CSE 21
Intro to Computing II

Lecture 6 – Object Oriented Programming (2)

Today

- Object Oriented Programming (2)
- Lab
  - Lab 7 due this week (3/11 – 3/17)
  - Lab 8 assigned this week
    - Array of Objects
  - Due in one week
    - Make sure to show your work to YOUR OWN TA (or me) before submission
      - Required to receive full credit
- Reading Assignment
  - Sections 10.1 – 10.5 (including participation activities)
    - Work on the **Participation Activities** in each section to receive participation grade at the end of semester (based on at least 80% completion)
    - Work on **Challenge Activities** to receive extra credit
  - Participation and Challenge activities evaluated at the end of semester

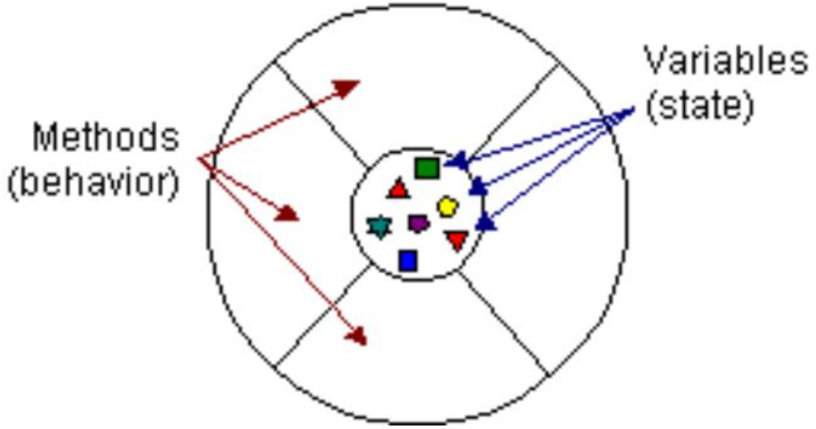
Object-Oriented Programming (review)

- Our new programming metaphor is multiple independent intelligent agents called ***Objects***
- An object can…
  - ask other objects to do things
    - this is called "message passing"
  - remember things about its own past history
    - this is called "local state"
  - behave just like another except for a few differences
    - this is called "inheritance"
- Many people find this way of thinking and modeling the world more intuitive
  - The world is made up of objects! people, desks, chairs, etc.
- Android Apps are built this way!

What is an Object? (review)

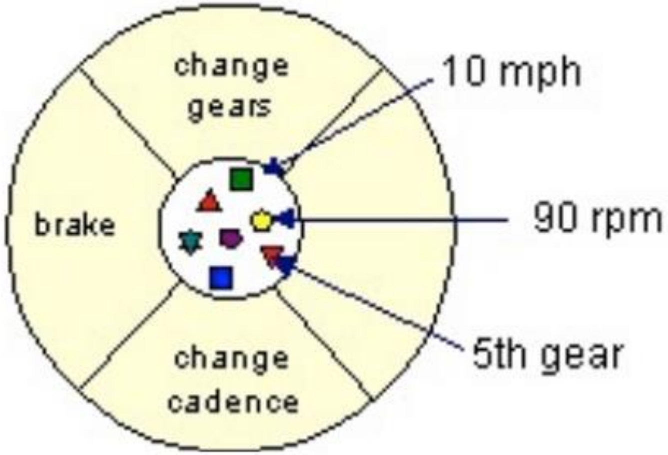
- Real-world objects share states and behaviors
  - E.g., cats have states (name, color, breed, hungry) and behaviors (meowing, sleeping, shredding rugs)
  - E.g., bikes have states (gear, # wheels, # gears) and behaviors (braking, changing gears)
- Software objects are modeled after real-world.
- A software object…
  - maintains its states in one or more variables
  - implements its behaviors with methods
  - An object is a software bundle of variables (what it knows) and related methods (what it can do)
- ***Classes*** are **"factories"** for generating objects

How can we visualize objects?



