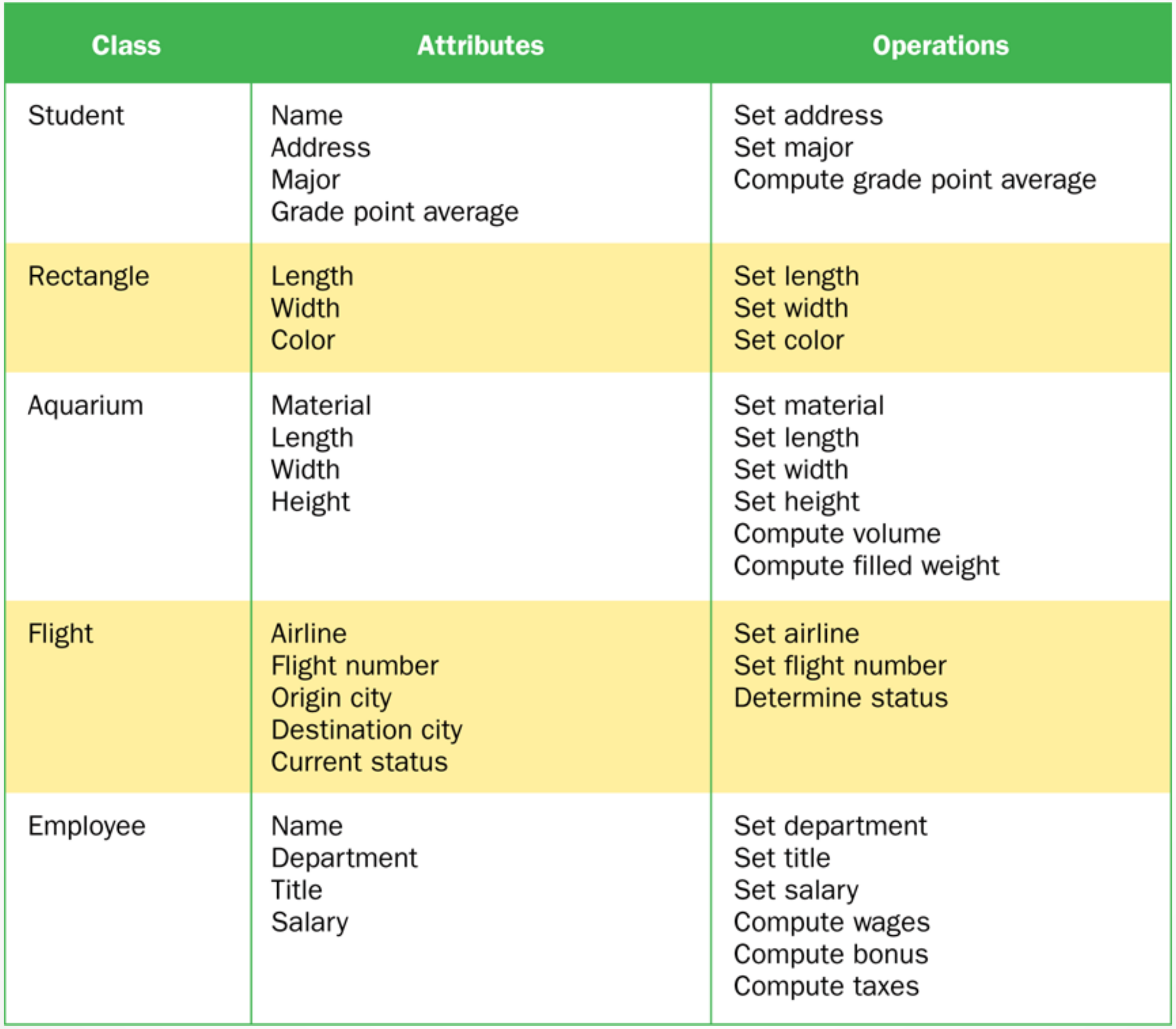
CSC220 (CSI)
Computational Problem Solving

Writing Classes

The College of New Jersey

Please turn off your cell phone!

