Using Objects

Continued...

Declaring Object Types

- A variable holds either a primitive type or a *reference* to an object
- A class name can be used as a type to declare an *object reference variable*

```
String title;
```

- No object is created with this declaration
- The object itself must be created separately
- After its creation, an object reference variable holds the address of an object stored in the main memory of the computer
- Before its creation, it holds *null*

Invoking Methods

- We've seen that once an object has been instantiated, we can use the *dot operator* to invoke its methods

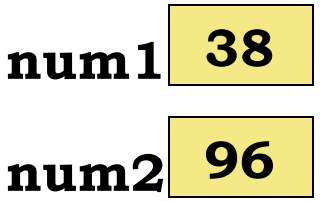
```
count = title.length()
```

- A method may *return a value*, which can be used in an assignment or expression
- A method invocation can be thought of as asking an object to perform a service

Assignment Revisited

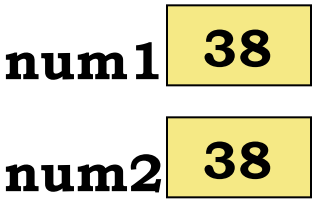
- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

Before:




```
num2 = num1;
```

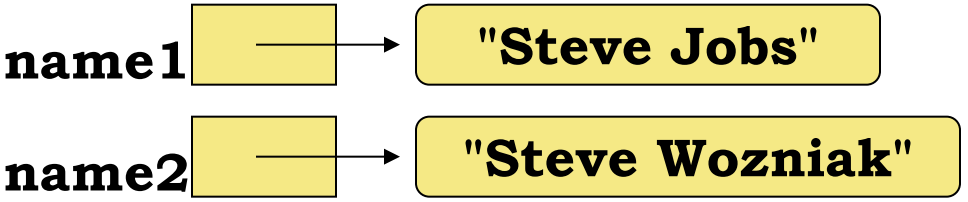
After:



Assignment Revisited

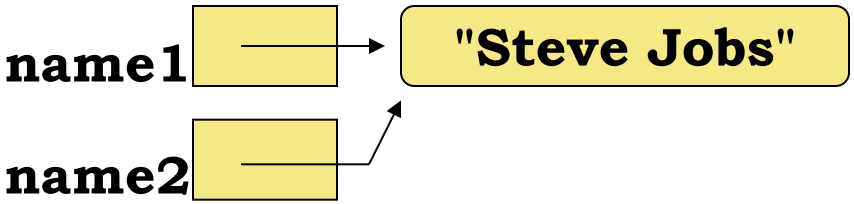
- For object references, the address is copied:

Before:




```
name2 = name1;
```

After:



Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- That creates an interesting situation: one object can be accessed using multiple reference variables
- Aliases can be useful, but should be managed carefully
- Changing an object through one reference changes it for all of its aliases, because there is really only one object

