CSE 1201
Object Oriented Programming

Writing Classes

Acknowledgement

- **For preparing the slides I took materials from the following sources**
  - **Course Slides of Dr. Tagrul Dayar, Bilkent University**
  - **Java book "*Java Software Solutions*" by Lewis & Loftus.**

Outline

- **Anatomy of a Class**
- **Anatomy of a Method**

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.**

- **We will first show a structured approach**

Structured Die

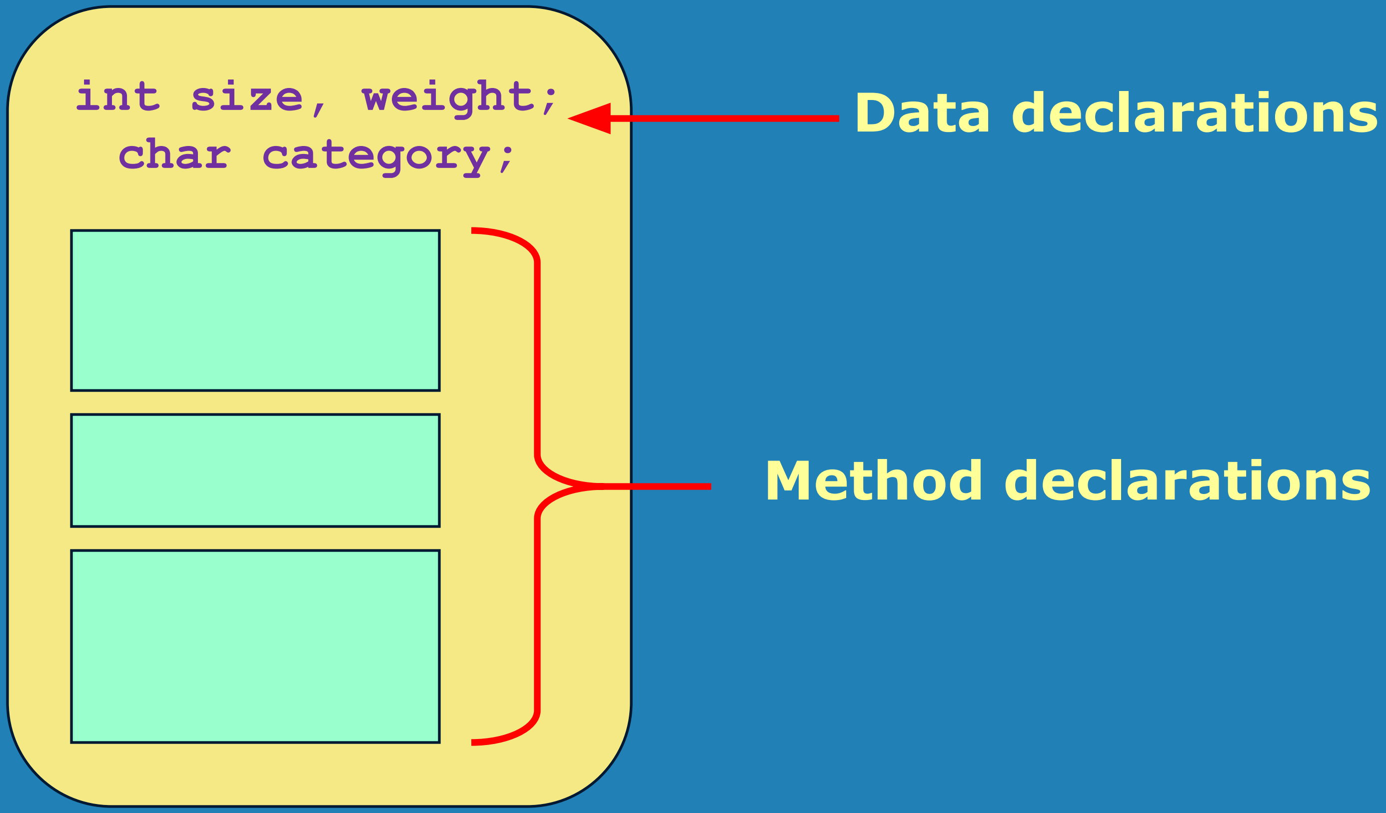
```
static Random rand = new Random();

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

- **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**
- **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

- See RollingDice.java (page 157)
- See Die.java (page 158)