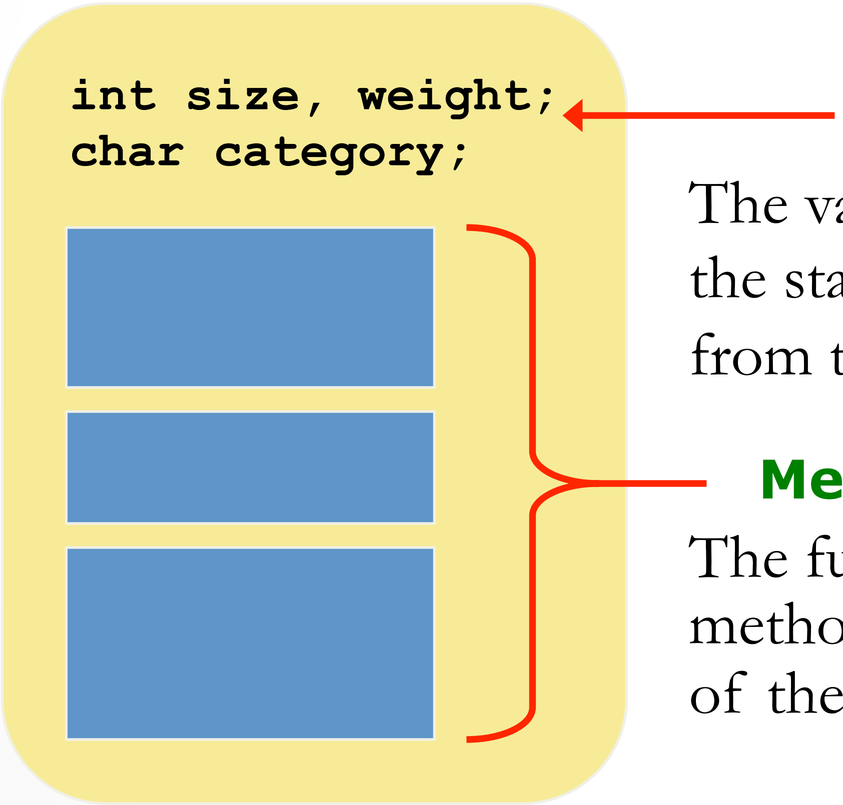
Examples of Classes



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

Classes

- A class can contain data declarations and method declarations



```
int size, weight;
char category;
```

Data declarations

The values of the data define the state of an object created from the class

Method declarations

The functionality of the methods define the behaviors of the object

Classes and Objects

- Recall that an object has state and behavior
- Consider a six-sided die (singular of dice)
  - It's state can be defined as which face is showing
  - It's primary behavior is that it can be rolled
- We represent a die by designing a class called `Die` that models this state and behavior
  - The class serves as the blueprint for a die object
- We can then instantiate as many die objects as we need for any particular program

```
//**********************************************
//  RollingDice.java       Author: Lewis/Loftus
//  Demonstrates the creation and use of a user-
//**********************************************
public class RollingDice {
   //  Creates two Die objects and rolls them se
   public static void main(String[] args){
      Die die1, die2;
      int sum;
      die1 = new Die();
      die2 = new Die();
      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);
   }
}
```

Sample Run

```
Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 6
```

```java
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//  Represents one die (singular of dice) 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
   //  Constructor: Sets the initial face value.
   public Die(){
      faceValue = 1;
   }
   //  Rolls the die and returns the result.
   public int roll(){
      faceValue = (int)(Math.random() * MAX) + 1;
      return faceValue;
   }
   //  Face value mutator.
   public void setFaceValue(int value) {
      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;
   }
}
```

Data Scope

- The scope of data : 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. Such data is called local data.
- In the `Die` class, the variable `result` is declared inside the `toString` method -- it is local to that method and cannot be referenced anywhere else