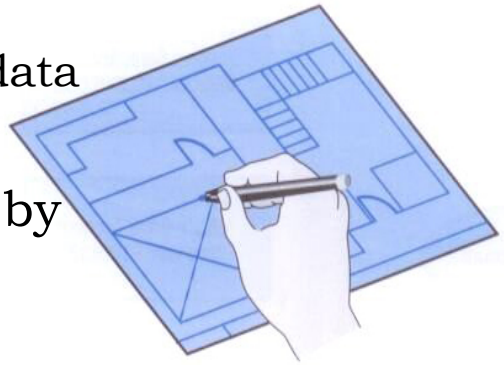
Classes and Objects

The heart of Object-Oriented Programming

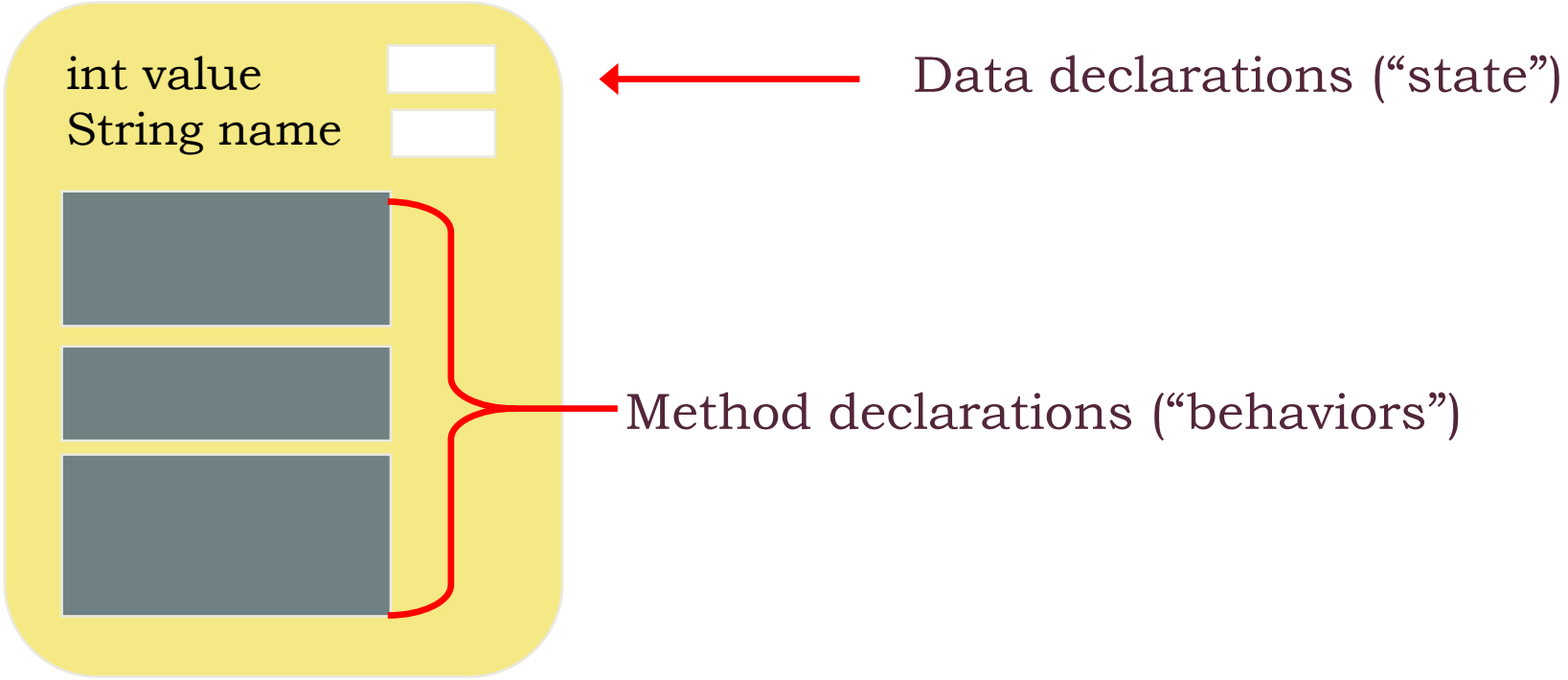
(Now it gets interesting!)

Objects and Classes

- The basic building block on an object-oriented language is an **object**, simulating a real-life object
- A **class** is like a blueprint from which you can create many objects that may have different characteristics
- 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 has **behaviors**, defined by the operations associated with it
  - Behaviors (operations) are implemented by the **methods** of the class

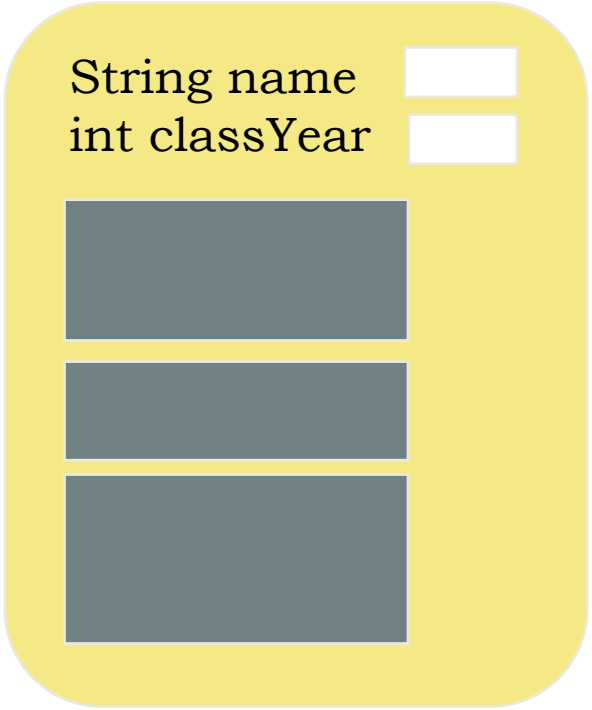


Defining a Class



- 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 using names that are **singular nouns**:
  - Examples: **Coin, Student, Classroom**
- Generally, the methods that encapsulate behaviors are called using names that are **verbs**:
  - Examples: **flip, register, assign, get, set**
- What are the data and methods you would define for class **Coin**?

What is the rule of thumb for finding classes?

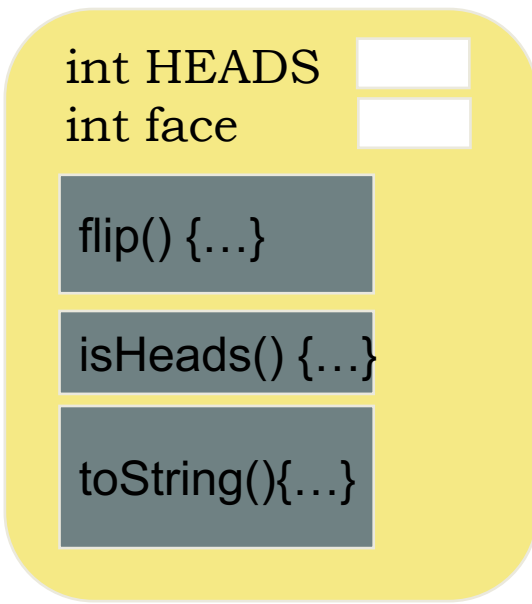
Answer: Look for nouns in the problem description.

Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?

Answer: Yes (`ChessBoard`) and no (`MovePiece`).

File: Coin.java

```java
/**
* Represents a coin with two sides that can be flipped.
* @author 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() { ... }
}
```



Self Check

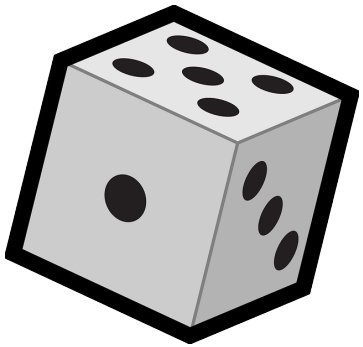
We have used `System.out` as a opaque box to cause output to appear on the screen. Who designed and implemented `System.out`?

Answer: The programmers who designed and implemented the Java library.

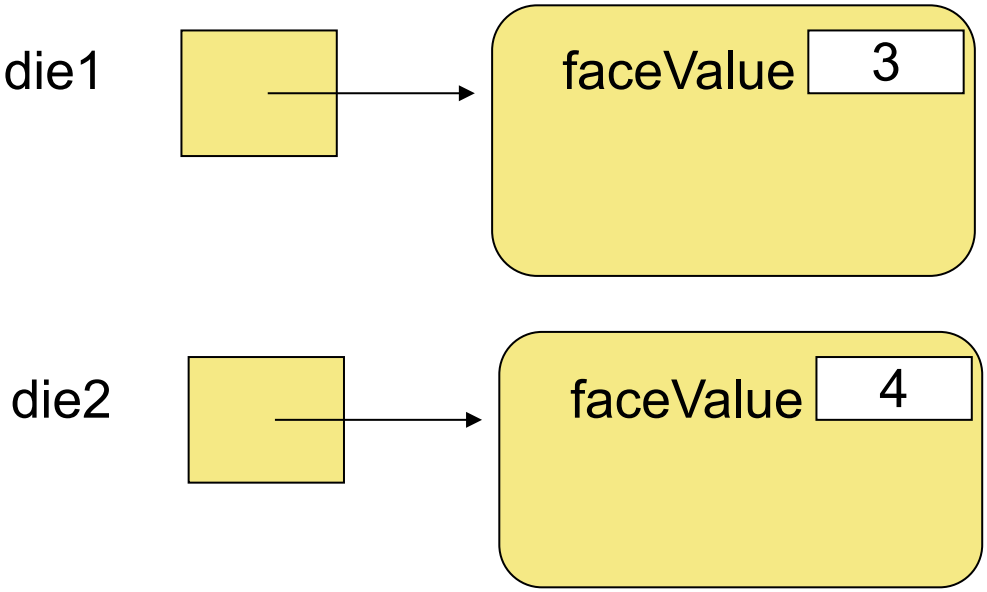
Designing a Class

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

Defining a Die Class

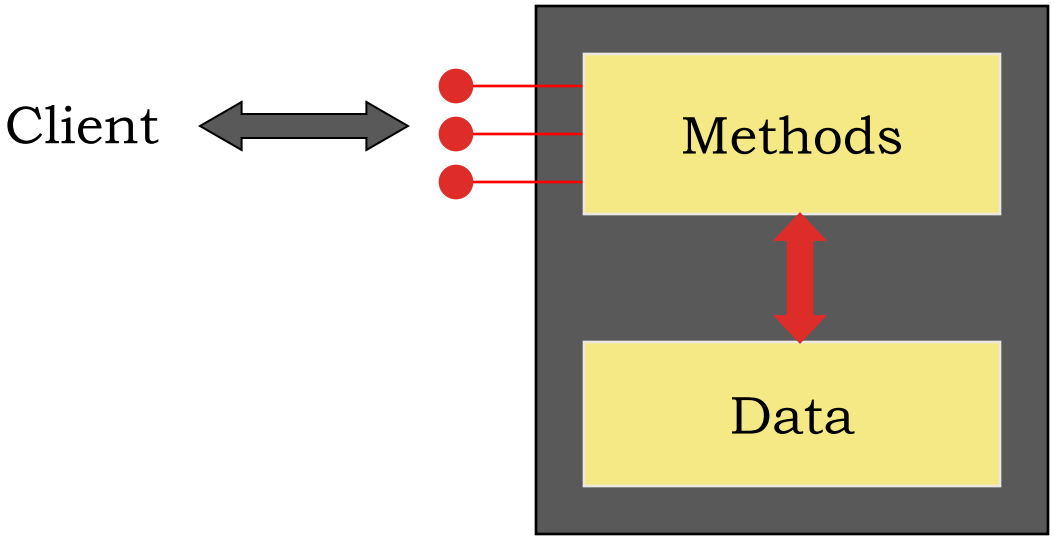


- Consider a six-sided **die** (singular of dice)
  - What should its **state** be?
  - What should its primary **behavior** be?
- We represent a die in Java by designing a class called **Die** that **models** its state and behavior
- We want to design the **Die** class with other data and methods to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



