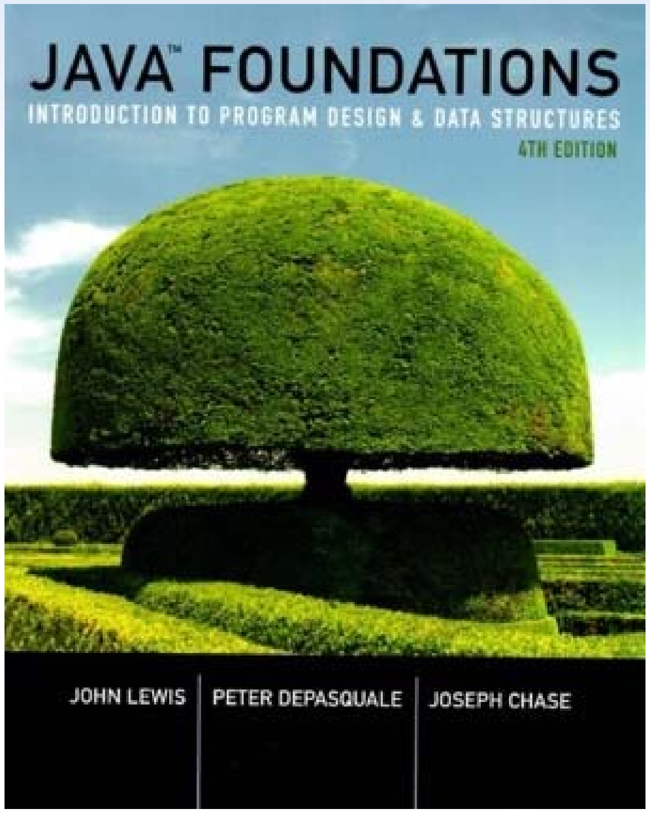


Chapter 5

Writing Classes

Chapter Scope

- Identifying classes and objects
- Structure and content of classes
- Instance data
- Visibility modifiers
- Method structure
- Constructors
- Relationships among classes
- Static methods and data