Instance Data

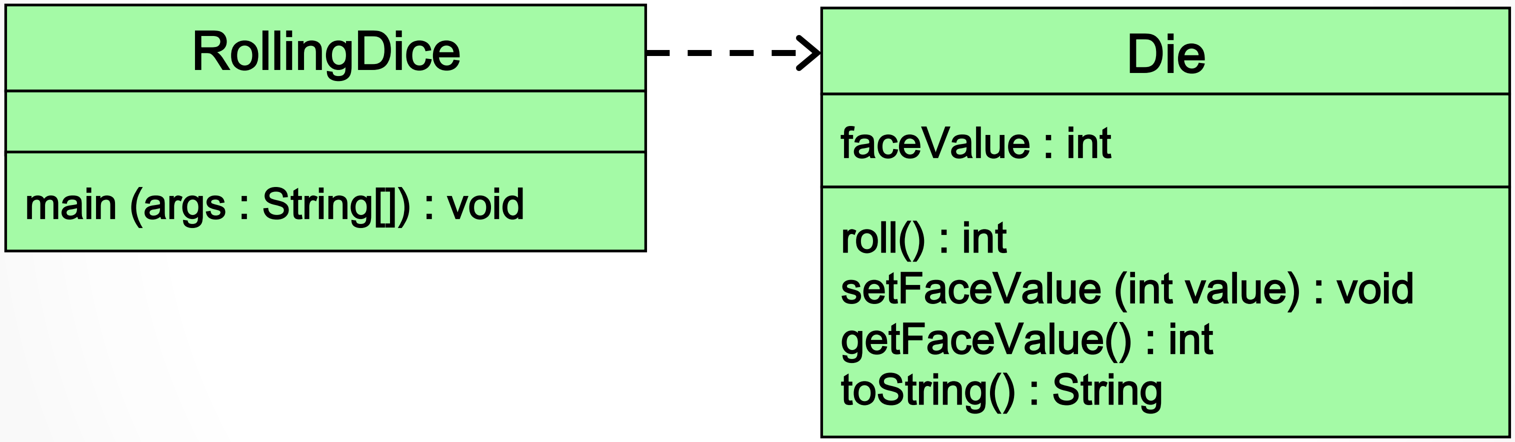
- A variable declared at the class level (such as `faceValue`) is called instance data
- Each instance (object) has its own instance variable
- A class declares the type of the data, but it does not reserve memory space for it
- Each time a `Die` object is created, a new `faceValue` variable is created 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

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:



RollingDice

main (args : String[]) : void

Die

faceValue : int

roll() : int
setFaceValue (int value) : void
getFaceValue() : int
toString() : String

Quick Check

Where is instance data declared?

At the class level.

What is the scope of instance data?

It can be referenced in any method of the class.

What is local data?

Local data is declared within a method, and is only accessible in that method.

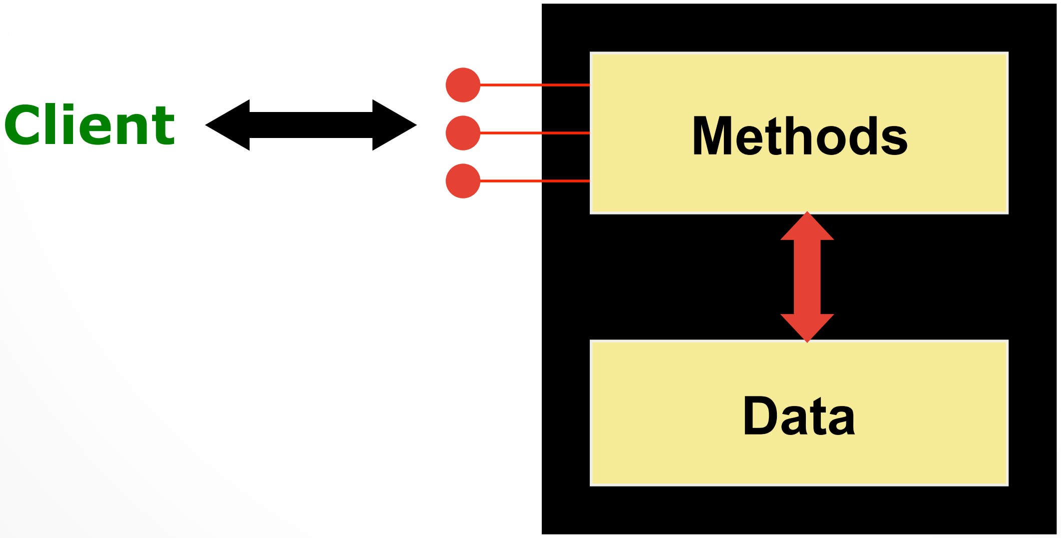
Encapsulation

- 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

- 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 and they manage 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
  - `final` modifier : define constants
- Visibility modifiers:
  - `public` : can be referenced anywhere
  - `protected` : involves inheritance, will discuss later
  - `private` : can be referenced only within the class

Visibility Modifiers

- If possible, use private. If there is not other choice, use public or protected.
- If you have to access some data from outside, it is a good practice to set the data itself to be *private*, but set the methods manipulate the data to be *public*.
- The manipulating methods can control who can access and how they access.
  - 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"

Visibility Modifiers

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

Quick Check

Why was the `faceValue` variable declared as `private` in the `Die` class?

