# CS225L Lab 8: Inheritance and Polymorphism

# Learning Outcomes

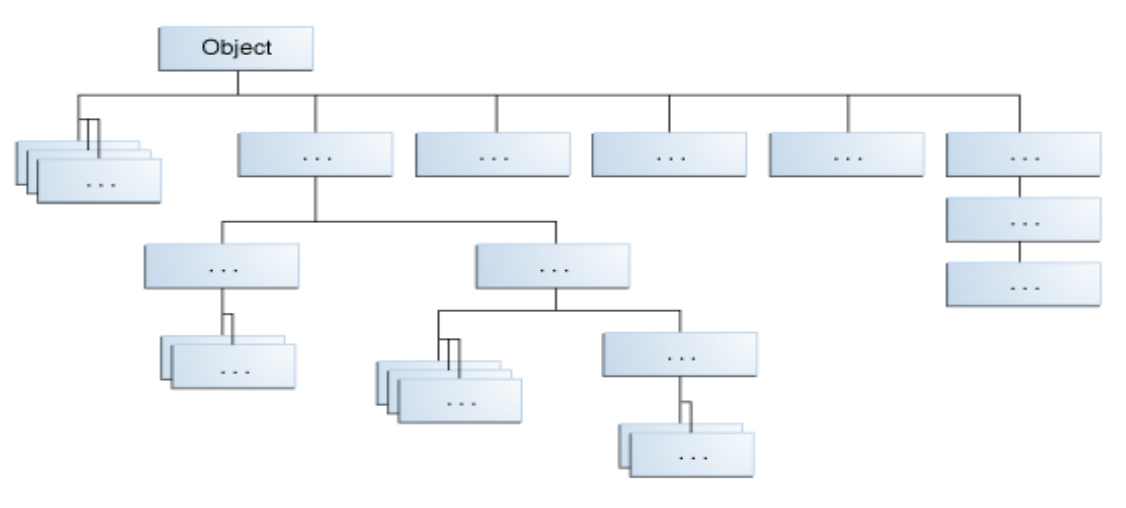
* Understand the concept of inheritance in object-oriented programming
* Utilize polymorphic behavior in object-oriented programming
* Use the keywords extends, implements, and super in a Java program

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**

In this lab, you will develop a SumarineSandwich class which extends Sandwich class based on the given UML diagram.

**Instructions**

1. Start by importing the lab files into Eclipse as an existing project.
2. Consider the UML diagram in Figure 1. It illustrates the entire program you were given, with one important exception: a whole class is missing!
3. Your task is to implement the missing SubmarineSandwich class, which extends the Sandwich class. While the code was lost, the specification of the class's fields and methods was kept in the UML diagram. For reference, here is the class diagram containing only SubmarineSandwich:



*Note: Your completed class should match this specification; you should not have any extra fields, constructors, or methods implemented.*



1. First, the constructors for the class should be implemented. When creating a SubmarineSandwich, one should specify the length of the sandwich in addition to the other attributes. These should follow the same pattern as the Sandwich class, as seen in the UML and the provided code.

Remember that the superclass constructors already take care of the work of setting a Sandwich's breadType and components. For the no-argument constructor, create a SubmarineSandwich with a sensible default length, such as 6 or 12. **(each constructor method worth 5 points, so 20 points in all, )**

1. With the constructors done, the only thing that should prevent you from running the main method in Connoisseur is a getter method for length. Implement that, then run the program. **(getLength() method worth 5 points)**
2. Next, the computation for getBitesLeft() needs to be implemented for the SubmarineSandwich class. To accomplish that, the method can be overridden. A given SubmarineSandwich can have a different “capacity” for bites taken out of it that depends on its length. To be more detailed, the maximum number of bites is simply 2 \* length and the number of bites left should be 2 \* length – number of bites already taken.

However, unlike breadType and components, the numBites field in Sandwich is marked private, so it is not accessible to your subclass. Instead of using that variable directly, you need to call the getter method from the superclass. (Hint: to call a method in a super class, use super.methodName()) **(getBitesLeft() method worth 5 points)**

1. Finally, the toString() method needs to be overridden to reflect the new attribute added to SubmarineSandwich. You can follow the pattern seen in Sandwich, except you need to incorporate length somehow. You do not need to incorporate numBites. **(toString() method worth 5 points)**
2. If it looks good, double check to make sure that your code matches the specification (the UML diagram) and is well-commented. When you're ready, go ahead and do the post-lab questions.

**Post Lab Deliverables**

First submit your SubmarineSandwich.java source code, then answer the below questions in a separate document: **(5 Points)**

1. After you overrode getBitesLeft() in SubmarineSandwich, you did not have to override isEaten(); the computation of that automatically reflected the number of bites left in SubmarineSandwiches. How was that possible? (Look at the code for isEaten(). . . )
2. Provide feedback:

(a) What is your lab section? (Section # or day of the week)

(b) What did you like about the lab? What did you dislike?

(c) What would you change about this lab to improve it?

To be clear, you should turn in two files, SubmarineSandwich.java and a document file containing your answers to the above questions.

Total Points from Lab 8: **40 Points**