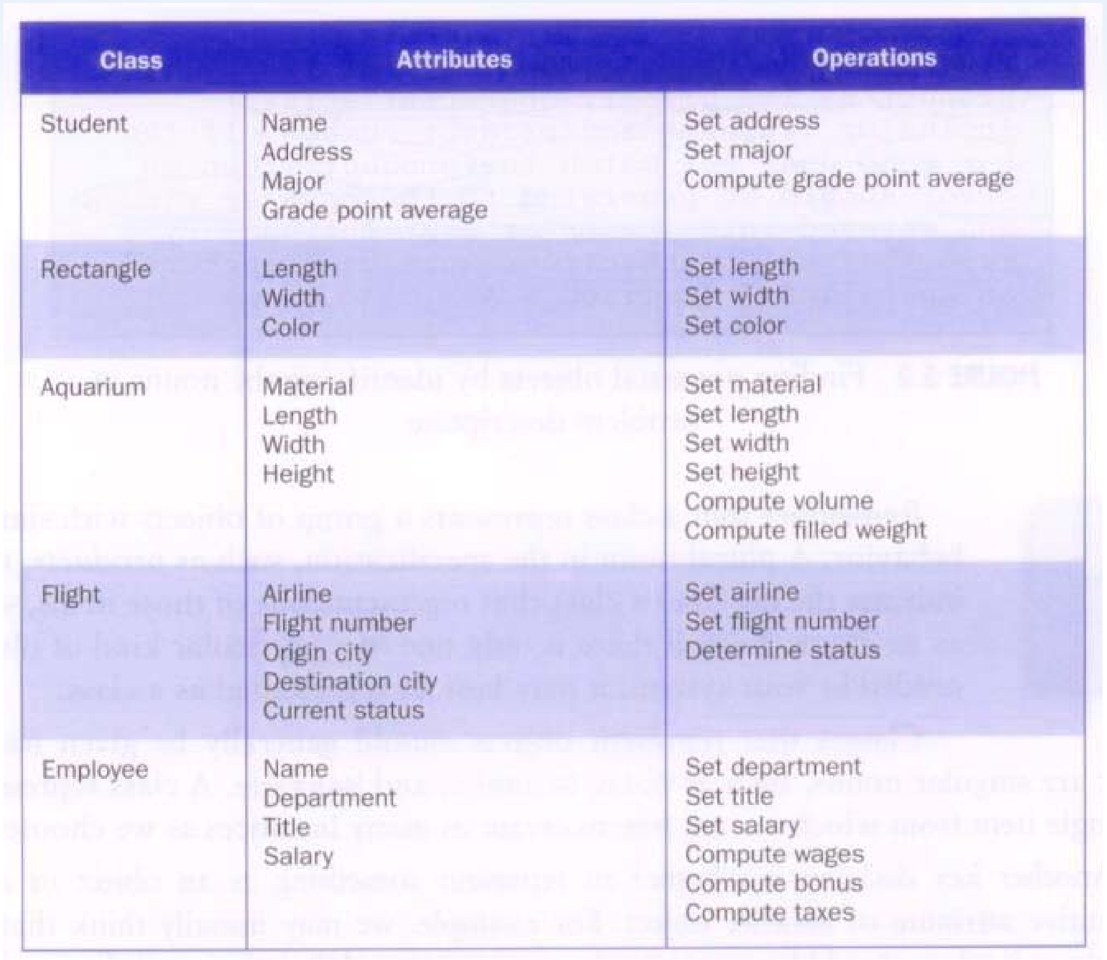
Classes and Objects Revisited

- The programs we’ve written in previous examples have used classes defined in the Java API
- 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

Classes and Objects

- Recall that 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 also has behaviors, defined by the operations associated with it
- Operations are defined by the methods of the class

Classes and Objects



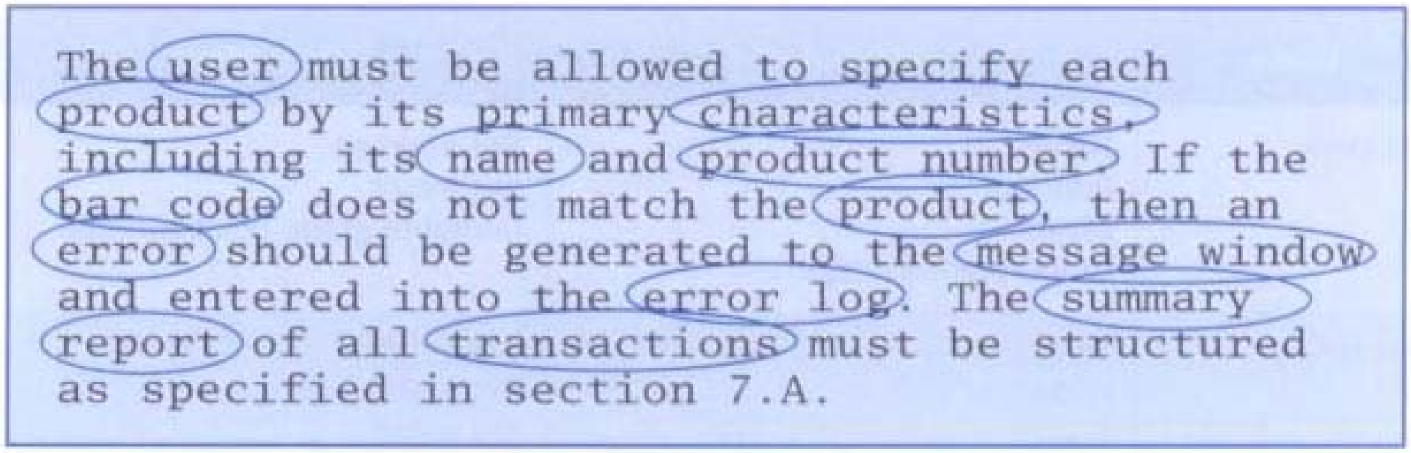
Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
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

Identifying Classes and Objects

- A class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object
- We are free to instantiate as many of each object as needed