```
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 Die Class

- The `Die` class contains two data values
  - numFaces that represents the maximum face value
  - an integer `faceValue` that represents the current face value
- The `roll` method uses the `random` method of the `Math` class to determine a new face value
- There are also methods to explicitly set and retrieve the current face value at any time

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**

Constructors

- As mentioned previously, a *constructor* is a special method that is used to set up an object when it is initially created
- A constructor has the same name as the class
- The `Die` constructor is used to set the number of faces value of each new die object to a user defined value (passed as a parameter)
- We examine constructors in more detail later

Data Scope

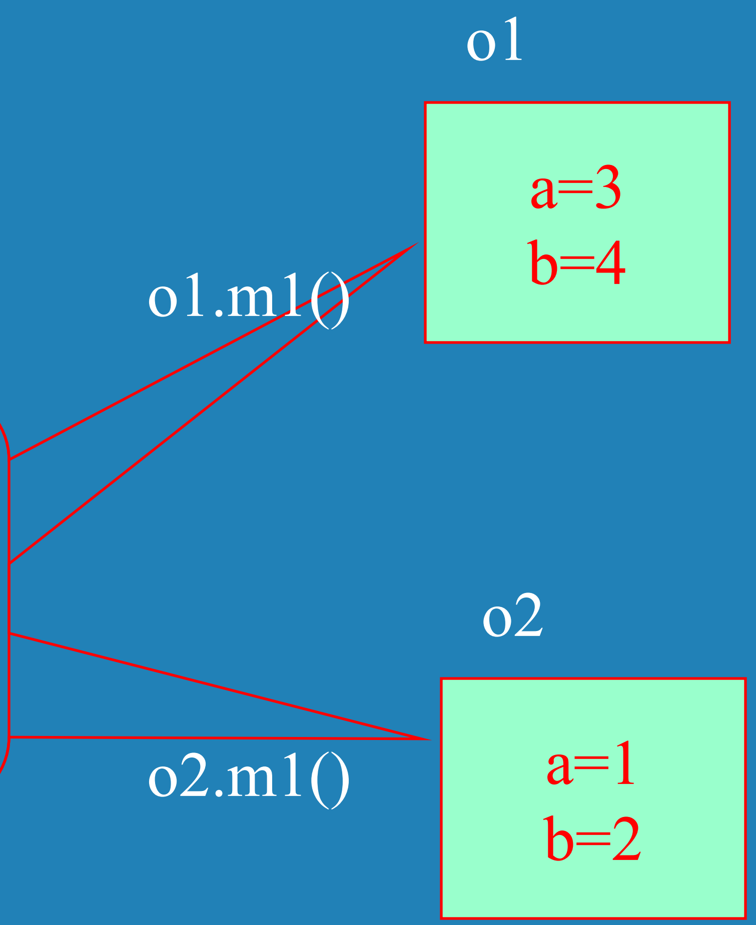
- **The *scope* of data is the area in a program in which that data can be referenced (used)**
- **Data declared at the class level can be referenced by all methods in that class**
- **Data declared within a method can be used only in that method**
- **Data declared within a method is called *local data***