> By making it private, each `Die` object controls its own data and allows it to be modified only by the well-defined operations it provides.

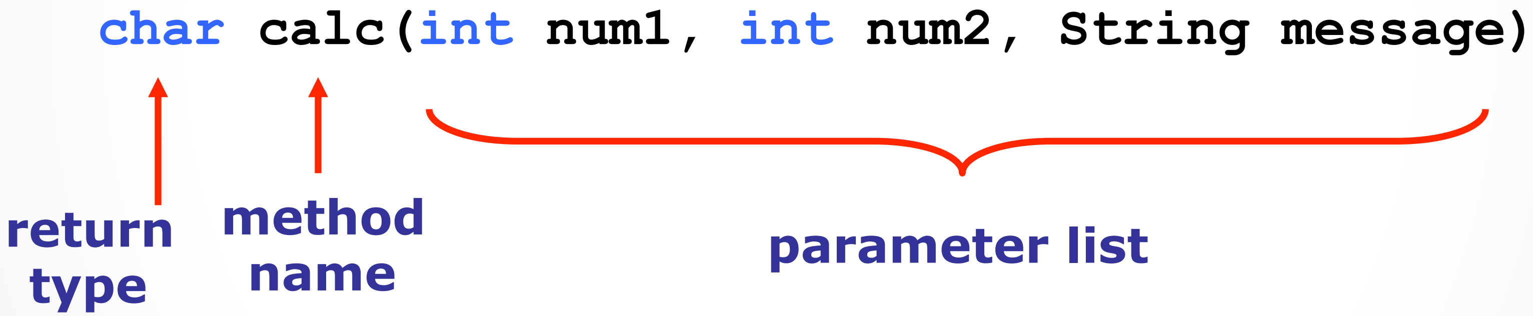
Why is it ok to declare `MAX` as `public` in the `Die` class?

> `MAX` is a constant. Its value cannot be changed. Therefore, there is no violation of encapsulation.

Method Declarations

- A method declaration specifies the code that will be executed when the method is invoked (called)

- A method declaration begins with a method header



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

return type

method name

parameter list

A method that does not return a value has a `void` 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;
}
```

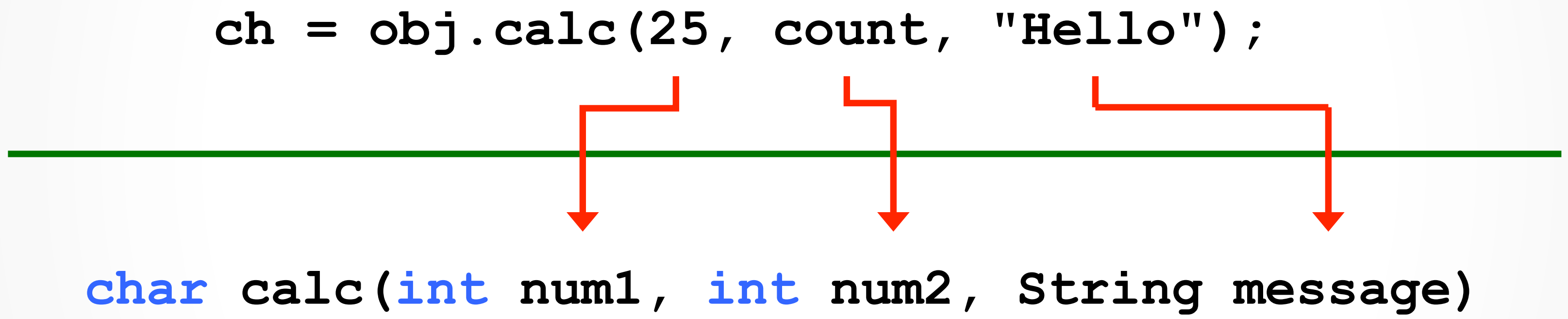
The return expression must be consistent with the return type

sum and result are local data

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

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");
```