- A particular object is called an **instance**
- Its variables are called **instance variables**

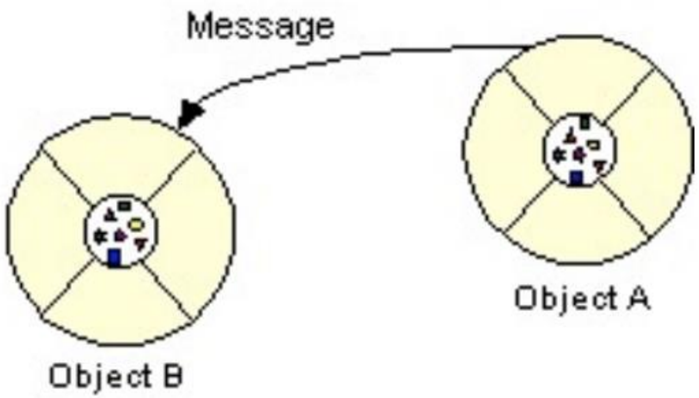
- A SW object modeling a bike:
  - States (in variables): speed = 10mph, cadence = 90rpm, gear = 5th
  - Methods: (brake, change gears, change cadence)
  - Note: no method to change speed directly, it's a side-effect of the gear and how fast you're pedaling!

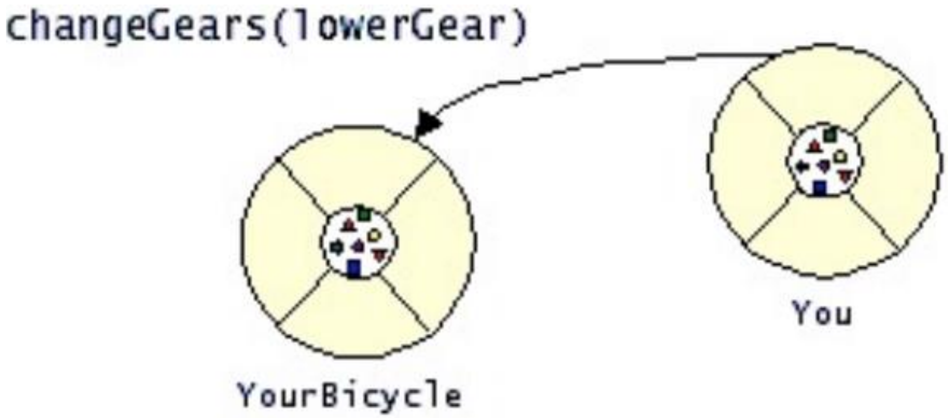


A Bike instance

What Is a Message?

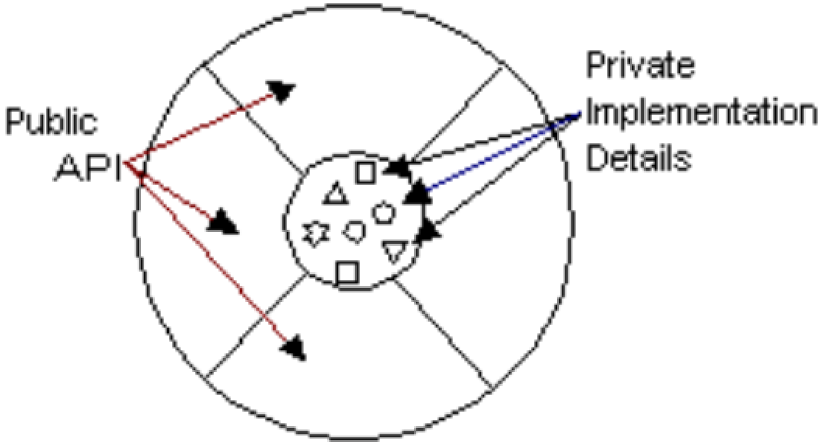
- A single object alone is not very useful…
- An object as a component of a program that has object-object interaction is powerful.
- If object A wants object B to perform one of B’s methods, A *sends a message* to B (sometimes with parameters)
- Here, you are asking `yourBicycle` to `changeGears` to `lowerGear`