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
    }
```

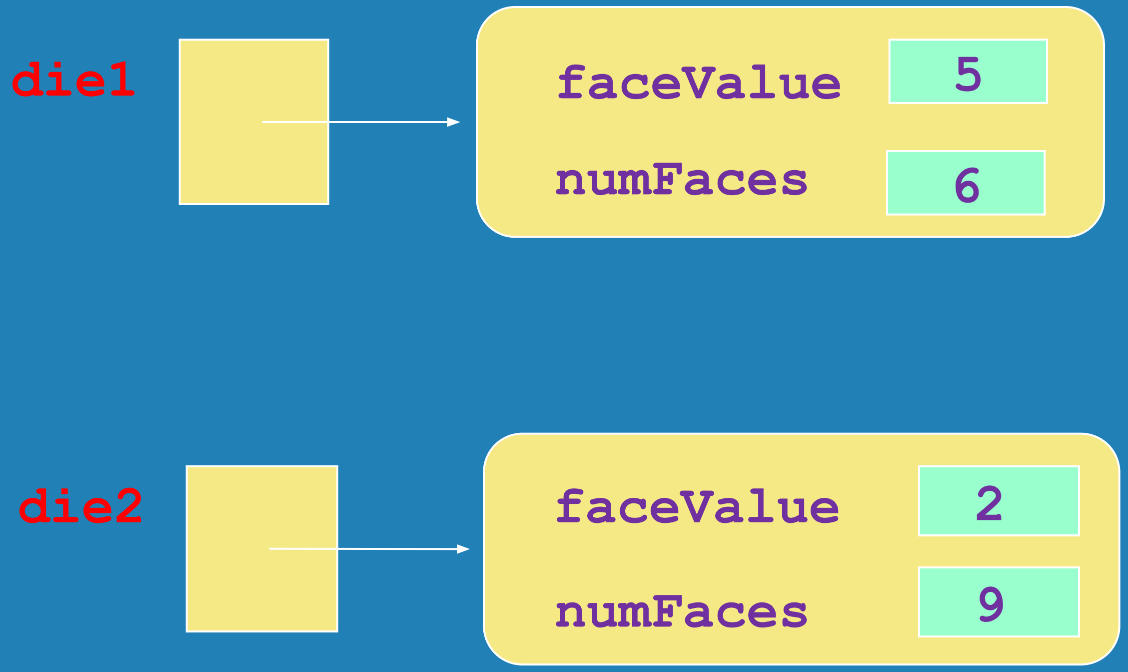
Instance Data

```
public class X {
    int a;
    int b;

    void m1 () {
     System.out.println(a);
     m2();
    }
    void m2() {
      System.out.println(b);
    }
```



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

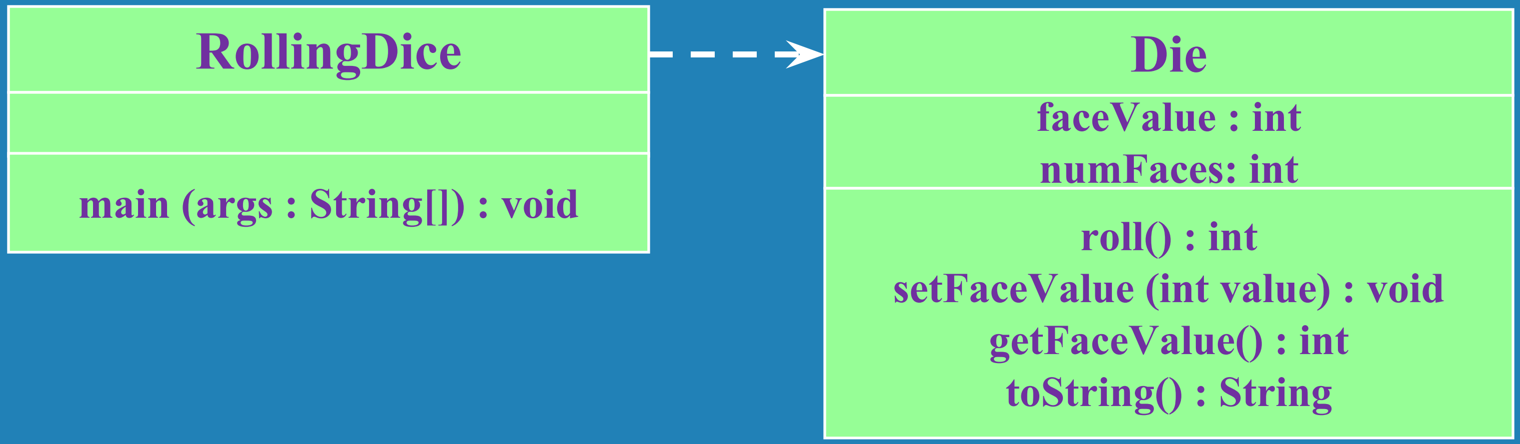
```
//  Flips two coins until one of them comes up
// heads three times  in a row.
public static void main (String[] args) {
    final int GOAL = 3;
    int count1 = 0, count2 = 0;