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 *non-transparent 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

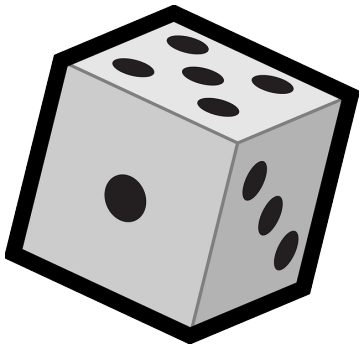


Visibility Modifiers

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

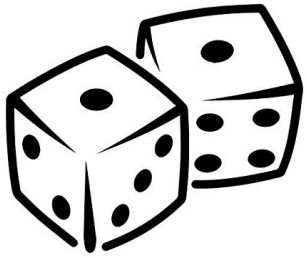
	public	**private**
Variables	violates encapsulation	enforces encapsulation
Methods	provides services to clients	supports other methods in class

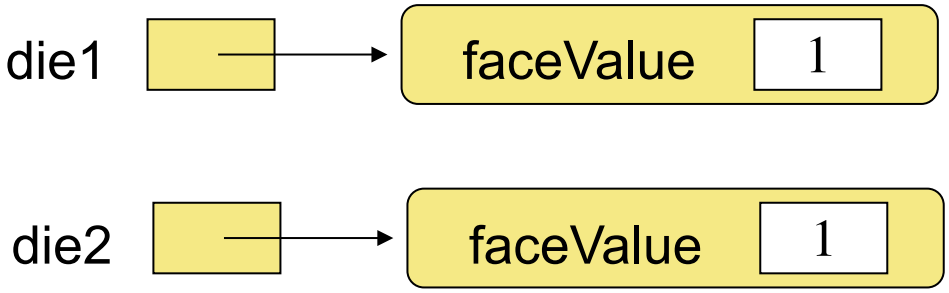
Anatomy of a Class



- 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**, that is, write a **client** for Die





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

Constructors

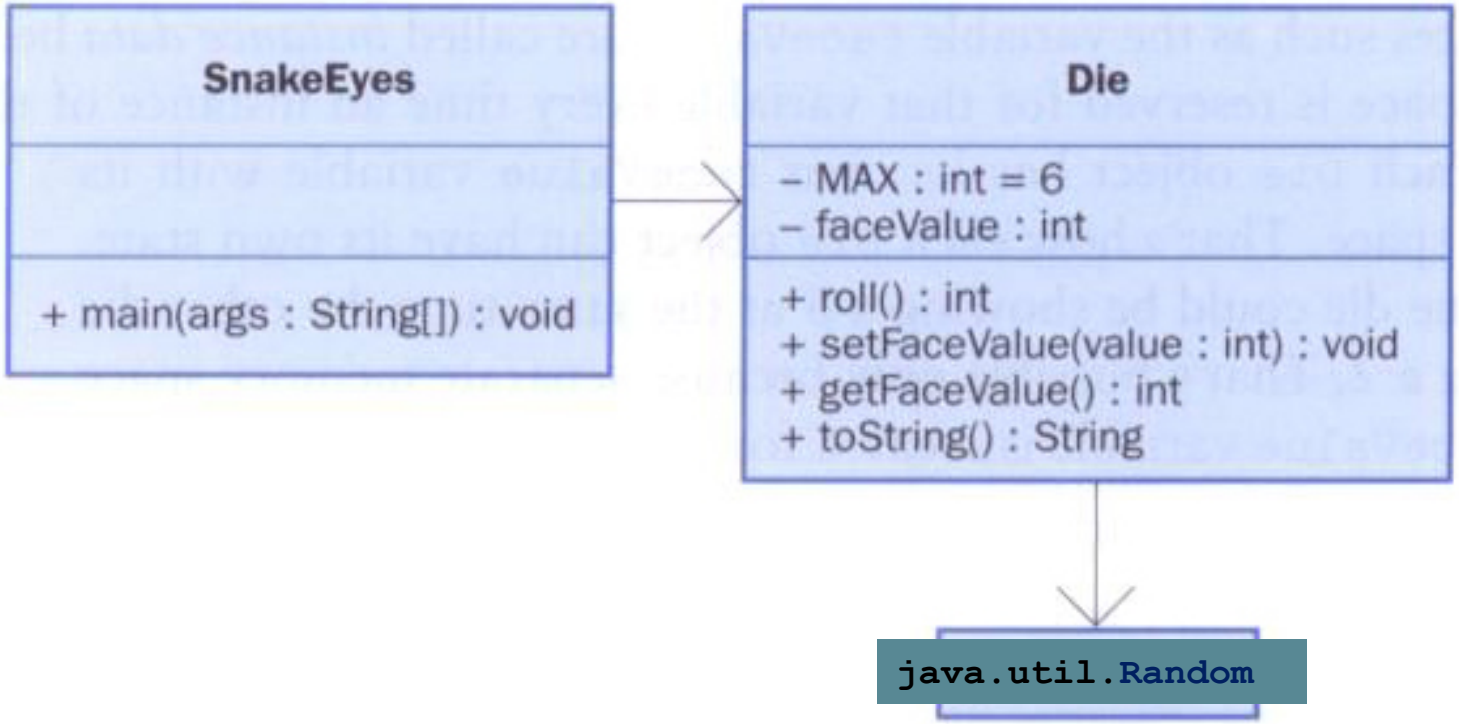
- A **constructor** is a special method which builds a new instance of the class
- Note that a constructor has **no return type** in the method header, **not even void**
- A common error is to put a return type on a constructor, which makes it a "regular" method that happens to have the same name as the class
- The programmer does not have to define a constructor for a class:
  - Each class has a ***default*** *constructor* that accepts no parameters

```
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(){
      }
```

```
    /**
     * 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){

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

    }
    /**
     * @return string representation of this die
     */
    public String toString() {

    }
}
```