Identifying Classes and Objects

- One way to find potential objects is by identifying the nouns in a problem description:



The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class
- For example, should an employee's address be represented as a set of 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

Identifying Classes and Objects

- We want to define classes with the proper amount of detail

- For example, 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

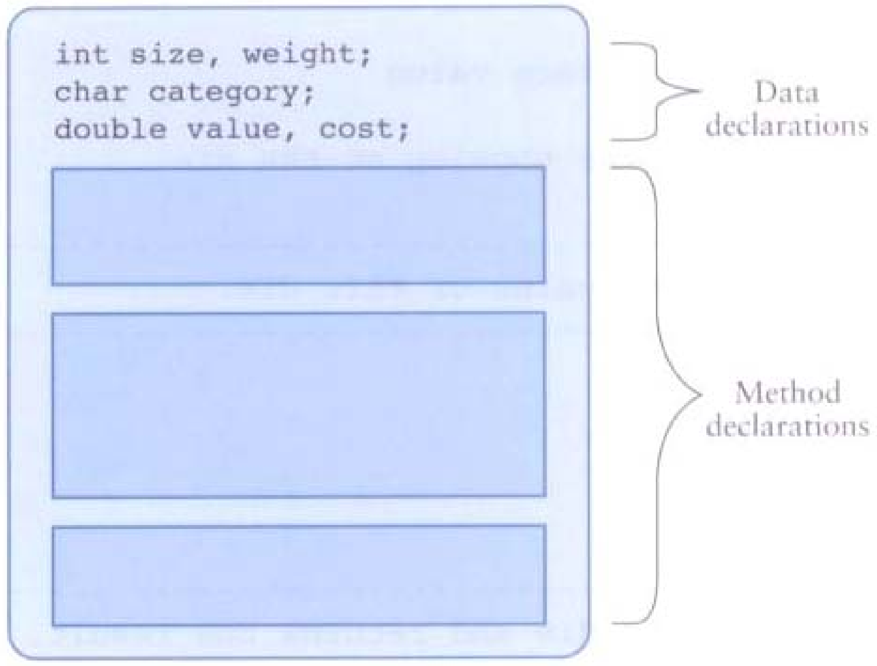
- It all depends on the details of the problem being solved