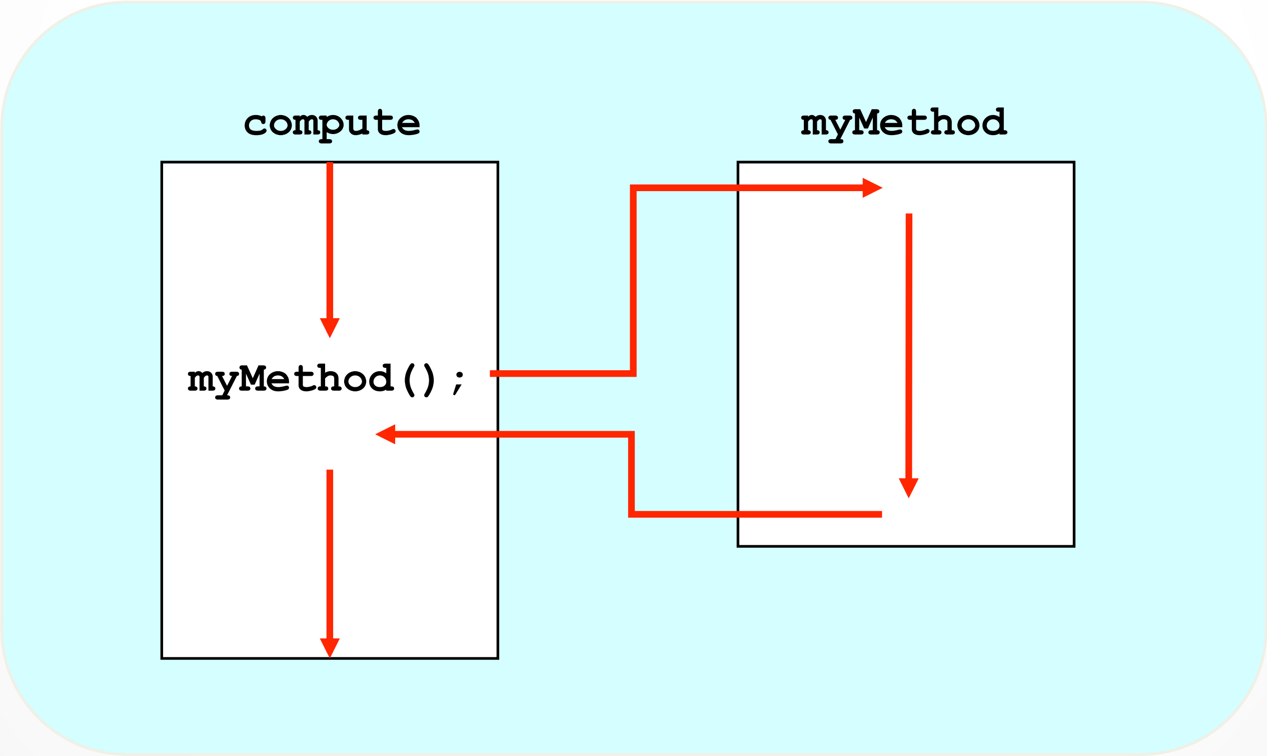
```
char calc(int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt(sum);

   return result;
}
```