UML Diagrams

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



Wrapper Classes in Java

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

Primitive	Object
int	Integer
long	Long
float	Float
double	Double
char	Char
boolean	Boolean

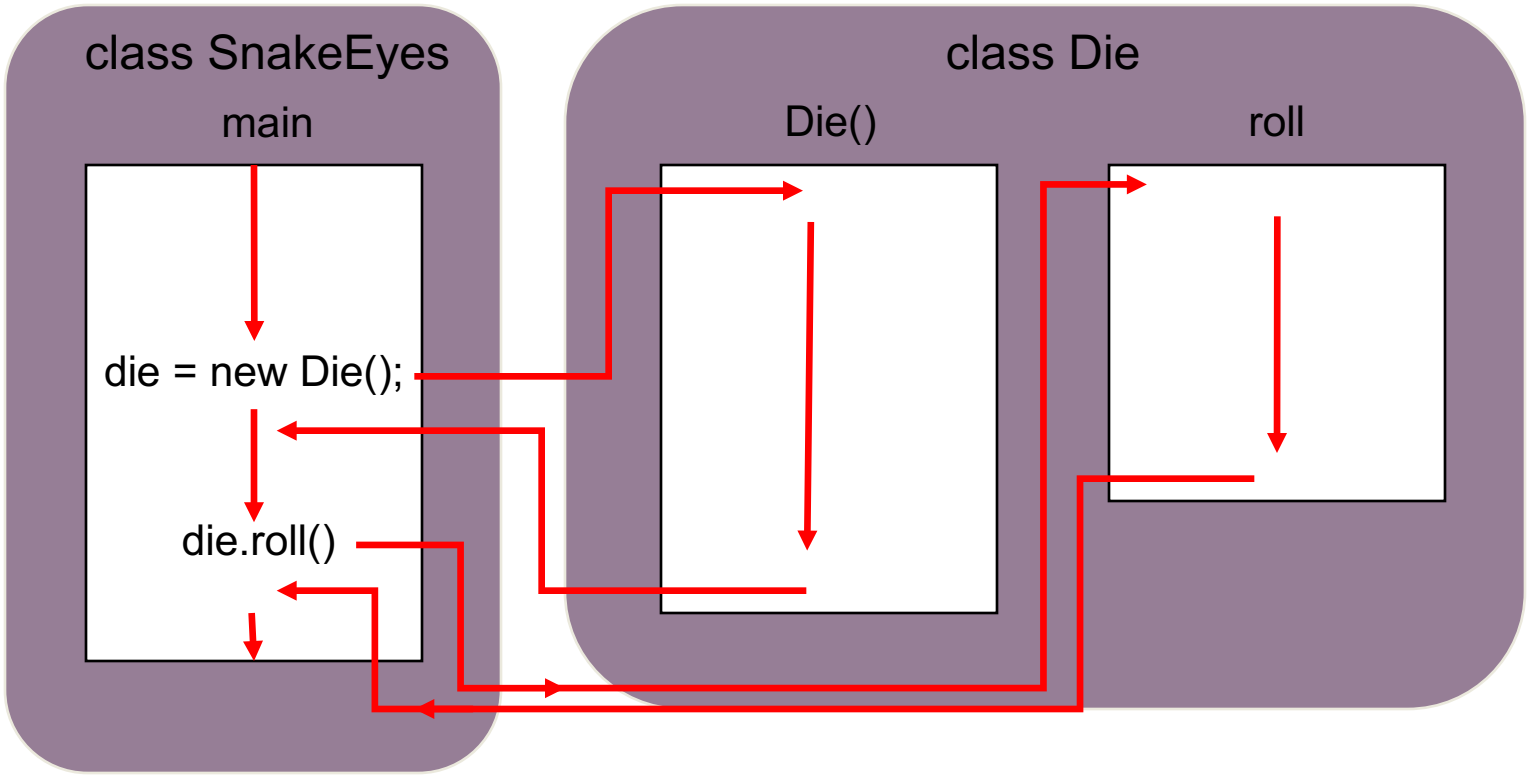
Control Flow

The order in which programs are executed.

