Writing Classes

- The programs we've written in previous examples have used classes defined in the Java standard class library

- Now we will begin to design programs that rely on classes that we write ourselves

- The class that contains the `main` method is just the starting point of a program

- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality

A sample problem

- **Write a method that will throw 2 Dice with varying number of sides a specified amount of times and reports how many times we got a snake eyes (both dice showing 1)**

- **For example numSnakeEyes(6, 13, 100) should return the number of snake eyes after throwing a 6 sided Die and 13 sided Die 100 times.**

Structured Die

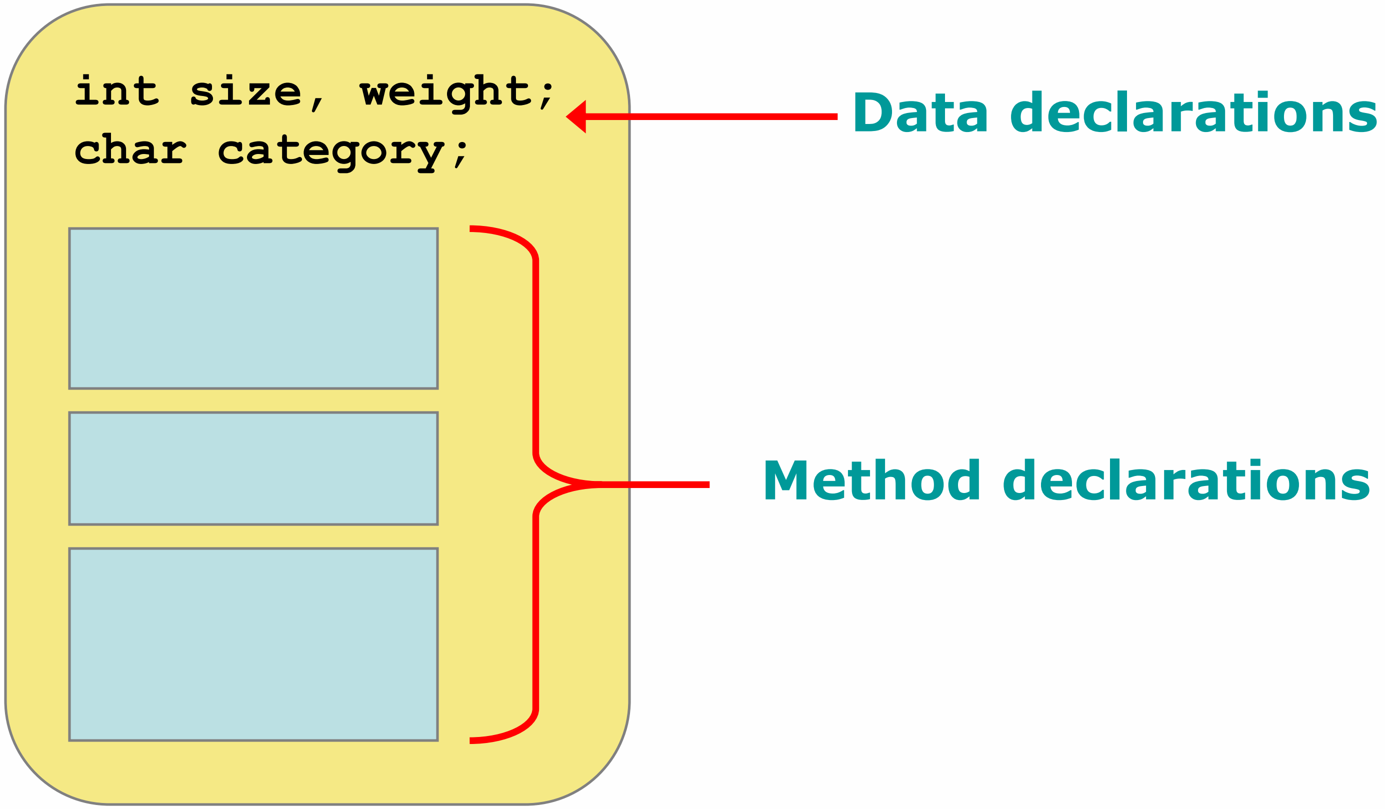
```
static Random rand = new Random();

static int roll(int numSides) {
    return 1 + rand.nextInt(numSides);
}

static int numSnakeEyes(int sides1, int sides2, int numThrows) {
    int count = 0;
    for(int i = 0; i < numThrows; i++) {
        int face1 = roll(sides1);
        int face2 = roll(sides2);
        if (face1 == 1 && face2 == 1)
            count++;
    }

    return count;
}
```

Object Oriented Approach

- **In OOP, we first focus on the main actors, not how things are done.**
- **The main actors here are Die objects. We need to define a Die class that captures the *state* and *behavior of a Die.***
- **We can then instantiate as many die objects as we need for any particular program**