    // 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

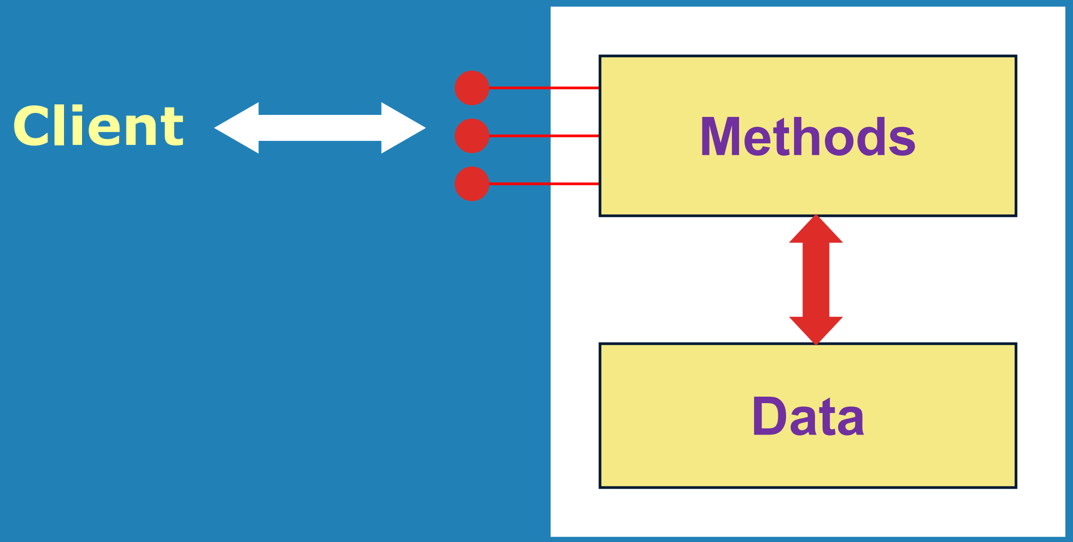
- **We can take one of two views of an object:**
  - **internal - the details of the variables and methods of the class that defines it**
  - **external - the services that an object provides and how the object interacts with the rest of the system**
- **From the external view, an object is an *encapsulated* 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

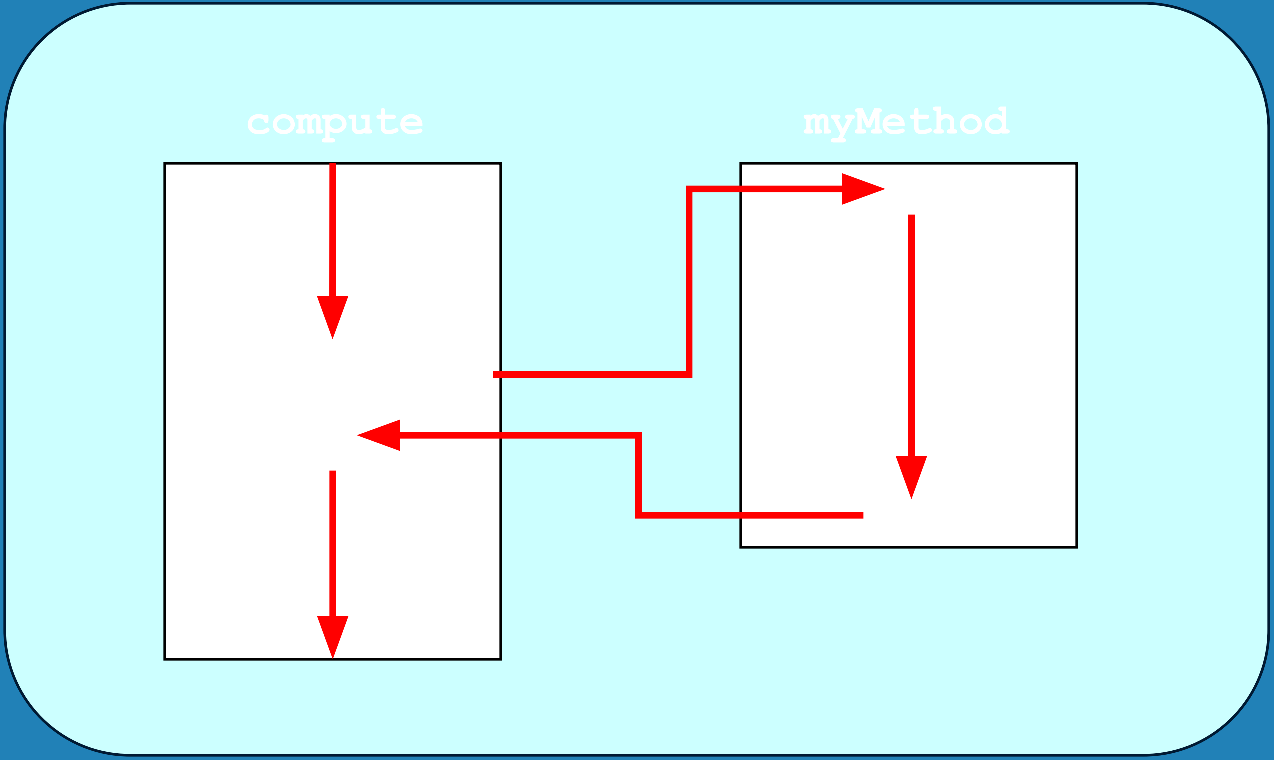
- **The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state**
- **A mutator is often designed so that the values of variables can be set only within particular limits**
- **For example, the `setFaceValue` mutator of the `Die` class restricts the value to the valid range (1 to `numFaces`)**