It all starts with the main() method...

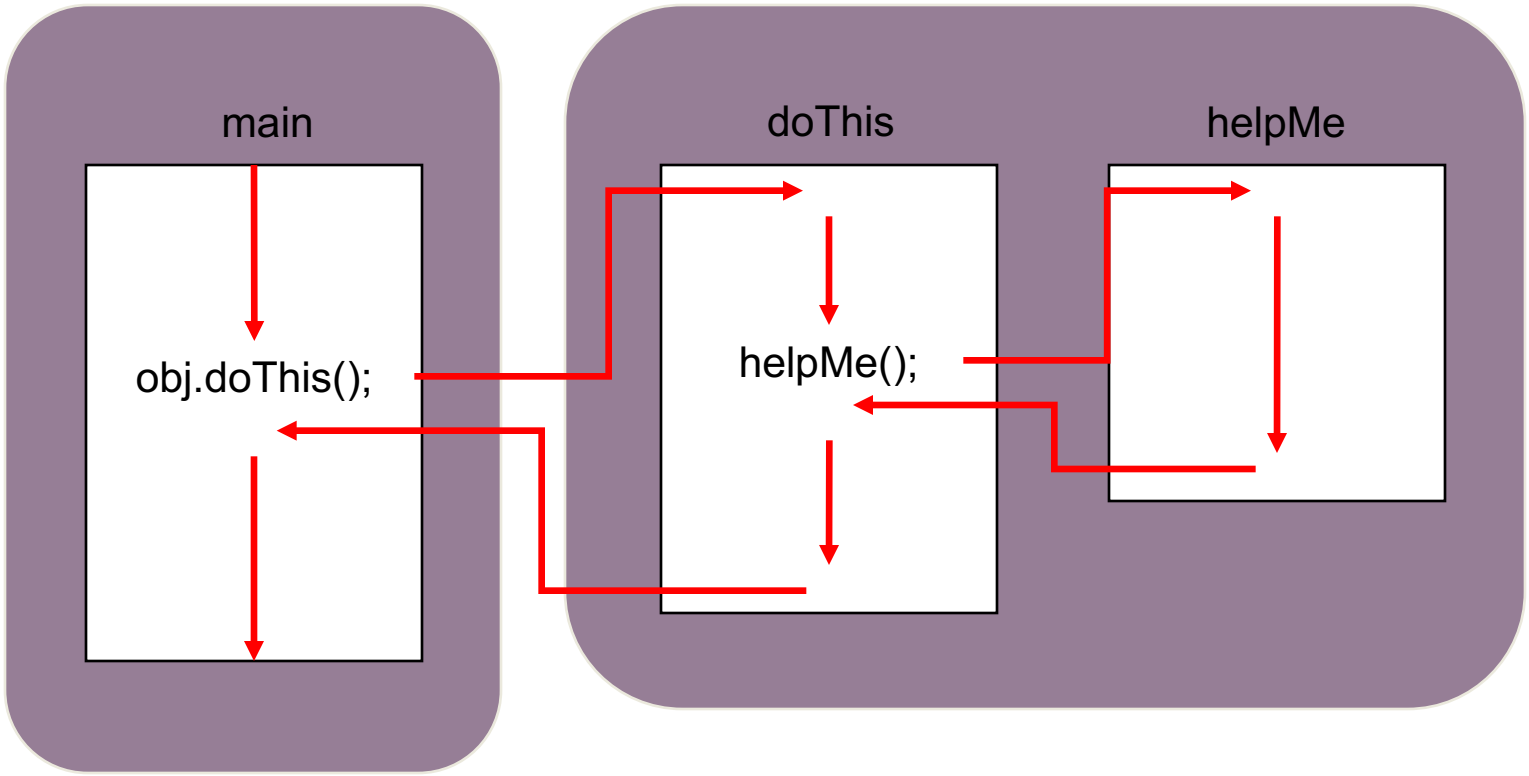
Control Flow

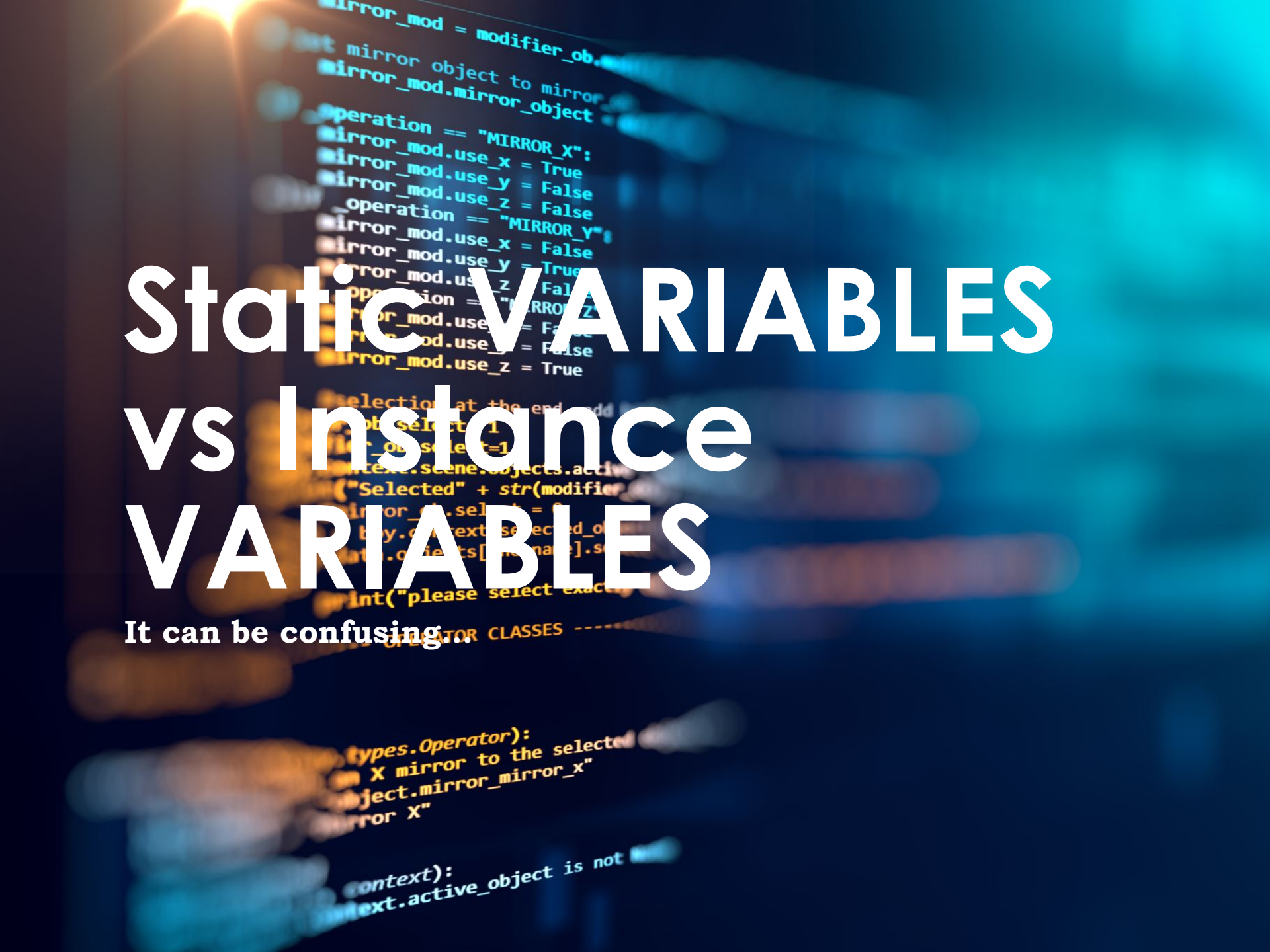
- Understanding the control flow is essential to **debugging**!



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!**




Static VARIABLES
vs Instance
VARIABLES
It can be confusing...

Static Variables

- A static variable belongs to the class, not to any object of the class.
- To assign bank account numbers sequentially

  Have a single value of `lastAssignedNumber` that is a property of the class, not any object of the class.

- Declare it using the `static` reserved word

```
public class BankAccount
{
   private double balance;
   private int accountNumber;
   private static int lastAssignedNumber = 1000;

   public BankAccount()
   {
      lastAssignedNumber++;
      accountNumber = lastAssignedNumber;
   }
   . . .
}
```

Static Variables

- Every `BankAccount` object has its own `balance` and `accountNumber` instance variables
- All objects share a single copy of the `lastAssignedNumber` variable

  That variable is stored in a separate location, outside any BankAccount objects

- Static variables should always be declared as private,

  This ensures that methods of other classes do not change their values

- `static` constants may be either private or public

```
public class BankAccount
{
   public static final double OVERDRAFT_FEE = 29.95;
   . . .
}
```

- Methods from any class can refer to the constant as `BankAccount.OVERDRAFT_FEE`.

