CSE 21
Intro to Computing II

Lecture 6 – Object Oriented Programming (2)

Announcement

- Lab 5 due before start of next lab
  - Type your answers in a text file and submit it as an attachment
- Project #1 out this Friday (9/30)
  - Due Friday (10/14) at 11:59PM
- Reading assignment
  - Chapter 7.5 to 7.8 of textbook

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 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 an important 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 state)
    - 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:

```
public <Class> (args…) { … }
  public Bike(double s) {
      speed = s;
  }
```

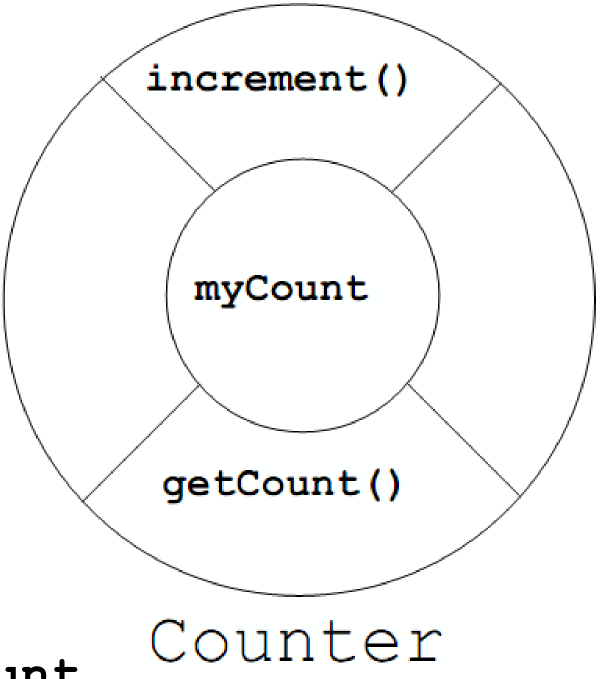
  - Called by:

```
<Class> var = new Class(args…)
  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 methods
    - Constructors
      - Use Java's default → **`Counter()`**
    - Accessors
      - How to query the value? → **`getCount()`**
    - Mutators
      - How to change the value? → **`increment()`**
  - Determine the set of variables
    - One internal instance variable counter → **`myCount`**



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

```
// 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
                                   ???
```

Let’s add a class var, toString to Counter

```
/** Class variable */
 public static int numCounters = 0;

 /** We override the default constructor */
 public Counter () {
   numCounters++;
 }

 /** Return a String representation of a Counter */
 public String toString() {
   return ("" + myCount);    // Return value of myCount as a string
 }
 //OR
 public String toString() {
   String s = new String();
   s += myCount
   return s;
 }
```

Using Counter Class

```
// Before we start, how many Counters exist?
Counter.numCounters         ⇒ 0
Counter c1 = new Counter(); // Make one
Counter.numCounters         ⇒ 1 //Can ask the Class
c1.numCounters              ⇒ 1 //Can also ask an instance
// Ask instance to increment thrice
c1.increment();             // (c1’s count is now set to 1)
c1.increment();             // (c1’s count is now set to 2)
c1.increment();             // (c1’s count is now set to 3)
// Make another counter…
Counter c2 = new Counter(); // (c2’s count reset to 0)
// Ask them what their counts are
c1.getCount()               ⇒ 3
c2.getCount()               ⇒ 0
// How many Counters exist?
Counter.numCounters         ⇒ 2 //Can ask the Class
c1.numCounters              ⇒ 2 //Can also ask an instance
c2.numCounters              ⇒ 2 //Can also ask an instance
// Ask them to print themselves
System.out.println(c1);     ⇒ 3
System.out.println(c2);     ⇒ 0
```

Review Terminology (1/2)

- Objects: some data and operations that manipulate that data
  - It can help to think of an object as a "thing“
- Variables: names for the data in objects
  - A named place to store some information pertaining to the object, that may or may not change
  - Variables are the only way to store "states"
- Methods: a procedure for the object
  - Something that the object can do
  - It is best if only methods are public – that is, other objects don't access variables directly
    - More flexibility (when inheriting, error checking)
    - Equally efficient (in most cases)

Review Terminology (2/2)

- Classes: factories for "generating" objects
- Package: a set of related classes
  - This is how you find existing code
- Project: a set of packages/classes that solve a problem (also a set of files on your computer)

Static and non-static methods

- Like variables, methods can be static or non-static.
  - static methods "reside" in the class, and should be called only with reference to a class:

```
Math.sqrt(5.0);  // "Math" is a class
```

  - non-static methods need a particular instance to be called, and do something based-on or to that instance.

```
Counter c = new Counter();
c.increment(5.0);     // "c" is an instance
```

What if increment is a static method?

```
Counter.increment(5.0)
```

It increments the counts from all instances!

Date Class Definition

```
public class Date {
 public int day;
 public int month;
 public int year;
 public Date() {                                   // Constructor 1
   day = month = year = 0;
 }
 public Date(int year) {                           // 2
  day = month = 0;
  this.year = year;
 }
 public Date(int year, int month) {                // 3
  day = 0;
  this.month = month;
  this.year = year;
 }
 public Date (int year, int month, int day) {      // 4
  this.day = day;
  this.month = month;
  this.year = year;
 }
}
```

We use “**this**” to explicitly access instance variables.

What’s wrong?

```
johnny = new Date();
johnny.month = 27;
johnny.day = -12;
johnny.year = 99999999;

johnny = new Date (13);
johnny = new Date (13, 13);
johnny = new Date (13, 13, 13);
johnny.year = 100;
```

Defensive Programming

```
public class Date {
   public Date (int month) {
    setMonth(month);
   }

   public void setMonth(int month) {
      if (month > 0 && month <= 12)
         this.month = month;
      else
         System.out.println(“Invalid month”);
   }
}
```

Incorporate error-checking mechanism!

Using dot to Access Everything

```
Date johnny = new Date( );

// instead of johnny.month = 7;
johnny.setMonth(7);  // method call

// month is a variable
System.out.println(“This person was born in
  month #” + johnny.month);
```