L Lab 5: Inheritance and Polymorphis2 m

# Learning Outcomes

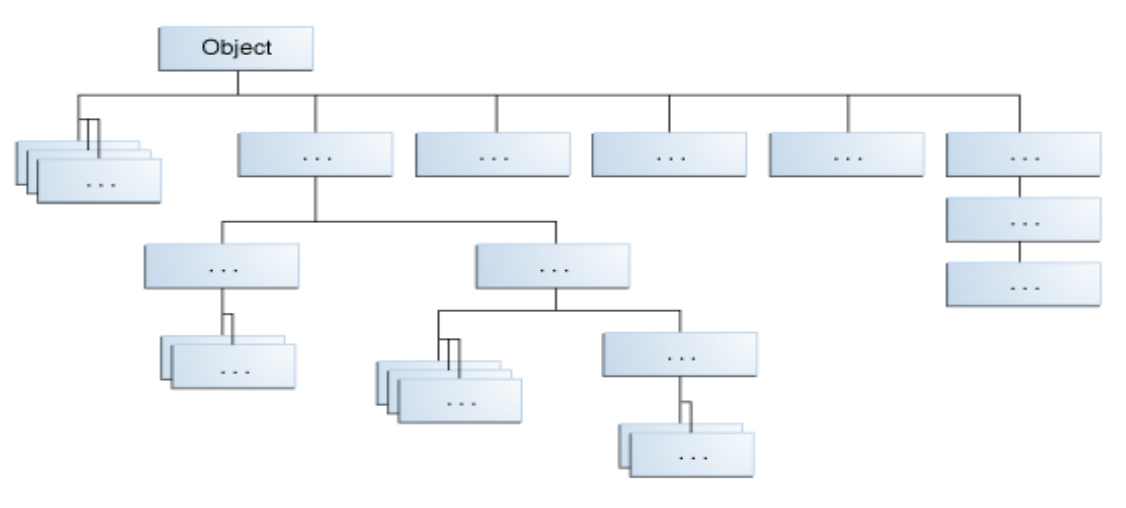
* Understand the concept of inheritance in object-oriented programming
* Utilize polymorphic behavior in object-oriented programming
* Use the keywords extends, implements, and super in a Java program

to implement these concepts

# Pre-lab

**Inheritance**

The Object class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

Here is the sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lab:

public class Bicycle {

    // the Bicycle class has three fields

    public int cadence;

    public int gear;

    public int speed;

    // the Bicycle class has one constructor

    public Bicycle(int startCadence, int startSpeed, int startGear) {

        gear = startGear;

        cadence = startCadence;

        speed = startSpeed;

    }

    // the Bicycle class has four methods

    public void setCadence(int newValue) {

        cadence = newValue;

    }

    public void setGear(int newValue) {

        gear = newValue;

    }

    public void applyBrake(int decrement) {

        speed -= decrement;

    }

    public void speedUp(int increment) {

        speed += increment;

    }

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 + ". ");

}

}

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field

    public int seatHeight;

    // the MountainBike subclass has one constructor

    public MountainBike(int startHeight,

              int startCadence,

                        int startSpeed,

                        int startGear) {

        super(startCadence, startSpeed, startGear);

        seatHeight = startHeight;

    }

    // the MountainBike subclass adds one method

    public void setHeight(int newValue) {

        seatHeight = newValue;

    }

}

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the Bicycle class were complex and had taken substantial time to debug.

**What You Can Do in a Subclass**

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

* The inherited fields can be used directly, just like any other fields.
* You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
* You can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
* You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
* You can declare new methods in the subclass that are not in the superclass.
* You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

**Polymorphism**

To demonstrate polymorphic features in the Java language, let’s extend the Bicycle class with a MountainBike in a different way. We 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’s 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.

**Reading Materials**

1. Inheritance from java Tutorial: <https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>
2. Polymorphism from Java Tutorial: <https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html>
3. Derek Banas on Youtube: <https://www.youtube.com/watch?v=Lsdaztp3_lw&list=PLE7E8B7F4856C9B19&index=14>

**Lab Activities**

Racer Game Requirements:

1. The program shall have the race be 100 units long.
2. The program shall have 4 racers in the race.
3. Racer1 shall move randomly either 4 or 8 units per turn.
4. Racer2 shall move randomly 2 to 10 units per turn.
5. Racer3 shall move randomly either
   1. A range from 0 to 10
   2. 7 \* cos(7)
6. Racer4 shall move spaces per turn but must move at least 1 unit.
7. The program shall get each racer a name.
   1. Racer 1’s name is Urza.
   2. Racer 2’s name is Fenix.
   3. Racer 3’s name is Drek.
   4. Racer 4’s name is Dijkstra.
8. The program shall end the race once 1 racer passes the ending point.
9. The program shall print out each racers name and current location each turn
10. The program shall print out who won the race.

Lab 4 Requirements:

1. The program shall use Class called Racer1 which extends GenericRacer to handle Racer1.
2. The program shall use Class called Racer2 which extends GenericRacer to handle Racer2.
3. The program shall use Class called Racer3 which extends GenericRacer to handle Racer3.
4. The program shall use Class called Racer4 which extends GenericRacer to handle Racer4.
5. The program shall use an array of type GenericRacer handle the race.

Given these requirements, build the racing game using the java files given.