Objects and Classes

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

Announcements

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

Classes and Objects

- The basic building block on an object-oriented language is an object, simulating a real-life object
- An object has state, defined by the values of its attributes
- The attributes are defined by the data associated with the object's class
- An object also has behaviors, defined by the operations associated with it
- Operations are defined by the methods of the class

Defining a class

- 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
Defining a class

- 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

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

---

Defining a Die Class

- Consider a six-sided **die** (singular of dice)
  - What should its **state** be?
  - What should its primary **behavior** be?
- We represent a die in Java by designing a class called **Die** that **models** its state and behavior
- We want to design the **Die** class with other data and methods to make it a versatile and **reusable** resource

![Die class diagram](image)

---

Defining a PictureFrame class

- 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?

![PictureFrame diagram](image)

---

Wrapper Classes in Java

- Not all variables in Java are objects
  - Some are **primitive data types** (but have related objects)
  - All primitive data types have a corresponding **Wrapper Class**

<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>
</tbody>
</table>
**Encapsulation**

- 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

**Visibility Modifiers**

- 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 encapsulation</td>
<td>enforces encapsulation</td>
</tr>
<tr>
<td>Methods</td>
<td>provides services to clients</td>
<td>supports other methods in class</td>
</tr>
</tbody>
</table>

**Anatomy of a Class**

- 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 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);
    }
}
```
Constructors

- 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
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.
        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() {
        return faceValue;
    }
}
```

UML Diagrams

- A UML class diagram showing the classes involved in the SnakeEyes program:
Control Flow

- Understanding the control flow is essential to debugging!

```java
class SnakeEyes {
    public int penNumber;
}

class Die {
    public int roll() {
        return 1; // Example roll value
    }
}
```

- An instance variable is specific to that instance of the object (there can be many instances of an object)
- A static variable belongs to the class (there is only one copy)
- A static method (or class method) effects the entire class

```java
public static int totalSheep = 0;

public static Sheep prioritPen(Sheep a, Sheep b) {
    // Example method implementation
    return a; // Replace with actual prioritization logic
}
```

More on Control Flow

- 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!

```java
public void main() {
    Die die = new Die();
    die.roll();
}
```

Instance vs. Static (See Ch. 5.5)

- An instance variable is specific to that instance of the object (there can be many instances of an object)
- A static variable belongs to the class (there is only one copy)
- A static method (or class method) effects the entire class

```java
public int getPenNumber() {
    return penNumber;
}
```

```java
Sheep pPen = prioritPen(dave, clara);
```

```java
// creates dave, initializes penNumber, increments totalSheep
Sheep dave = new Sheep();

// instance call, public int getPenNumber(), returns 4
int pen = dave.getPenNumber();

// static call, public static Sheep prioritPen(Sheep a, Sheep b)
Sheep pPen = prioritPen(dave, clara);
```
Write a Grade class that contains the following one constructor, three instance methods, and one class method:

```java
// 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)
```

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

```java
// 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));
}
```

Then executing the Grade application would produce the following output:

```
false
Letter grade: B-
Score: 82.1
```

---

Reusing Classes, e.g. Coin.java

```java
/**
 * Demonstrates the use of a programmer-defined class.
 * Library 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 = 100;
        int heads = 0, tails = 0;
        Coin myCoin = new Coin();
        for (int count=0; count <= FLIPS; count++) {
            myCoin.flip();
            if (myCoin.isHeads())
                heads++;
            else
tails++;
        }
        System.out.println("Number of heads: " + heads);
        System.out.println("Number of tails: " + tails);n
    }
}
```

---

CountFlips.java uses Coin.java
FlipRace.java also uses Coin.java

```java
* Demonstrates the reuse of a programmer-defined class.
* Author: Java Foundations

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 + "\nCoin 2: " + coin2);
            count1 = (coin1.isHeads()) ? count1 + 1 : 0; // Increment or reset the counters
            count2 = (coin2.isHeads()) ? count2 + 1 : 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!");
    }
}
```