Components of a Class

- A class can contain data declarations and method declarations
- The values of the data define the state of an object created from the class
- The functionality of the methods define the behaviors of the object

5.1 – Identifying Classes (& Objects)

- A class represents a group of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns. Examples: Coin, Student, Message
- Sometimes it is challenging to decide whether something should be represented as a class
  - Should an employee’s address be represented as a set of instance variables or as an Address object?
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

5.2 – Anatomy of a Class

- We want to define classes with the proper amount of detail
  - It may be unnecessary to create separate classes for each type of appliance in a house
  - It may be sufficient to define a more general Appliance class with appropriate instance data
  - It all depends on the details of the problem being solved
- Part of identifying the classes we need is the process of assigning responsibilities to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- We generally use verbs for the names of methods
  - In early stages it is not necessary to determine every method of every class begin with primary responsibilities and evolve the design
- 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
- Any given program will not necessarily use all aspects of a given class
- Let’s see how we would use someone’s class of the Die to play snakeEyes
```
5.2 – SnakeEyes.java
Demonstrates the use of a programmer-defined class.

public class SnakeEyes
{
    // Creates two Die objects and rolls 500 times,
    // counting the number of snake eyes (1) that occur.
    public static void main (String[] args) {
        final int ROLLS = 500;
        int num1, num2, count = 0;
        Die die1 = new Die();
        Die die2 = new Die();
        for (int roll=1; roll <= ROLLS; roll++) {
            num1 = die1.roll();
            num2 = die2.roll();
            if (num1 == 1 && num2 == 1) // check for snake eyes
                count++;
        }
        System.out.println ("Number of rolls: " + ROLLS);
        System.out.println ("Number of snake eyes: " + count);
        System.out.println ("Ratio: " + (float) count / ROLLS);
    }
}

5.2 – Die.java
Represents one die with faces showing values between 1 and 6.

public class Die
{
    private final int MAX = 6; // maximum face value
    private int faceValue; // current value showing on the die
    public Die()
    { faceValue = 1; }
    public int roll()
    { faceValue = (int)(Math.random() * MAX) + 1;
        return faceValue;
    }
}

// Face value mutator.
// If the specified value is not valid face value does not change
// (Q: Is it a good idea to have it public?)
public void setFaceValue (int value)
{
    if (value > 0 && value <= MAX)
        faceValue = value;
}

// Face value accessor.
public int getFaceValue()
{ return faceValue; }

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

5.2 - UML Class Diagrams

A “Unified Modeling Language” class diagram for SnakeEyes:

```
Note: UML does not explicitly mentioning a constructor.
```
Reusing Classes, e.g. a Coin.java

// Coin.java
// Represents a coin with two sides that can be flipped.
//********************************************************************
public class Coin
{
    // private instance variables, no need to know them
    public Coin () { // Constructor creates a new Coin.
    }
    public void flip () { // Flips this coin.
    }
    // Returns true if the current face of this coin is heads.
    public boolean isHeads () {
    }
    // Returns the current face of this coin as a string.
    public String toString () {
}
}

CountFlips.java uses Coin.java

// CountFlips.java
One program that uses Coin

public class CountFlips
{
    // Flips a coin multiple times and counts the number of heads & tails.
    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);
    }
}

FlipRace.java also uses Coin.java

// FlipRace.java
Another program that uses Coin

public class FlipRace
{  // Flips two coins until one of them comes up heads 3 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 + ", Coin 2: " + coin2);
            // Increment or reset the counters
            count1 = (coin1.isHeads()) ? count1 + 1 : 0;
            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!");
    }
}

5.3 – Encapsulation

An encapsulated object can be thought of as a black box – its inner workings are hidden from the client.

The client invokes the interface methods of the object, which manages the instance data.

Client

Methods

Data
5.3 - Visibility Modifiers

- Java does encapsulation through the use of visibility modifiers
- A modifier specifies particular characteristics of a method or data
- Java has three visibility modifiers: public, protected, and private
- The protected modifier involves inheritance (later on this)

<table>
<thead>
<tr>
<th></th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Violate encapsulation</td>
<td>Enforce encapsulation</td>
</tr>
<tr>
<td>Methods</td>
<td>Provide services to clients</td>
<td>Support other methods in the class</td>
</tr>
</tbody>
</table>

5.4 - Method Control Flow

- If the called method is in the same class, only the method name is needed
- The called method is often part of another class or object
- Understanding the control flow is essential to debug!

5.4 - Driver Programs

- A driver program drives the use of other, more interesting parts of a program
- A TestCoin class could contain a main method that drives the use of the Coin class, exercising its methods
- I recommend to have a driver inside the class definition as the class’ main() method.
  That way the testing becomes part of the development.
- In any case, you should be able to write the driver/main() before writing the class itself!

Constructors have no return type

- 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
- Constructors are often overloaded.