Local Data

- 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

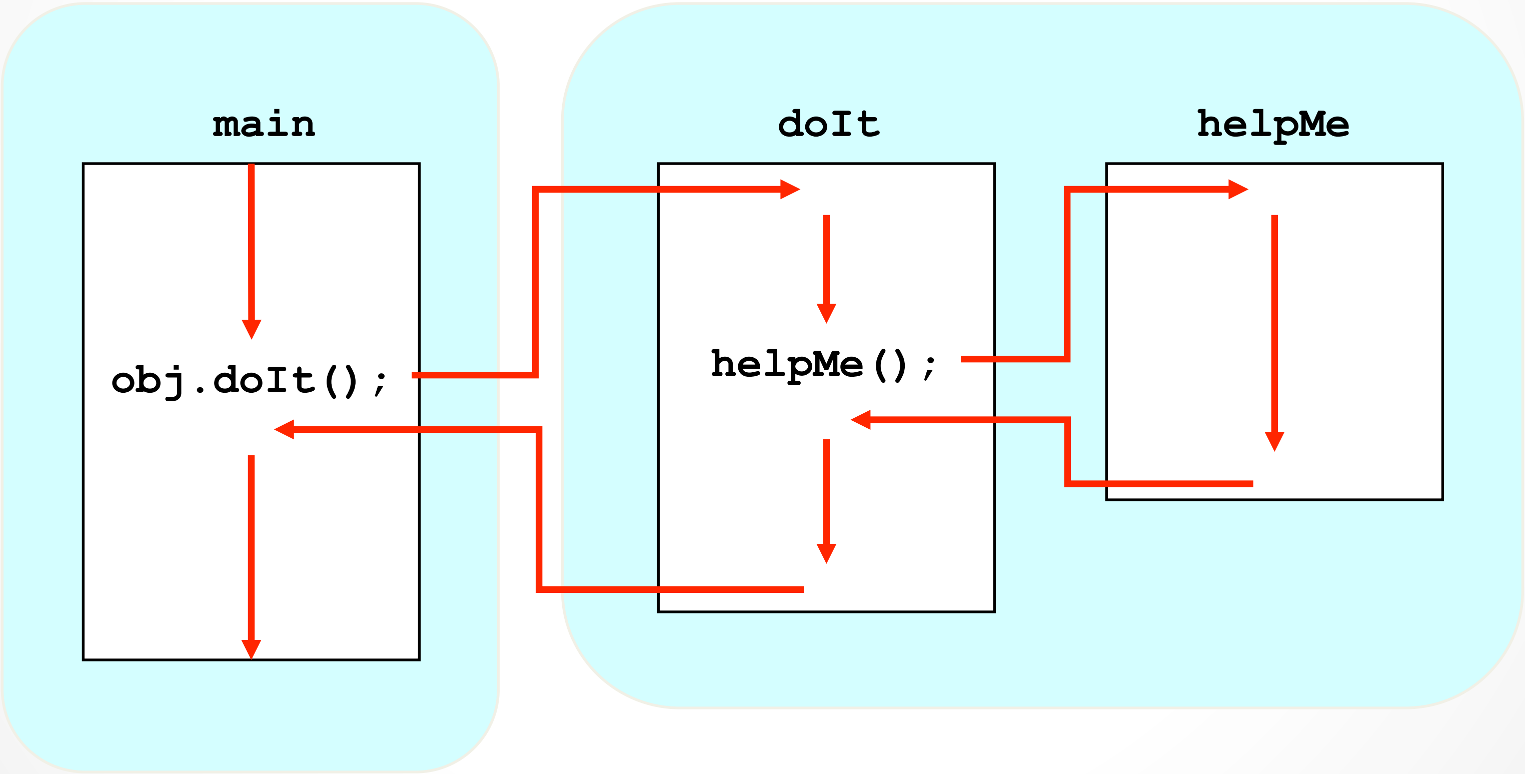
Method Control Flow

- 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



Bank Account Example

- We'll represent a bank account by a class named `Account`
- Its 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

Driver Programs

- A *driver program* drives the use of other, more interesting parts of a program

- Driver programs are often used to test other parts of the software

- The `Transactions` class contains a `main` method that drives the use of the `Account` class, exercising its services

- See `Transactions.java`
- See `Account.java`

```java
//********************************************************************
//  Transactions.java       Author: Lewis/Loftus
//  Demonstrates the creation and use of multiple Account objects.
//********************************************************************
public class Transactions{
   //-----------------------------------------------------------------
   //  Creates some bank accounts and requests various services.
   //-----------------------------------------------------------------
   public static void main(String[] args){
      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("Smith 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);
   }
}
```

Output

```
Smith balance after deposit: 540.0
Smith balance after withdrawal: 107.55

72354   Ted Murphy      $132.90
69713   Jane Smith      $111.52
93757   Edward Demsey   $785.90
```