Classes

- **A class can contain data declarations and method declarations**



Data and Methods

- **For our `Die` class, we might declare an integer that represents the current value showing on the face, and another to keep the number of faces**
- **One of the methods would "roll" the die by setting that value to a random number between one and number of faces, we also need methods to give us information about our object.**

Classes

- **We'll 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**

```
public class Die {
    private int numFaces;  // maximum face value
    private int faceValue;  // current value showing on the die

    //  Constructor: Sets the initial face value.
    public Die(int _numFaces)  {
        numFaces = _numFaces;
        roll();
    }

    //  Rolls the die
    public void roll() {
        faceValue = (int)(Math.random() * numFaces) + 1;
    }

    //  Face value setter/mutator.
    public void setFaceValue (int value)  {
        if (value <= numFaces)
            faceValue = value;
    }
```

Die Cont.

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

//  Face value getter/accessor.
public int getNumFaces() {
        return numFaces;
}

//  Returns a string representation of this die.
public String toString() {
        return “number of Faces “ + numFaces +
                “current face value “ + faceValue);
}
}
```

The new Version

```
static int numSnakeEyes(int sides1, int sides2, int numThrows) {
    Die die1 = new Die(sides1);
    Die die2 = new Die(sides2);

    int count = 0;
    for(int i = 0; i < numThrows; i++) {
        die1.roll();
        die2.roll();
        if (die1.getFaceValue == 1 && die2.getFaceValue == 1 )
            count++;
    }

    return count;
}
```

Using Die class in general

```
Die die1, die2;
int sum;

die1 = new Die(7);
die2 = new Die(34);

die1.roll();
die2.roll();
System.out.println ("Die One: " + die1 + ", Die Two: " + die2);

die1.roll();
die2.setFaceValue(4);
System.out.println ("Die One: " + die1 + ", Die Two: " + die2);

sum = die1.getFaceValue() + die2.getFaceValue();
System.out.println ("Sum: " + sum);

sum = die1.roll() + die2.roll();
System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
System.out.println ("New sum: " + sum);
```

The toString Method

- **All classes that represent objects should define a `toString` method**
- **The `toString` method returns a character string that represents the object in some way**
- **It is called automatically when an object is concatenated to a string or when it is passed to the `println` method**

Data Scope

- The *scope* of data is the area in a program in which that data can be referenced (used)
- Data declared at the class level can be referenced by all methods in that class
- Data declared within a method can be used only in that method
- Data declared within a method is called *local data*

Local and Class scope

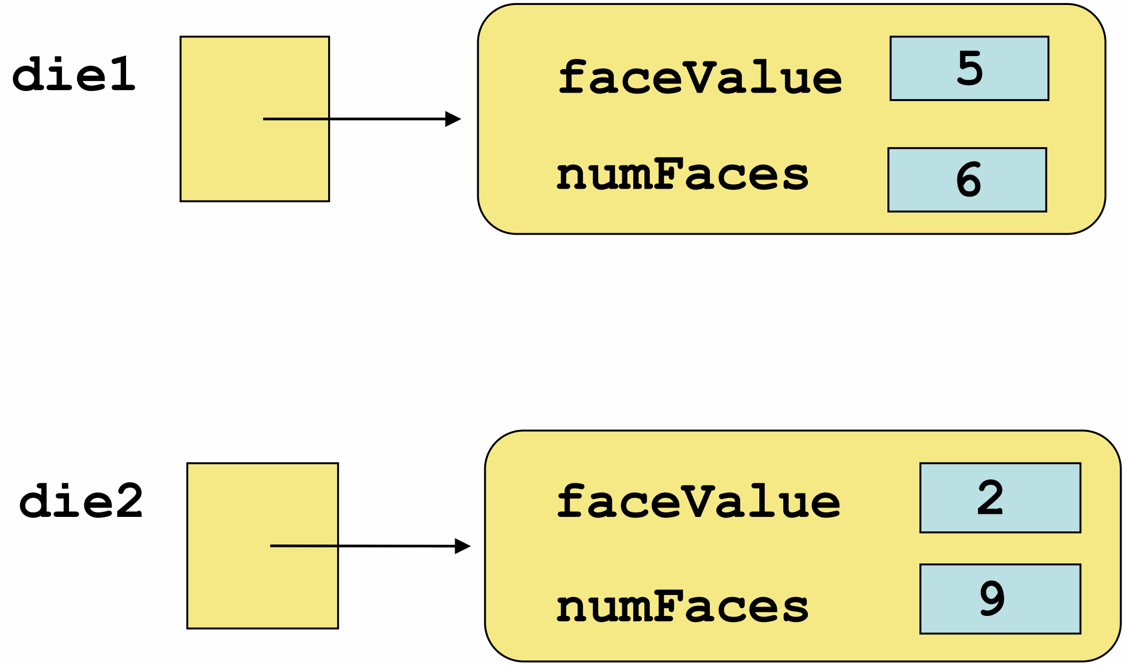
```
public class X{
    private int a; // a has class scope, can be seen from
                       // anywhere inside the class
    ….
    public void m() {
            a=5; // no problem
            int b = 0; // b is declared inside the method, local scope
    …..
    } // here variable b is destroyed, no one will remember him

