Assignments:
- Assignment 1 due by 11:59pm last night
- Assignment 2 is available
  - due 11:59 pm Monday 18 September 2017

Reading for next lecture is Ch. 7
- Focus on 7.1, 7.2, and 7.6
- Read the rest of Ch. 7 for next Tuesday
- Ch. 7.4 explains that String [] args thing in main!

Primitives – not objects
```java
int x = 5;
char letter = 'p';
boolean george = true;
```

Objects! String, Scanner, Integer, etc.
```java
String message = "hello";
String remark = new String("galloping lizards!");

message.length();
remark.toUpperCase();
```

Class
A class contains data declarations and method declarations
- The values of the data are the object's state
- The functionality of the methods define the object's behavior
- Generally, classes that represent tangible things are called names that are **singular nouns**:
  - Examples: **Coin**, **Student**, **Classroom**
- Generally, the methods that encapsulate behaviors are called names that are **verbs**:
  - Examples: **get, set, calculate, convert, initiate**

- Let's say you have a **PictureFrame** class – what's the point of this class? What do its objects represent?

  - What data and behaviors might the **PictureFrame** have?

- What data and behaviors might a **Meeting** object have?

- A well **encapsulated** object can be thought of as a **black box** - the inner workings are hidden from whomever is using it (the **client**)
- The client invokes the interface methods of the object, which manages the instance data

- A **modifier** specifies particular characteristics of a method or data
- Java has three visibility modifiers: **public**, **protected**, and **private**

<table>
<thead>
<tr>
<th></th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>violates</td>
<td>enforces</td>
</tr>
<tr>
<td></td>
<td>encapsulation</td>
<td>encapsulation</td>
</tr>
<tr>
<td>Methods</td>
<td>provides</td>
<td>supports</td>
</tr>
<tr>
<td></td>
<td>services to</td>
<td>other</td>
</tr>
<tr>
<td></td>
<td>clients</td>
<td>methods in</td>
</tr>
<tr>
<td></td>
<td>supports</td>
<td>class</td>
</tr>
</tbody>
</table>
Consider a six-sided die (singular of dice)
- Its state can be defined as which face is showing
- Its primary behavior is that it can be rolled

We can represent a die in Java by designing a class called Die that models this state and behavior

We want to design the Die class with other data and methods to make it a versatile and reusable resource

Let’s see how we would use a class of the Die to play snakeEyes

```java
public class SnakeEyes {
    public static void main(String[] args) {
        final int ROLLS = 500;
        int num1, num2, count = 0;
        //Declare and instantiate two new Die objects
        Die die1 = new Die();
        Die die2 = new Die();
        for (int roll = 1; roll <= ROLLS; roll++) {
            //Roll die, save each faceValue into num1 and num2
            num1 = die1.roll();
            num2 = die2.roll();
            //Check for snake eyes
            if (num1 == 1 && num2 == 1) count++;
        }
        System.out.println("Number of rolls: "+ROLLS);
        System.out.println("Number of snake eyes "+count);
        System.out.println("Ratio: "+(float)count/ROLLS);
    }
}
```

Understanding the control flow is essential to debugging!

- If the called method is in the same class, only the method name is needed
- If the called method is part of another class, use the dot notation
- Understanding the control flow is essential to debugging!
**A constructor** is a special method which builds a new instance of the class

- Note that a constructor has **no return type** in the method header, **not even** `void`

- A **common error** is to put a return type on a constructor, which makes it a "regular" method that happens to have the same name as the class

- The programmer does not have to define a constructor for a class:
  - Each class has a **default constructor** that accepts no parameters

```java
public class Die {
    private final int MAX = 6;    //max face value
    private int faceValue;       //current value showing

    public Die(){ // Constructor! Sets initial value.
        faceValue = 1;
    }

    /**
     * Computes a new face value for this die
     * @return the new face value between 1 and MAX
     */
    public int roll(){
        Random r = new Random();
        faceValue = r.nextInt(MAX) + 1;
        return faceValue;
    }

    /**
     * Face value mutator. Only modified if value is valid
     * @param value die is set to this integer, 1 to MAX
     */
    public void setFaceValue(int value){
        if (value > 0 && value <= MAX)
            faceValue = value;
    }

    /**
     * Face value accessor.
     * @return the current face value of this die
     */
    public int getFaceValue() {
        return faceValue;
    }

    /**
     * @return string representation of this die
     */
    public String toString() {
        String result = Integer.toString(faceValue);
        return result;
    }
}
```