Identifying Classes and Objects

- 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

Anatomy of a Class

- A class contains data declarations and method declarations



Anatomy of a Class

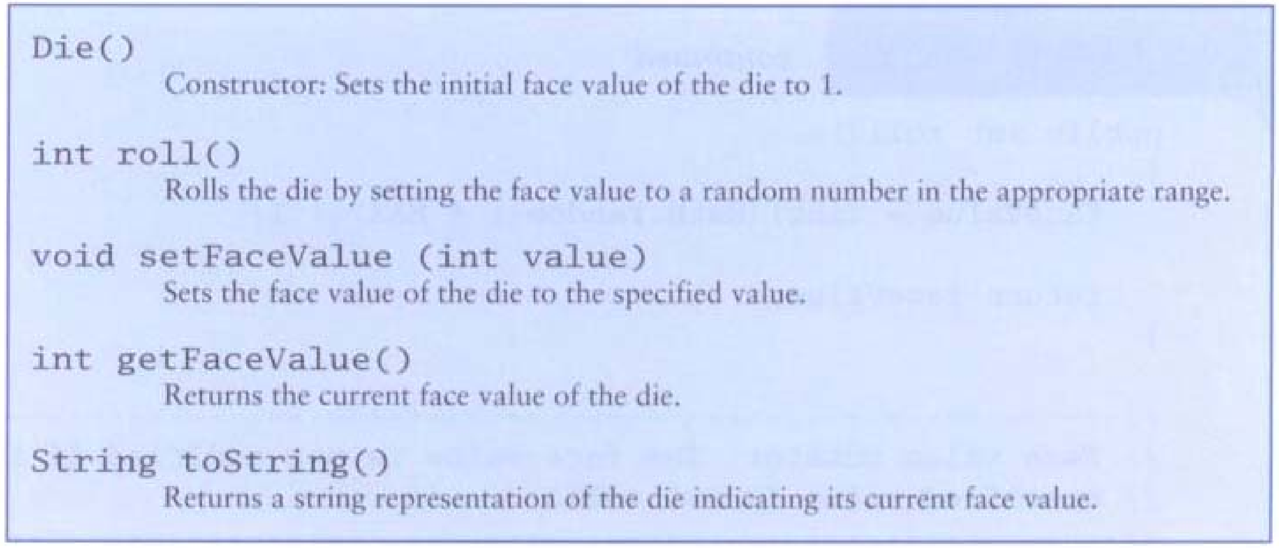
- Consider a six-sided die (singular of dice)
  - It’s state can be defined as which face is showing
  - It’s primary behavior is that it can be rolled
- We can represent a die in software by designing a class called `Die` that models this state and behavior
- We’ll want to design it so that it's a versatile and reusable resource
- Any given program will not necessarily use all aspects of a given class

```
//********************************************************************
//  SnakeEyes.java       Java Foundations
//
//  Demonstrates the use of a programmer-defined class.
//********************************************************************

public class SnakeEyes
{
   //-----------------------------------------------------------------
   //  Creates two Die objects and rolls them several times, counting
   //  the number of snake eyes 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);
   }
}
```

```java
//********************************************************************
//  Die.java       Java Foundations
//
//  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 of this die.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }

   //-----------------------------------------------------------------
   //  Computes a new face value for this die and returns the result.
   //-----------------------------------------------------------------
   public int roll()
   {
      faceValue = (int)(Math.random() * MAX) + 1;

      return faceValue;
   }
   //-----------------------------------------------------------------
   //  Face value mutator. The face value is not modified if the
   //  specified value is not valid.
   //-----------------------------------------------------------------
   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;
   }
}
```

Anatomy of a Class

- The primary difference between the `Die` class and other classes you've used is that the `Die` class is not part of the Java API



`Die()`
Constructor: Sets the initial face value of the die to 1.

`int roll()`
Rolls the die by setting the face value to a random number in the appropriate range.

`void setFaceValue (int value)`
Sets the face value of the die to the specified value.

`int getFaceValue()`
Returns the current face value of the die.

`String toString()`
Returns a string representation of the die indicating its current face value.

The toString Method

- Most classes 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 printed using 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 initial face value of each new die object to one
- We examine constructors in more detail later in this chapter

Data Scope

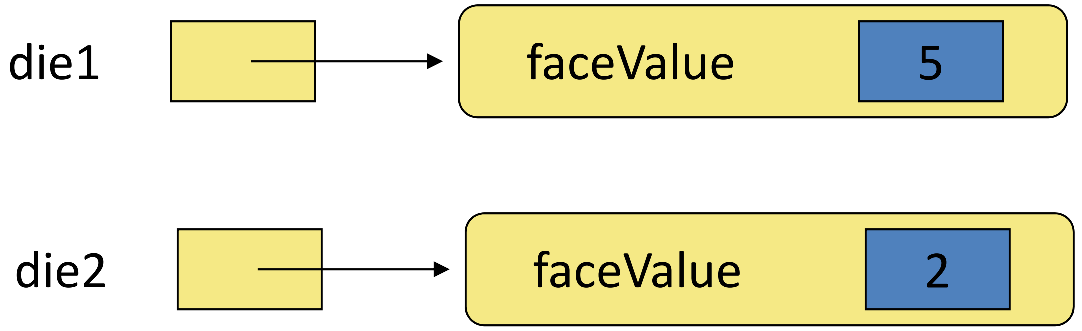
- The *scope* of data is the area in a program in which that data can be referenced (used)
- Data declared at the class level can be referenced by all methods in that class
- Data declared within a method can be used only in that method
- Data declared within a method is called *local data*
- In the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

- 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 `SnakeEyes` program as follows:



