The heart of Object-Oriented Programming

“Method” is an object-oriented word for “function”

“Call” is synonymous with “invoke”, and refers to executing a method.

For example:

// Call the parseInt method of the Integer class
// on the argument (or “actual parameter”) “123”.
Integer.parseInt(“123”);
// () is the invocation operator

Assignment 1 due Thursday at 11:59pm
- Java is a **statically typed** language
  - You must explicitly define variable types at declaration
- Unlike Python, not all variables in Java are objects
  - Some are **primitive data types** (but have related objects)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Char</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Scanner</td>
</tr>
<tr>
<td></td>
<td>File</td>
</tr>
</tbody>
</table>

- A class contains **data declarations** and **method declarations**
- An **object** is an instantiation of a class
- The **values** of the data are the **object's state**
- The **functionality** of the methods define the **object's behavior**

```java
int x, y;
char a;
```

- Generally, classes that represent tangible things are called names that are **singular nouns**:
  - Examples: Coin, Student, Classroom, File, Scanner, String, URL, Performer
- Generally, the methods that encapsulate behaviors are called names that are **verbs**:
  - Examples: get, set, calculate, convert, initiate, dance, sing
- What are the data and methods you would define for class Student?

- Let's say you have a PictureFrame class. What's the point of this class? Is it representing a product? A virtual object?
- What **data and methods** might the PictureFrame have?

- What **data and methods** might a Year object have?
  - [https://docs.oracle.com/javase/8/docs/api/java/time/Year.html](https://docs.oracle.com/javase/8/docs/api/java/time/Year.html)
Classes can have two (simultaneous) purposes

- An **instance variable** is specific to a particular instance of the class (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

How about that public thing?

- Enforces access to an object's data only through specific methods - **PROTECTS** the class implementation
- 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**
  - **private**

<table>
<thead>
<tr>
<th>Variables</th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violates</td>
<td>violates encapsulation</td>
<td>enforces encapsulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides</td>
<td>provides services to clients</td>
<td>supports other methods in 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 *Die* to play *snakeEyes*

```java
public class SnakeEyes {  
    public static void main(String[] args) {  
        final int ROLLS = 500;  
        int num1, num2, count = 0;  
        // Instantiate two new Die objects
        Die die1 = new Die();  
        Die die2 = new Die();  
        for(int roll = 1; roll <= ROLLS; roll++) {  
            // Roll die, save faceValues 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);  
    }  
}
```

- 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 and does nothing
import java.util.Random;
/**
 * Represents one die with faces between 1 and 6
 * @author Java Foundations
 */
public class Die {
    private final int MAX = 6; //max face value
    private int faceValue; //current value showing

    public Die(){ // Constructor! Sets initial value.
    }
    /**
     * Computes a new face value for this die
     * @return the new face value between 1 and MAX
     */
    public int roll(){
        Random r = new Random();
        this.faceValue = r.nextInt(MAX) + 1;
        return faceValue;
    }
    public void setFaceValue(int value){
        if(value > 0 && value <= MAX) {
            this.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(this.faceValue);
        return result;
    }
}

/*Understanding the control flow is essential to debugging! */

```
import java.util.Random;
/**
 * Represents one die with faces between 1 and 6
 * @author Java Foundations
 */
public class Die {
    private final int MAX = 6; //max face value
    private int faceValue; //current value showing

    public Die(){ // Constructor! Sets initial value.
    }
    /**
     * Computes a new face value for this die
     * @return the new face value between 1 and MAX
     */
    public int roll(){
        Random r = new Random();
        this.faceValue = r.nextInt(MAX) + 1;
        return faceValue;
    }
    public void setFaceValue(int value){
        if(value > 0 && value <= MAX) {
            this.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(this.faceValue);
        return result;
    }
}
```

“He came back and I almost said asked, but the truth is, demanded that his entire programming team get a demo of the Smalltalk System and the then head of the science centre asked me to give the demo because Steve specifically asked for me to give the demo and I said no way. I had a big argument with these Xerox executives telling them that they were about to give away the kitchen sink and I said that I would only do it if I were ordered to do it cause then of course it would be their responsibility, and that’s what they did.”

~Adele Goldberg
Understanding the control flow is essential to debugging!

