# CS225L Lab 4: Objects

# Learning Outcomes

* What is an object
* Difference and relationship between object and class
* How to create, use, modify objects
* How to call object methods and update object attributes
* Implement object methods to achieve specific functionalities

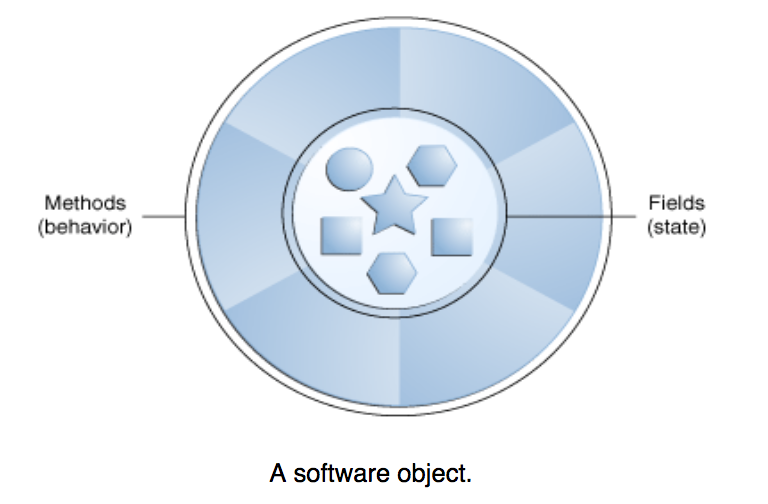
# Pre-lab

**What is an object?**

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

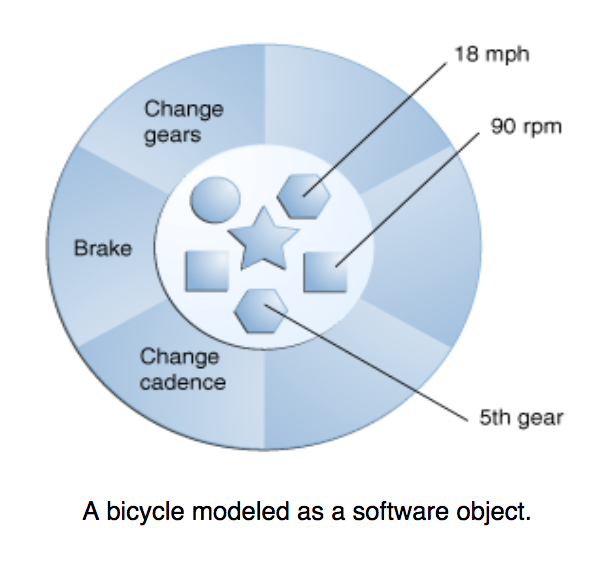
Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:



By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

**What is a class?**

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

The following [Bicycle](http://docs.oracle.com/javase/tutorial/java/concepts/examples/Bicycle.java) class is one possible implementation of a bicycle:

class Bicycle {

    int cadence = 0;

    int speed = 0;

    int gear = 1;

    void changeCadence(int newValue) {

         cadence = newValue;

    }

    void changeGear(int newValue) {

         gear = newValue;

    }

    void speedUp(int increment) {

         speed = speed + increment;

    }

    void applyBrakes(int decrement) {

         speed = speed - decrement;

    }

    void printStates() {

         System.out.println("cadence:" + cadence +

" speed:" + speed +

" gear:" + gear);

    }

}

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be *used* in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a [BicycleDemo](http://docs.oracle.com/javase/tutorial/java/concepts/examples/BicycleDemo.java) class that creates two separate Bicycle objects and invokes their methods:

class BicycleDemo {

    public static void main(String[] args) {

        // Create two different

        // Bicycle objects

        Bicycle bike1 = new Bicycle();

        Bicycle bike2 = new Bicycle();

        // Invoke methods on

        // those objects

        bike1.changeCadence(50);

        bike1.speedUp(10);

        bike1.changeGear(2);

        bike1.printStates();

        bike2.changeCadence(50);

        bike2.speedUp(10);

        bike2.changeGear(2);

        bike2.changeCadence(40);

        bike2.speedUp(10);

        bike2.changeGear(3);

        bike2.printStates();

    }

}

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

cadence:50 speed:10 gear:2

cadence:40 speed:20 gear:3

**Reference Materials**

1. What’s an object: <http://docs.oracle.com/javase/tutorial/java/concepts/object.html>
2. What’s a class: <http://docs.oracle.com/javase/tutorial/java/concepts/class.html>

**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 to handle Racer1.
2. The program shall use Class called Racer2 to handle Racer2.
3. The program shall use Class called Racer3 to handle Racer3.
4. The program shall use Class called Racer4 to handle Racer4.

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