Method Declarations

- Let's now examine method declarations in more detail
- A *method declaration* specifies the code that will be executed when the method is invoked (called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined

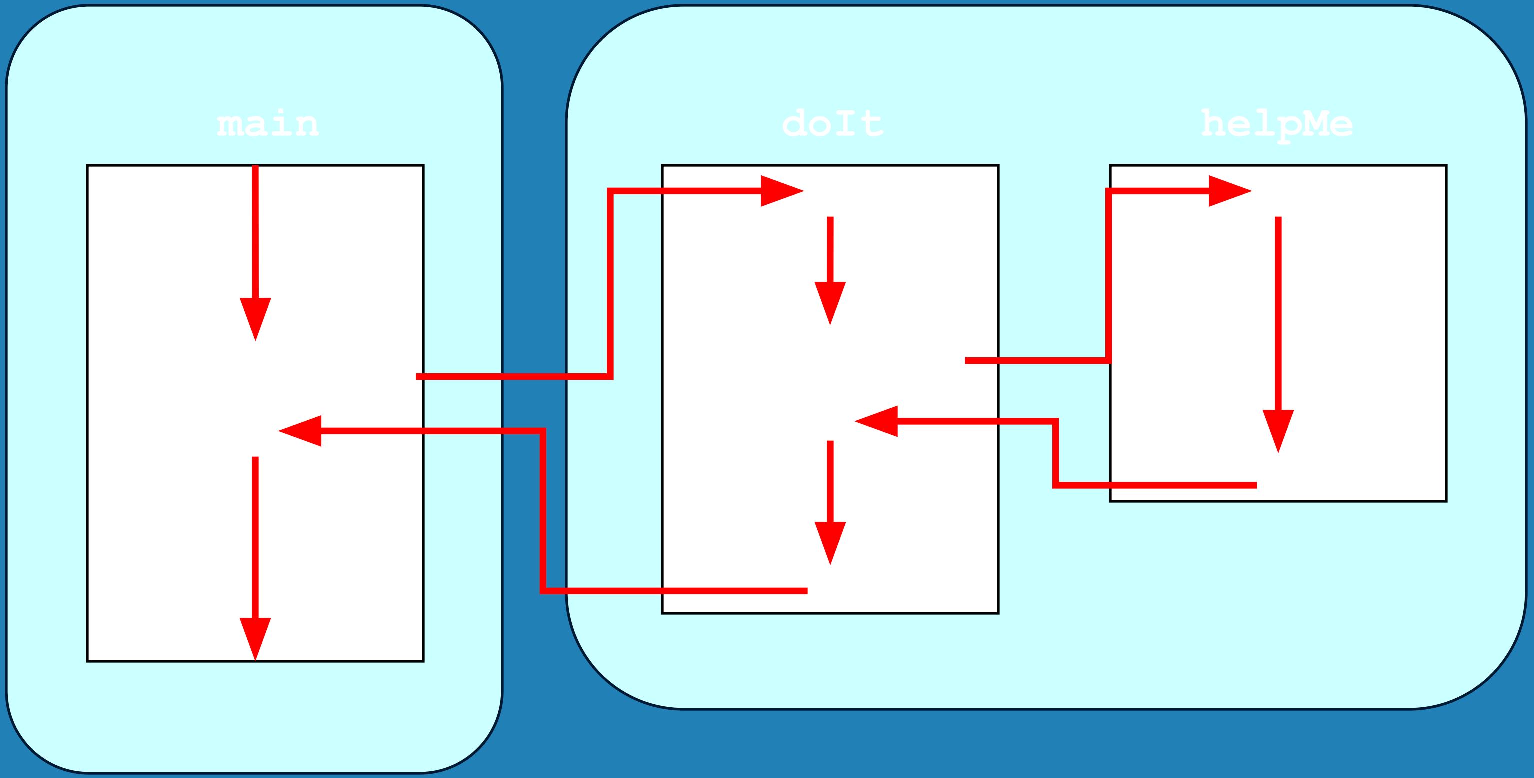
Method Control Flow

- If the called method is in the same class, only the method name is needed



Method Control Flow

- The called method is often part of another class or object



Method Header

- **A method declaration begins with a *method header***

```
char calc (int num1, int num2, String message)
```

method name

parameter list

return type

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Body

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

The return expression must be consistent with the return type

`sum` and `result` are local data

They are created each time the method is called, and are destroyed when it finishes executing

The return Statement

- **The *return type* of a method indicates the type of value that the method sends back to the calling location**
- **A method that does not return a value has a `void` return type**
- **A *return statement* specifies the value that will be returned**

```
return expression;
```

- **Its expression must conform to the return type**

Parameters

- When a method is called, the *actual parameters* in the invocation are copied into the *formal parameters* in the method header

```
ch = obj.calc (25, count, "Hello");
```

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

Local Data

- **As we've seen, local variables can be declared inside a method**
- **The formal parameters of a method create *automatic local variables* when the method is invoked**
- **When the method finishes, all local variables are destroyed (including the formal parameters)**
- **Keep in mind that instance variables, declared at the class level, exists as long as the object exists**

Constructors Revisited

- Recall that a constructor is a special method that is used to set up a newly created object

- When writing a constructor, remember that:
  - it has the same name as the class
  - it does not return a value
  - it has no return type, not even `void`
  - it often sets the initial values of instance variables

- The programmer does not have to define a constructor for a class

Bank Account Example

- **Let's look at another example that demonstrates the implementation details of classes and methods**
- **We'll represent a bank account by a class named `Account`**
- **It's state can include the account number, the current balance, and the name of the owner**