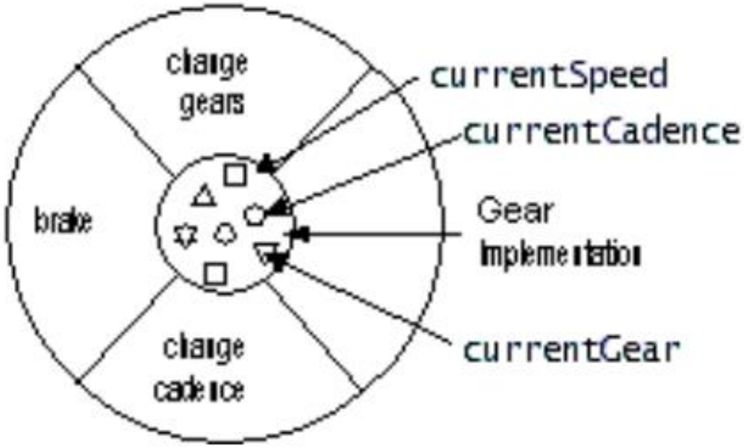


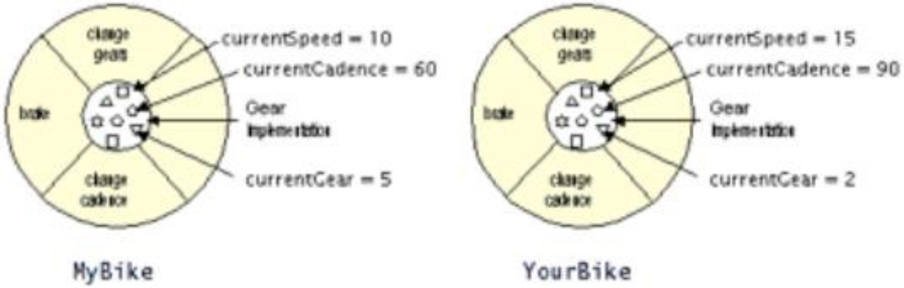


What is a Class?

- A class is the basic unit of Java. It’s the **"blueprint"** or **"factory"** that defines the variables and methods ***common*** to all objects of a certain kind.
- Methods isolate, or ***encapsulate*** the data inside from the outside.
  - Other objects ask about this object's state via methods.
- After you have your `Bike` class, you can create any number of (instances of) bike objects!







How to tell which object is doing what?

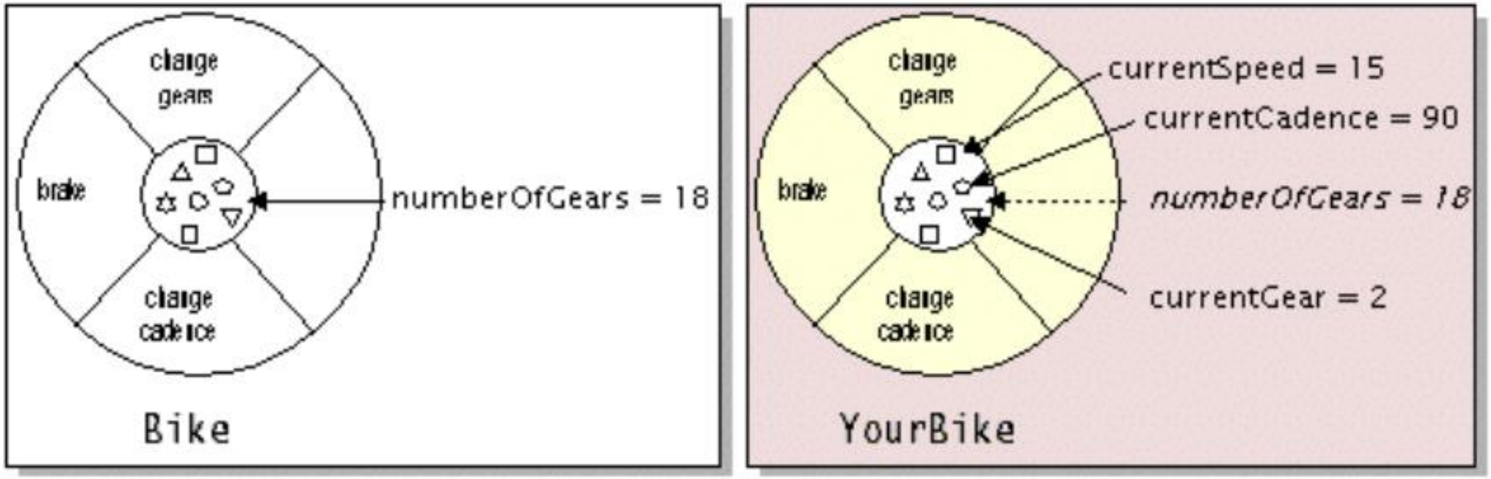
- The members of an object (instance variables and methods) are accessed using the member access operator, or dot operator (.)