- An **instance variable** is specific to that instance of the object (there can be many instances of an object)

- A **static variable** is belongs to the class (there is only one)

- A **static method** (or **class method**) effects the entire class, so no instance variables or methods are be called in it
Sheep dave = new Sheep(4);
//instance call, finds dave, returns 4
int pen = dave.getPenNumber();
//static call, public static shear(Sheep sheep)
Wool wool = Sheep.shear(dave);

For example, suppose the Grade class contained the following main method driver:

// Main method.. the Bronte sisters’ grades in CS230
public static void main(String[] args) {
    Grade charlotte = new Grade("B-", 82.1);
    Grade emily = new Grade("A", 94.5);
    Grade anne = new Grade("C+", 79.0);

    System.out.println(charlotte.isHigherThan(emily));
    System.out.println(Grade.max(charlotte,anne).toString());
}

Then executing the Grade application would produce the following output:

false
Letter grade:    B-
Score:           82.1

Write a Grade class that contains the following one constructor, three instance methods, and one class method:

// Constructor
public Grade(String letterGrade, double numericalGrade)

// Returns the score associated with this Grade
public double getScore()

// Returns true if this Grade is higher than Grade g
public boolean isHigherThan(Grade g)

// Returns a String representation of this grade
public String toString()

// Returns the maximum of the two Grade objects
public static Grade max(Grade g1, Grade g2)

/* Represents a coin with two sides that can be flipped.
 * For the Java Foundations
 */
public class Coin {
    private final int HEADS = 0; // tails is 1
    private int face; // current side showing

    /* Constructor: Sets up this coin by flipping it initially.
     */
    public Coin () { ... }

    /* Flips this coin by randomly choosing a face value.
     */
    public void flip () { ... }

    /* Return true if the current face of this coin is heads, false otherwise
     */
    public boolean isHeads () { ... }

    /* Return string representation of this coin
     */
    public String toString() { ... }
}
** Demonstrates the use of a programmer-defined class.
* @author Java Foundations
*
```java
public class CountFlips {
    /**
     * Driver: Flips a coin multiple times and counts the number of heads
     * and tails that result.
     */
    public static void main (String[] args) {
        final int FLIPS = 1000;
        int heads = 0, tails = 0;
        Coin myCoin = new Coin();
        for (int count=1; count <= FLIPS; count++) {
            myCoin.flip();
            if (myCoin.isHeads())
                heads++;
            else
                tails++;
        }
        System.out.println ("Number of flips: " + FLIPS);
        System.out.println ("Number of heads: " + heads);
        System.out.println ("Number of tails: " + tails);
    }
}
```

** Demonstrates the reuse of a programmer-defined class.
* @author Java Foundations
*
```java
public class FlipRace {
    /**
     * Driver: Flips two coins until one of them comes up heads three
times in a row.
     */
    public static void main (String[] args) {
        final int GOAL = 3;
        int count1 = 0, count2 = 0;
        Coin coin1 = new Coin(), coin2 = new Coin();
        while (count1 < GOAL && count2 < GOAL) {
            coin1.flip();
            coin2.flip();
            System.out.println ("Coin 1: " + coin1 + " vs Coin 2: " + coin2);
            if (coin1.isHeads())
                count1++;
            else
                count1 = 0;
            if (coin2.isHeads())
                count2++;
            else
                count2 = 0;
        }
        if (count1 < GOAL)
            System.out.println ("Coin 2 Wins!");
        else
            if (count2 < GOAL)
                System.out.println ("Coin 1 Wins!");
            else
                System.out.println ("It's a TIE!");
    }
}
```