    public void m2() {
            a=3; // ok
            b = 4; // who is b? compiler will issue an error
    }
```

Instance Data

- The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the method definitions, but each object has its own data space
- That's the only way two objects can have different states

Instance Data

- We can depict the two `Die` objects from the `RollingDice` program as follows:



Each object maintains its own `faceValue` and `numFaces` variable, and thus its own state

Coin Example

- Write a program that will flip a coin 1000 times and report the number of heads and tails

- Flips two coins until one of them comes up heads three times in a row, and report the winner.

Coin Class

```
public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;

   private int face;

  public Coin () {
     flip();
  }
  public void flip () {
     face = (int) (Math.random() * 2);
  }
```

```
public boolean isHeads () {
      return (face == HEADS);
   }
  public String toString() {
      String faceName;
      if (face == HEADS)
         faceName = "Heads";
      else
         faceName = "Tails";
      return faceName;
   }
}
```

Count Flips

```
final int NUM_FLIPS = 1000;
int heads = 0, tails = 0;
Coin myCoin = new Coin();  // instantiate the Coin object

for (int count=1; count <= NUM_FLIPS; count++)
{
   myCoin.flip();

   if (myCoin.isHeads())
      heads++;
   else
      tails++;
}

System.out.println ("The number flips: " + NUM_FLIPS);
System.out.println ("The number of heads: " + heads);
System.out.println ("The number of tails: " + tails);
```

FlipRace

```
//  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;

     // Create two separate coin objects
     Coin coin1 = new Coin();
     Coin coin2 = new Coin();

     while (count1 < GOAL && count2 < GOAL)
     {
        coin1.flip();
        coin2.flip();

        // Print the flip results (uses Coin's toString
     method)
        System.out.print ("Coin 1: " + coin1);
        System.out.println ("   Coin 2: " + coin2);

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

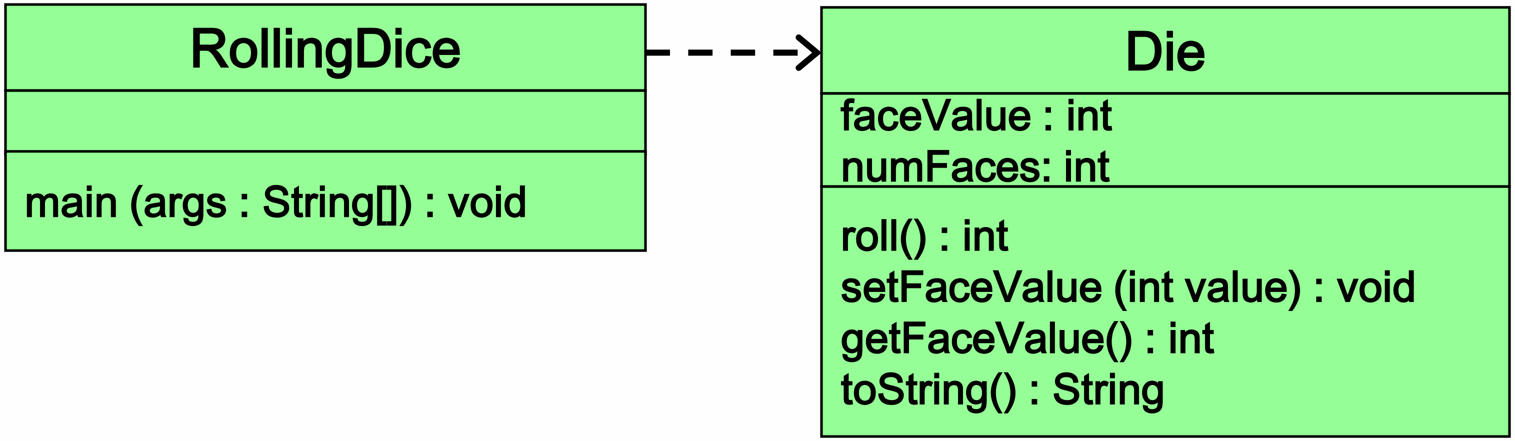
```
// Determine the winner
     if (count1 < GOAL)
        System.out.println ("Coin 2 Wins!");
     else
        if (count2 < GOAL)
           System.out.println ("Coin 1 Wins!")
        else
           System.out.println ("It's a TIE!");
}
```