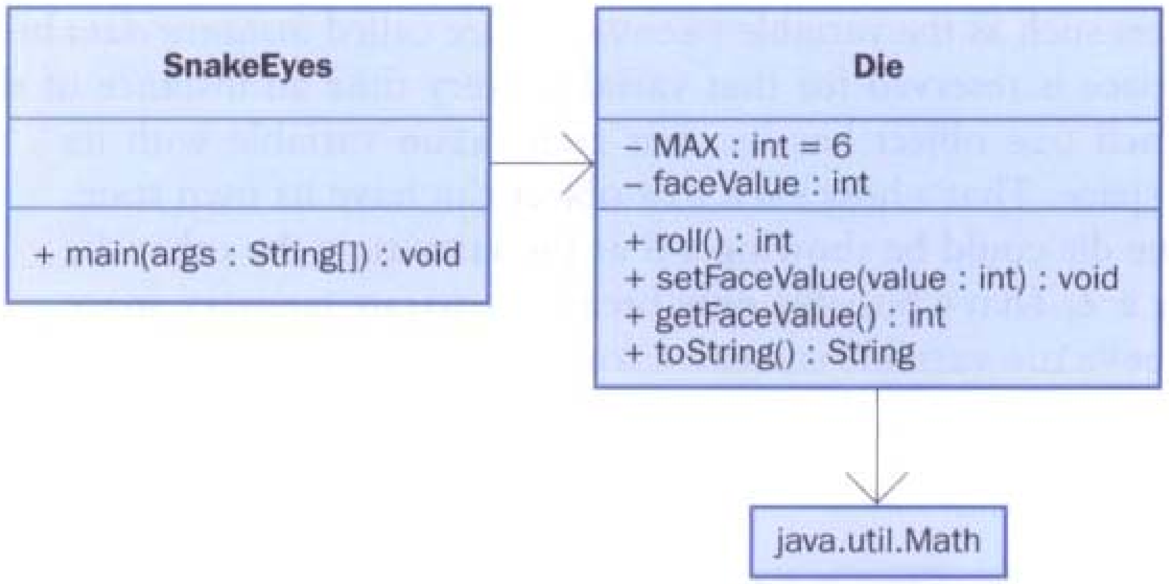
- Each object maintains its own `faceValue` variable, and thus its own state

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 solid arrow shows that one class *uses* the other (calls its methods)

UML Diagrams

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

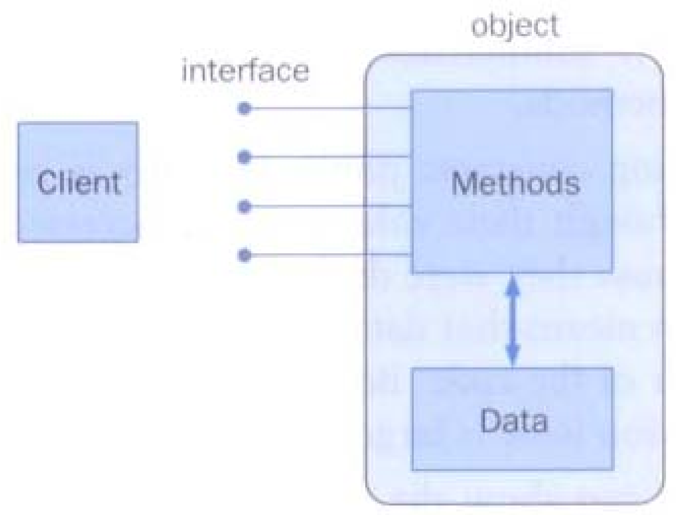
Encapsulation

- 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 an 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