- As in…

```
mybike.speed
yourbike.changeGears()
```

- Note: methods and variables can have the same name
  - Use () to disambiguate!

Instance vs. Class (static) variables



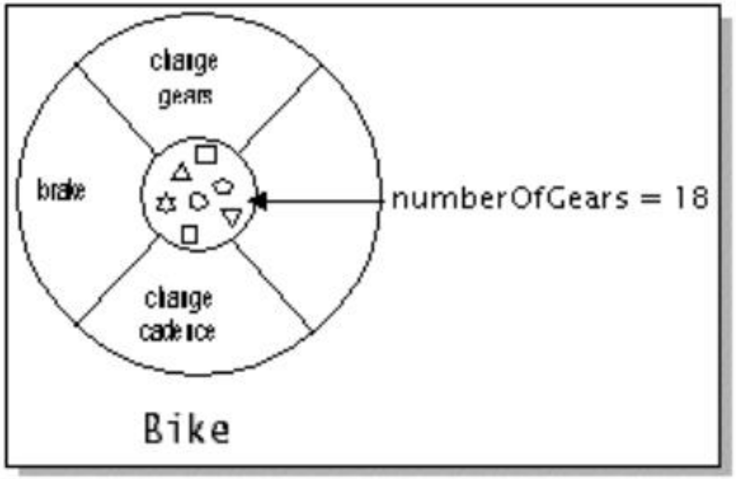
Class

Instance of a Class

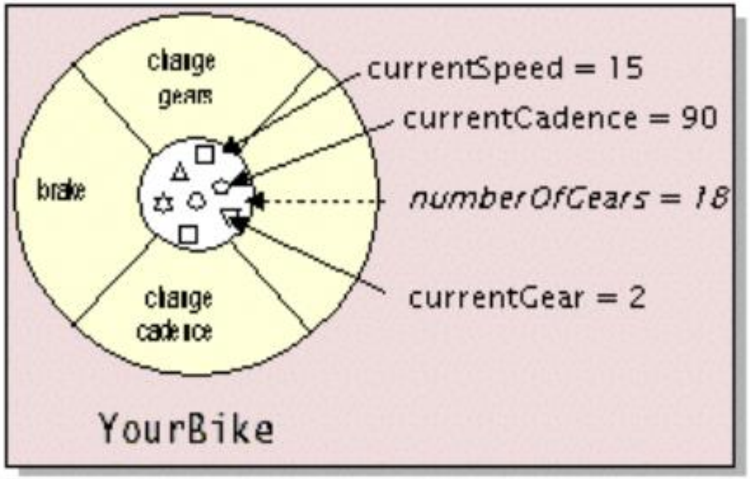
- A ***class variable*** (aka ***static variable***) is shared by all instances of the same class.
  - Unlike ***instance variables*** that can be different for each instance.
  - E.g., suppose all bikes had the same number of gears. If we made this a class variable, and we wanted to change it, it would change for ALL bikes.

```
static int numGears;
```

Instance vs. Class (static) variables



Class



Instance of a Class

- Access static variables from the class, not from an instance.

```
Bike yourbike = new Bike();
System.out.println(yourbike.currentSpeed);        // OK
System.out.println(yourbike.numberOfGears);     // NO!
System.out.println(Bike.numberOfGears);          // Good
```

Common Methods in a Class

- Methods common to many classes
  - *Constructors* are called if you ask for a ***new*** object
    - Java provides a ***default*** constructor (with no arguments)
  - ***Accessors***, or "get methods", or "getters" are used to read/retrieve the values of instance variables
    - Including predicate methods returning booleans
  - ***Mutators***, or "set methods", or "setters" are used to set the values of instance variables
  - **toString** method creates a String representation of the contents of the object
    - **System.out.println(obj)** calls object's **toString**