- **An account's behaviors (or services) include deposits and withdrawals, and adding interest**

Using the Account class

```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
Account acct2 = new Account ("Jane Smith", 69713, 40.00);
Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

acct1.deposit (25.85);

double smithBalance = acct2.deposit (500.00);
System.out.println ("Smith balance after deposit: " +
                    smithBalance);

System.out.println ("Murphy balance after withdrawal: " +
                    acct2.withdraw (430.75, 1.50));

acct1.addInterest();
acct2.addInterest();
acct3.addInterest();

System.out.println ();
System.out.println (acct1);
System.out.println (acct2);
System.out.println (acct3);
```

Account class

```
public class Account
{
   private final double RATE = 0.035;  // interest rate of 3.5%

   private long acctNumber;
   private double balance;
   private String name;

   //-----------------------------------------------------------------
   //  Sets up the account by defining its owner, account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account (String owner, long account, double initial)
   {
      name = owner;
      acctNumber = account;
      balance = initial;
   }
```

```
//-----------------------------------------------------------------
// Deposits the specified amount into the account. Returns the
// new balance.
//-----------------------------------------------------------------
public double deposit (double amount)  {
   if (amount > 0)
        balance = balance + amount;
   return balance;
}

//-----------------------------------------------------------------
// Withdraws the specified amount from the account and applies
// the fee. Returns the new balance.
//-----------------------------------------------------------------
public double withdraw (double amount)  {
    if (amount <= balance)
        balance = balance - amount ;
   return balance;
}
```

```java
//-----------------------------------------------------------------
   //  Returns the current balance of the account.
   //-----------------------------------------------------------------
   public double getBalance ()
   {
      return balance;
   }

   //-----------------------------------------------------------------
   //  Returns a one-line description of the account as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      NumberFormat fmt = NumberFormat.getCurrencyInstance();

      return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
   }
}
```