- An **instance variable** is specific to a particular instance of the class (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();
//instance call, creates dave, returns 4
int pen = dave.getPenNumber();
//static call, public static shear(Sheep aSheep)
Wool wool = Sheep.shear(dave);

In a **static** method, you cannot use the **this** keyword

If you do, you’ll see an error like this:

```
Person.java:3: error: non-static variable this cannot be referenced from a static context
```

Ask yourself:
What do you mean for **this** to refer to?
Was an instance passed to this method?
Invocations of **static** methods use the class’s name, e.g.:

```java
String.join("|", thingsToJoin)
```

An **instance** method is invoked using an instance of the relevant class, e.g.:

```java
name.charAt(0);
```

---

**Discuss:**

Should `windSpeedToCategory` be a static method or an instance method?

```java
// Convert a windSpeed to a storm category
public static int windSpeedToCategory(double windSpeed)
```
// Constructor
public Hurricane(String name, double windSpeed)

// Returns the wind speed of this storm
public double getWindspeed()

// Returns the category, based on its wind speeds
public String getCategory()

// Returns true if this storm has faster winds than h
public boolean fasterWindsThan(Hurricane h)

// Returns the maximum of the two Hurricane objects
public static Hurricane max(Hurricane h1, Hurricane h2)

// Assume you start with a correct implementation of:
public static int windSpeedToCategory(double windSpeed)

Discuss fasterWindsThan and max with a neighbor

Write a Hurricane class that contains these methods:

// Constructor
public Hurricane(String name, double windSpeed)

// Returns the wind speed of this storm
public double getWindspeed()

// Returns the category, based on wind speeds
public String getCategory()

// Returns true if this storm has faster winds than h
public boolean fasterWindsThan(Hurricane h)

// Returns the maximum of the two Hurricane objects
public static Hurricane max(Hurricane h1, Hurricane h2)

// Assume you start with a correct implementation of:
public static int windSpeedToCategory(double windSpeed)

Now, suppose the Hurricane class contains the following main method driver:

// Main uses Hurricanes
public static void main(String[] args) {
    Hurricane alice = new Hurricane("Alice", 82.1);
    Hurricane bob = new Hurricane("Bob", 120.5);
    Hurricane eve = new Hurricane("Eve", 170.0);

    System.out.println(alice.fasterWindsThan(bob));
    System.out.println(Hurricane.max(alice, bob));
}

Then executing the Hurricane application would produce the following output:
    false
    Hurricane Bob (Category 3)
1 /*
2  * Demonstrates the use of a programmer-defined class.
3  * Author: Java Foundations
4  */
5 public class CountFlips {
6    /*
7    * Driver: Flips a coin multiple times and counts the number of heads
8    * and tails that result.
9    */
10   public static void main (String[] args) {
11      final int FLIPS = 1000;
12      int heads = 0, tails = 0;
13      Coin myCoin = new Coin();
14      for (int count=1; count <= FLIPS; count++) {
15        myCoin.flip();
16        if (myCoin.isHeads())
17          heads++;
18        else
19          tails++;
20      }
21      System.out.println ("Number of flips: " + FLIPS);
22      System.out.println ("Number of heads: " + heads);
23      System.out.println ("Number of tails: " + tails);
24   }
25  }

1 /*
2  * Demonstrates the reuse of a programmer-defined class.
3  * Author: Java Foundations
4  */
5 public class FlipRace {
6    /*
7    * Driver: Flips two coins until one of them comes up heads three
times in a row.
8    */
9    public static void main (String[] args) {
10      final int GOAL = 3;
11      int count1 = 0, count2 = 0;
12      Coin coin1 = new Coin(), coin2 = new Coin();
13      while (count1 < GOAL || count2 < GOAL) {
14        coin1.flip();
15        coin2.flip();
16        System.out.println ("Coin 1: " + coin1 + "\tCoin 2: " + coin2);
17        count1 = (coin1.isHeads()) ? count1+1 : 0; // Increment or reset the counters
18        count2 = (coin2.isHeads()) ? count2+1 : 0;
19      }
20      if (count1 < GOAL)
21        System.out.println ("Coin 2 Wins!");
22      else
23        if (count2 < GOAL)
24          System.out.println ("Coin 1 Wins!");
25        else
26          System.out.println ("It's a TIE!");
27  }