Designing a Class

- To design a class, think about what the objects in that class should do
  - Determine the set of variables (your states)
    - inside each object (instance variables)
    - shared by all objects in a class (class variables)
  - Determine methods (your API, or "behavior")
    - Constructors (these build an instance)
    - Accessors (these query info of your state)
    - Mutators (if any) (these change the object)

Constructors

- Constructors are called when you request a new object
  - Method Signature:

```
Template: public <ClassName>(params …) { … }
Example:  public Bike(double s) {
                   speed = s;
          }
```

There is NO return type!

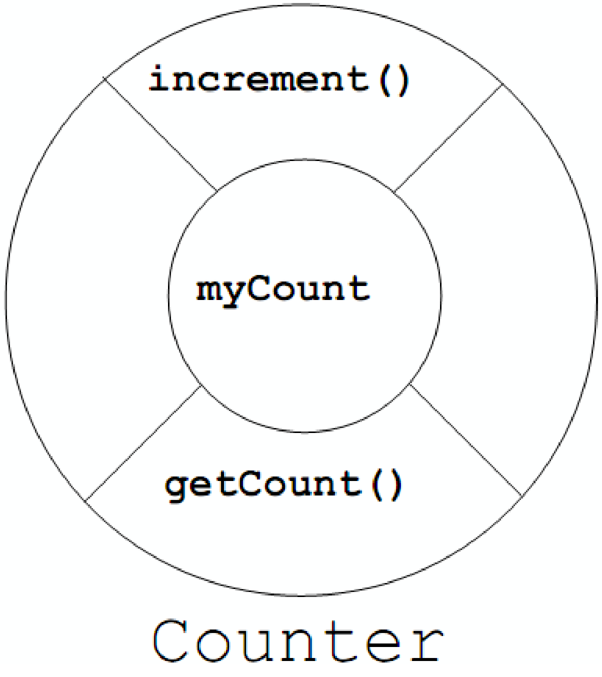
  - Called by:

```
Template: <ClassName> varName = new ClassName(args …)
Example:  Bike myBike = new Bike(3.5);
```

  - Java provides a ***default*** constructor (with no arguments)

Example: A simple counter!

- We'd like a "counter" that remembers the number of times we ask it to increment itself.
  - Determine the set of variables
    - One internal instance variable counter → **myCount**
  - Determine methods
    - Constructors
      - Use Java's default → **Counter()**
    - Accessors
      - How to query the value? → **getCount()**
    - Mutators
      - How to change the value? → **increment()**



Counter : Class Skeleton

```
/* A Counter remembers the number of times it has
 *  been asked to increment itself.
 */
public class Counter {

        /* Instance variable */
        int myCount = 0;

        /* Modify the counter by incrementing itself. */
        public void increment() { … }

        /* Return the current counter reading. */
        public int getCount() { … }
}
```

These are called method "Signatures ".
This is a design step!

Counter : Class Definition

```
/* A Counter remembers the number of times it has
 * been asked to increment itself.
 */
public class Counter {

    /* Instance variable */
    int myCount = 0;

    /* Modify the counter by incrementing itself. */
    public void increment () {
        myCount++;
    }

    /* Return the current counter reading. */
    public int getCount () {
        return myCount;
    }
}
```

Using the Counter Class (in main)

```
// Make a our first counter!
Counter c1 = new Counter(); // (c1's count reset to 0)
// Ask it (send a message to it) what its count is
c1.getCount();          ⇒  0
// Ask it to increment
c1.increment(); // (c1's count is now set to 1)
// Ask it to increment again
c1.increment(); // (c1's count is now set to 2)
// Ask it (send a message to it) what its count is
c1.getCount();          ⇒  2
// Make another counter!
Counter c2 = new Counter(); // (c2's count reset to 0)
// Ask them what their counts are
c1.getCount();          ⇒  2
c2.getCount();          ⇒  0
// Ask it to print itself
System.out.println(c2);     ⇒  Counter@34b350
                                    ???
```