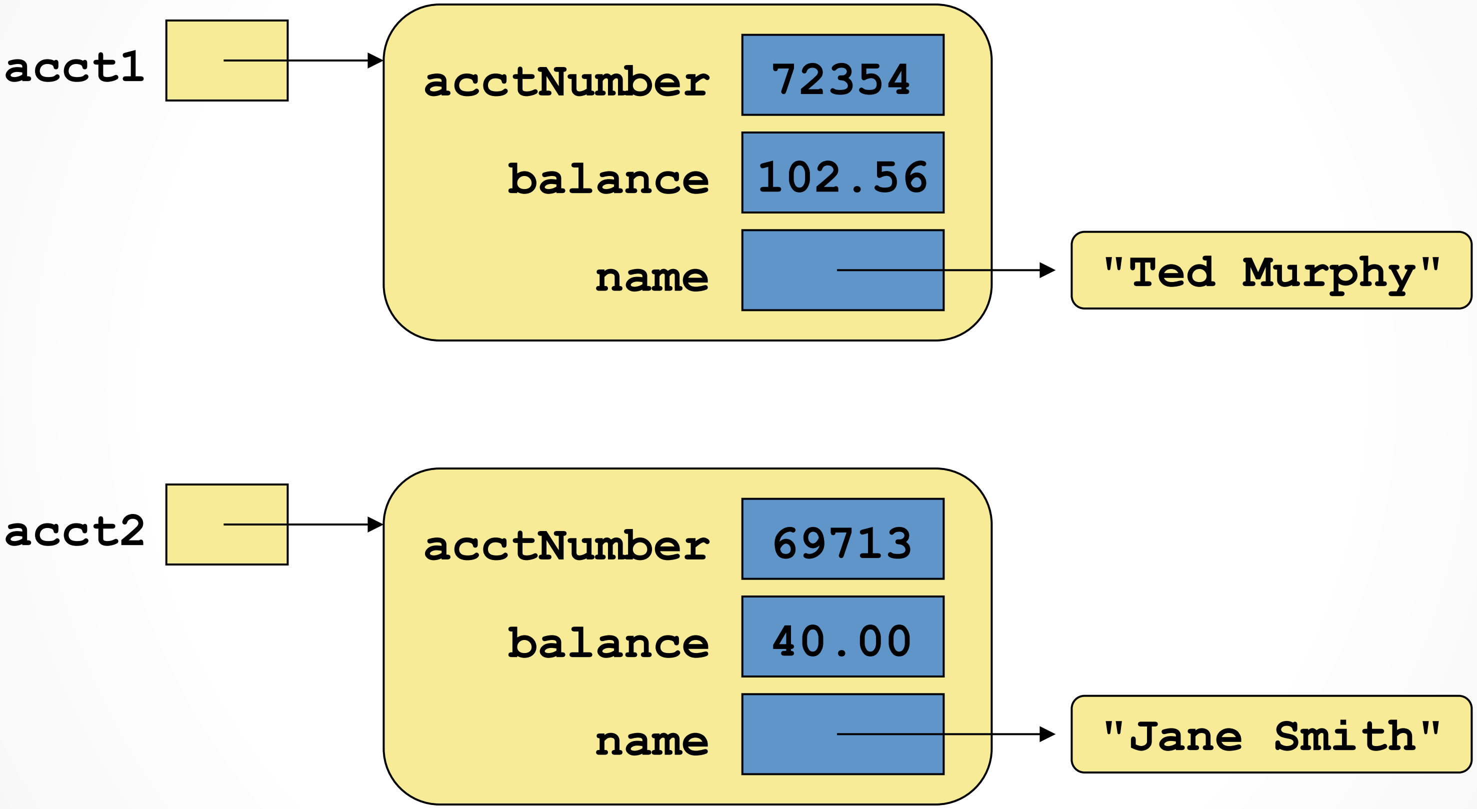
```
//********************************************************************
//  Account.java       Author: Lewis/Loftus
//  Represents a bank account with basic services such as deposit
//  and withdraw.
//********************************************************************
import java.text.NumberFormat;
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){
      balance = balance + amount;
      return balance;
   }
```

continue

continue

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

   //  Adds interest to the account and returns the new balance.
   public double addInterest(){
      balance += (balance * RATE);
      return balance;
   }

   //  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));
   }
}
```

Bank Account Example



Bank Account Example

- There are some improvements that can be made to the `Account` class
- Formal getters and setters could have been defined for all data
- The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw` method is positive

Constructors Revisited

- Note that a constructor has no return type specified 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

Quick Check

How do we express which `Account` object's balance is updated when a deposit is made?

Each account is referenced by an object reference variable:

```
Account myAcct = new Account(…);
```

and when a method is called, you call it through a particular object:

```
myAcct.deposit(50);
```