- An overview of all Java modifiers is presented in Appendix E

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

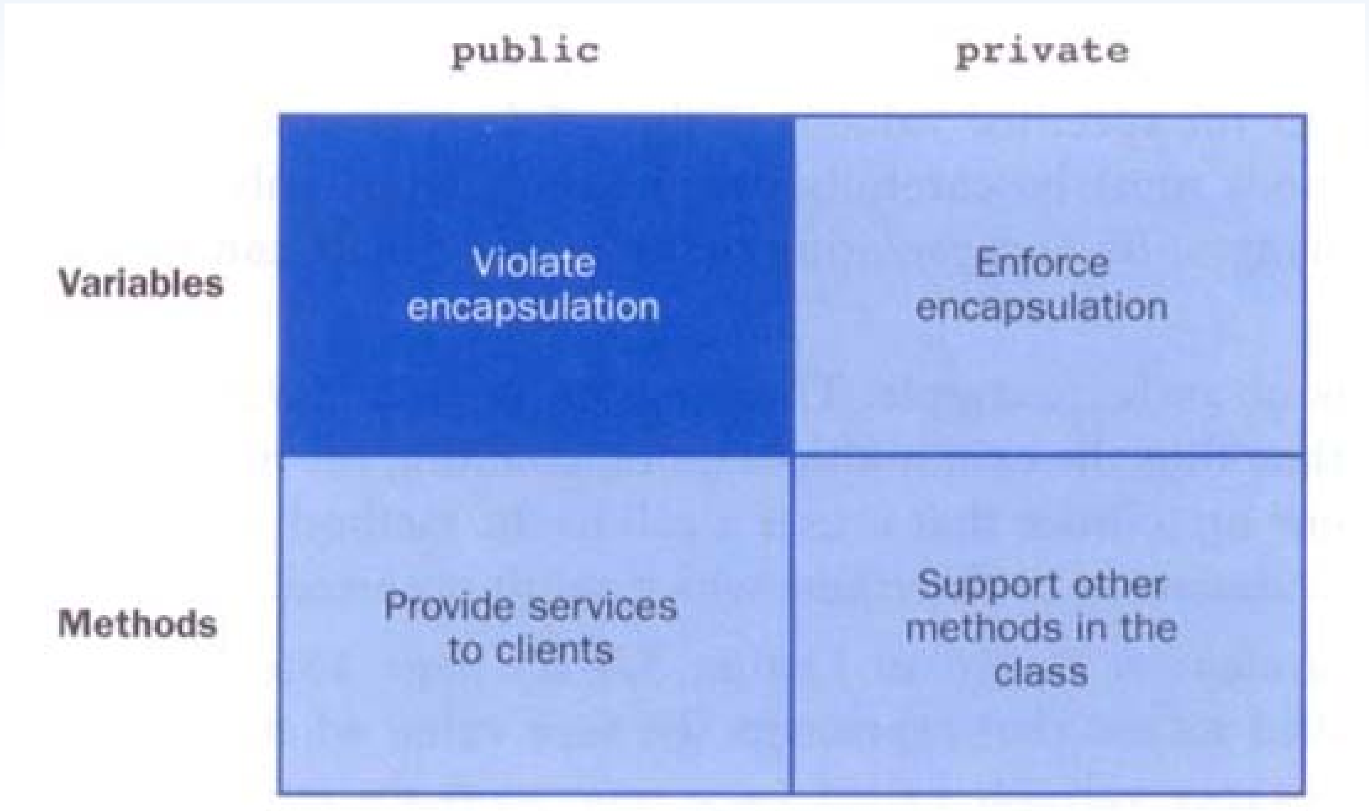
- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients

- 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

Accessors and Mutators

- They are sometimes called “getters” and “setters”
- In the `Coin` class
  - The `isHeads` method is an accessor
  - The `flip` method is a mutator
- The `Coin` class is used in two different examples: `CountFlips` and `FlipRace`

```java
//********************************************************************
//  CountFlips.java       Java Foundations
//
//  Demonstrates the use of programmer-defined class.
//********************************************************************

public class CountFlips
{
   //-----------------------------------------------------------------
   //  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);
   }
}
```

```java
//********************************************************************
//  Coin.java       Java Foundations
//
//  Represents a coin with two sides that can be flipped.
//********************************************************************

public class Coin
{
   private final int HEADS = 0;  // tails is 1

   private int face;  // current side showing

   //-----------------------------------------------------------------
   //  Sets up this coin by flipping it initially.
   //-----------------------------------------------------------------
   public Coin()
   {
      flip();
   }

   //-----------------------------------------------------------------
   //  Flips this coin by randomly choosing a face value.
   //-----------------------------------------------------------------
   public void flip()
   {
      face = (int) (Math.random() * 2);
   }
   //-----------------------------------------------------------------
   //  Returns true if the current face of this coin is heads.
   //-----------------------------------------------------------------
   public boolean isHeads()
   {
      return (face == HEADS);
   }

   //-----------------------------------------------------------------
   //  Returns the current face of this coin as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return (face == HEADS) ? "Heads" : "Tails";
   }
}
```