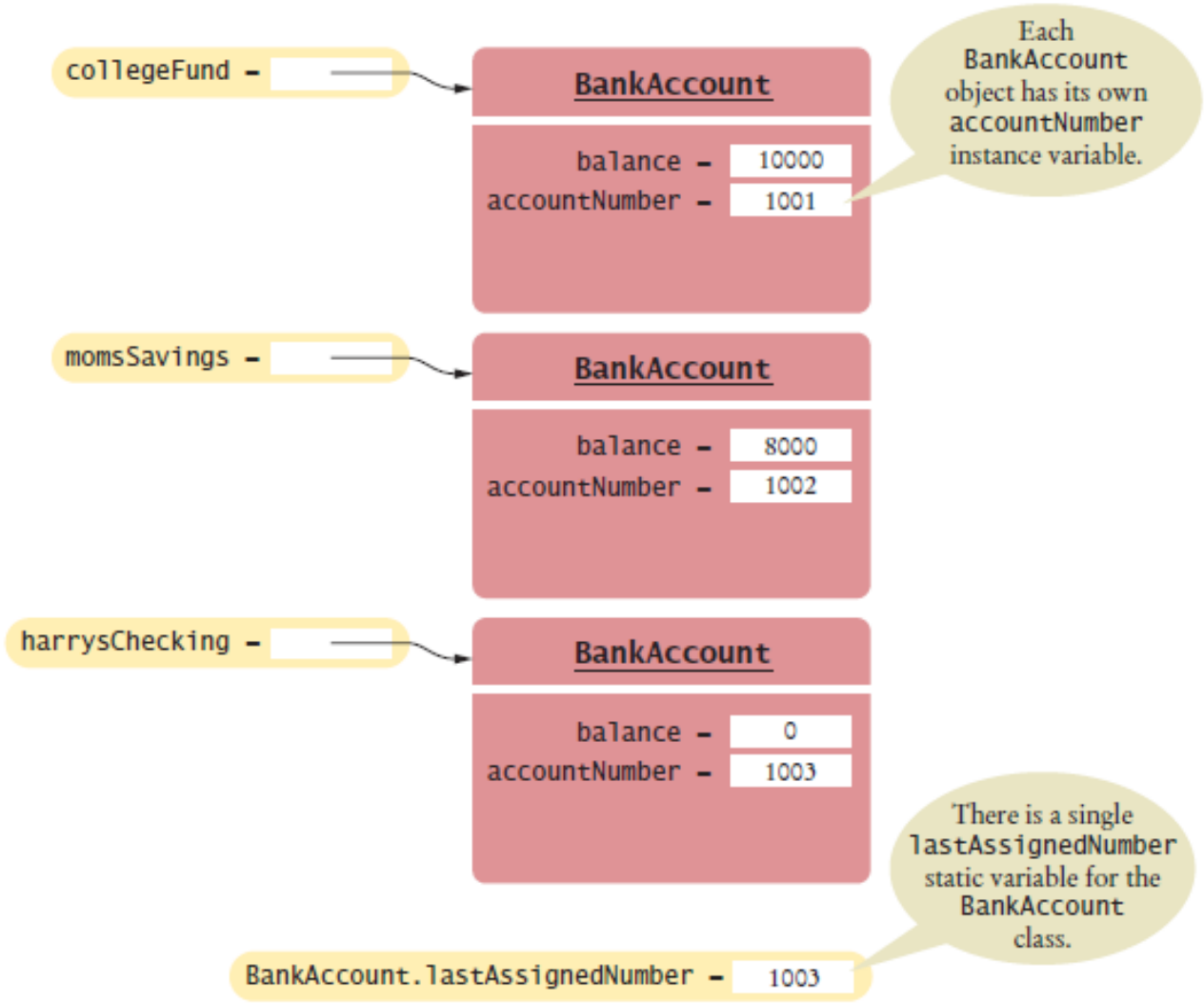


Figure 5 A Static Variable and Instance Variables

Static methods vs Instance methods

It can be confusing...

Static Methods

- Sometimes a class defines methods that are not invoked on an object. Called a **static method**

  Example: `sqrt` method of `Math` class

- if `x` is a number, then the call `x.sqrt()` is not legal

  `Math` class provides a static method: invoked as `Math.sqrt(x)`

  No object of the `Math` class is constructed.

  The `Math` qualifier simply tells the compiler where to find the `sqrt` method.

When calling such a method, supply the name of the class containing it:

```
double tax = Financial.percentOf(taxRate, total);
```

The `main` method is always `static`.

When the program starts, there aren't any objects.

Therefore, the first method of a program must be a static method.

Programming Tip: Minimize the Use of Static Methods

Grade Class driver (a client)

Suppose we execute the following **main()** method:

```
// 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,emily));
}
```

Instance vs Static

We need to write a **Grade** class that contains (at least) a constructor, and a few of methods.
You may think that **isHigherThan** and **max** do essentially the same thing (a comparison of scores) but they are defined differently:

```
// Constructor creating a Grade represented with
// a letter and a number
public Grade(String letterGrade, double numericalGrade)


// Compares this Grade's score to another Grade g and
// returns true if this Grade is higher than Grade g
public boolean isHigherThan(Grade g)


// Compares the scores of two grades and
// returns the maximum of the two Grade objects
public static Grade max(Grade g1, Grade g2)
```

Reusing Classes

The power of Object-Oriented Programming

Reusing Classes, e.g. Coin.java

```java
/**
* Represents a coin with two sides that can be flipped.
* @author 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() { ... }
}
```

CountFlips.java uses Coin.java

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

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 + "\tCoin 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!");
   }
```