The heart of Object-Oriented Programming
(Now it gets interesting!)

- 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>

- Assignment 1 due Thursday at 11:59pm
- Reading for next lecture is Ch. 7
  - Focus on 7.1, 7.2, and 7.6
  - Read the rest of Ch. 7 for class after next
  - Ch. 7.4 explains that `String [] args` thing in `main`!

- 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

Generally, the methods that encapsulate behaviors are called names that are **verbs**:
- Examples: get, set, calculate, convert, initiate

What are the data and methods you would define for class **Student**?

String name;
int classYear;

Let's say you have a **PictureFrame** class.
- What's the point of this class?
- What do its objects represent?
- What **data** and **methods** might the **PictureFrame** have?

What **data** and **methods** might a **Course** object have?

- 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, and private**

<table>
<thead>
<tr>
<th></th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variables</strong></td>
<td>violates encapsulation</td>
<td>enforces encapsulation</td>
</tr>
<tr>
<td><strong>Methods</strong></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**

```
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(){
    }
}
```

```
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 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);
    }
}
```

• 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
/** 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){
    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 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

```java
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.:

```
String.join(“|”, thingsToJoin)
```

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

```
name.charAt(0);
```
Write a **Hurricane** class that contains the following one constructor, three instance methods, and one class method:

```java
// 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 g
public boolean fasterWindsThan(Hurricane g)

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

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

```java
... public static int windSpeedToCategory(double windSpeed){
    if (windSpeed < 74) {
        return 0;
    } else if (windSpeed < 96) {
        return 1;
    } else if (windSpeed < 111) {
        return 2;
    } else if (windSpeed < 130) {
        return 3;
    } else if (windSpeed < 157) {
        return 4;
    } else {
        return 5;
    }
}

// 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)
/**
 * Demonstrates the use of a programmer-defined class.
 * @author Java Foundations
 */

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);
}

}