UML Diagrams

- **UML stands for the *Unified Modeling Language***
- ***UML diagrams* show relationships among classes and objects**
- **A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)**
- **Lines between classes represent *associations***
- **A dotted arrow shows that one class *uses* the other (calls its methods)**

UML Class Diagrams

- **A UML class diagram for the `RollingDice` program:**



Encapsulation

- **We can take one of two views of an object:**
  - **internal - the details of the variables and methods of the class that defines it**
  - **external - the services that an object provides and how the object interacts with the rest of the system**
- **From the external view, an object is an *encapsulated* entity, providing a set of specific services**
- **These services define the *interface* to the object**

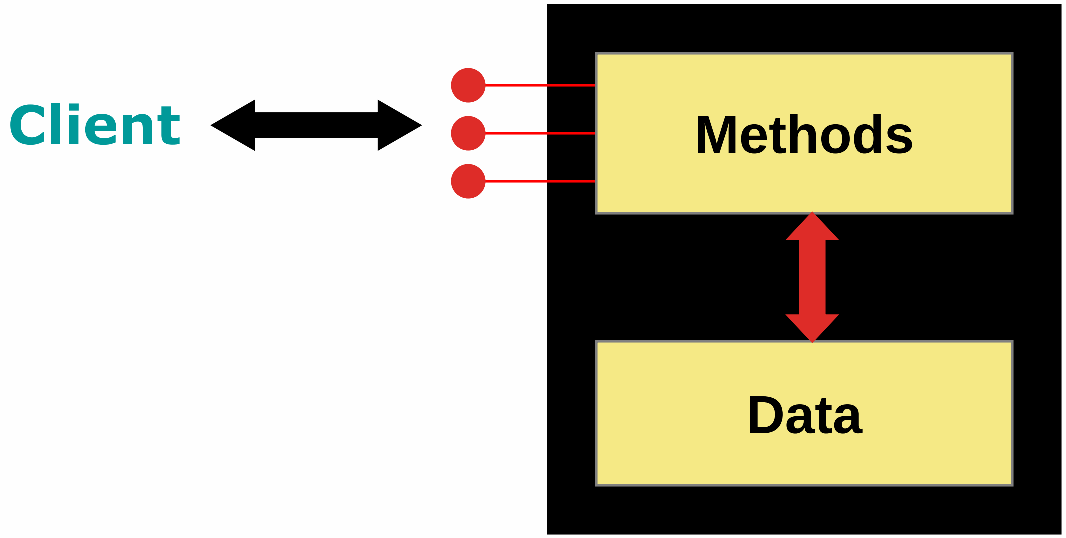
Encapsulation

- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object's state (its variables) should be made by that object's methods
- We should make it difficult, if not impossible, for a client to access an object's variables directly
- That is, an object should be *self-governing*

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



Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data
- We've used the `final` modifier to define constants
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility Modifiers

- Members of a class that are declared with ***public visibility*** can be referenced anywhere
- Members of a class that are declared with ***private visibility*** can be referenced only within that class
- Members declared without a visibility modifier have ***default visibility*** and can be referenced by any class in the same package

```
package s.t;
public class A {
   private int pv;
   int d;
   public int pb;

   m(…) {
          pv = 0; // OK
          d = 0; // OK
          pb = 0; // OK
```

```
package s.t;
public class B {
   …
   m(…) {
          A a = new A(..);
          a.pv = 0; // ERROR
          a.d = 0; // OK
          a.pb = 0; // OK
```

```
package s.u;
public class C {
   …
   m(…) {
          A a = new A(..);
          a.pv = 0; // ERROR
          a.d = 0; // ERROR
          a.pb = 0; // OK
```

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to “reach in” and modify the values directly
- Therefore instance variables should not be declared with public visibility
- It is acceptable to give a constant public visibility, which allows it to be used outside of the class
- Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed

Visibility Modifiers

- **Methods that provide the object's services are declared with public visibility so that they can be invoked by clients**

- **Public methods are also called *service methods***

- **A method created simply to assist a service method is called a *support method***

- **Since a support method is not intended to be called by a client, it should not be declared with public visibility**

Visibility Modifiers

	`public`	`private`
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Accessors and Mutators

- **Because instance data is private, a class usually provides services to access and modify data values**
- **An *accessor method* returns the current value of a variable**
- **A *mutator method* changes the value of a variable**
- **The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value**
- **They are sometimes called “getters” and “setters”**

Mutator Restrictions

- **The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state**

- **A mutator is often designed so that the values of variables can be set only within particular limits**

- **For example, the `setFaceValue` mutator of the `Die` class restricts the value to the valid range (1 to `numFaces`)**