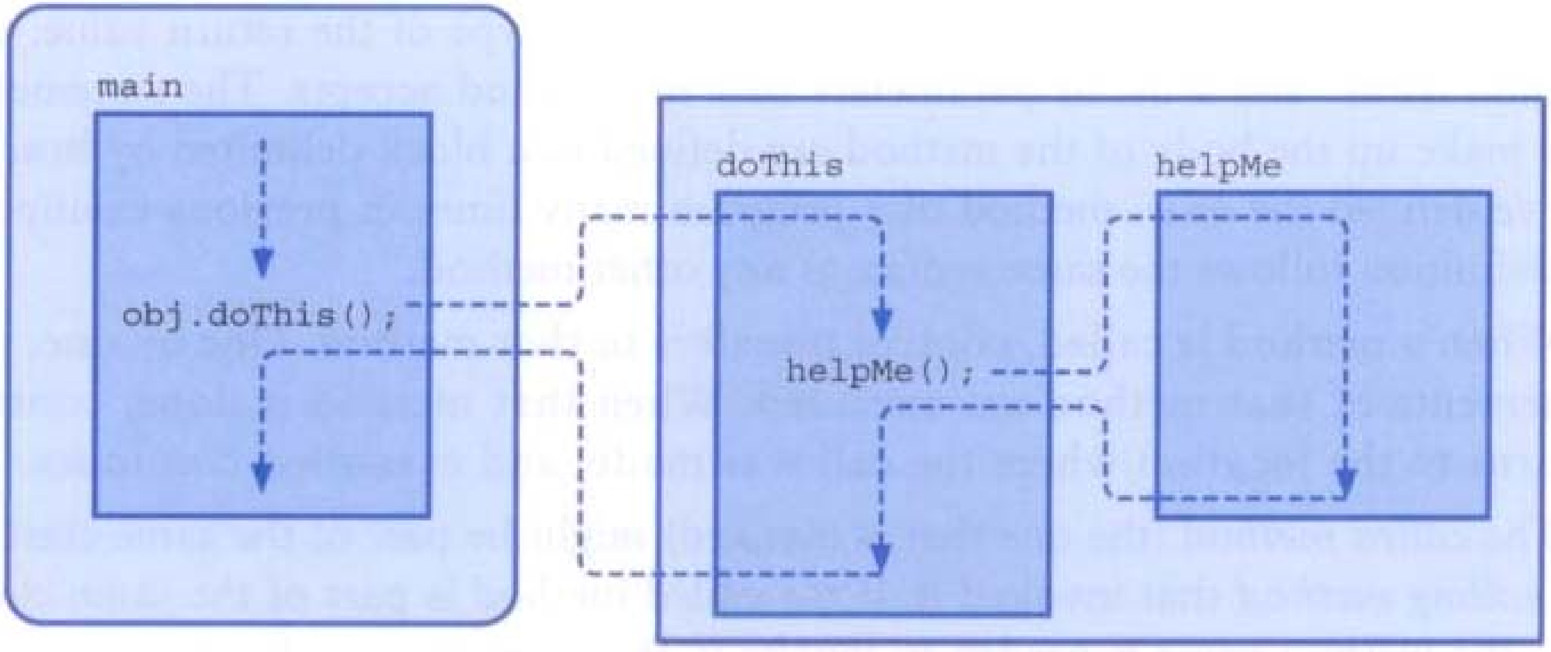
```java
//********************************************************************
//  FlipRace.java       Java Foundations
//
//  Demonstrates the reuse of programmer-defined class.
//********************************************************************

public class 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;
      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);

         // 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!");
   }
}
```

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

Methods

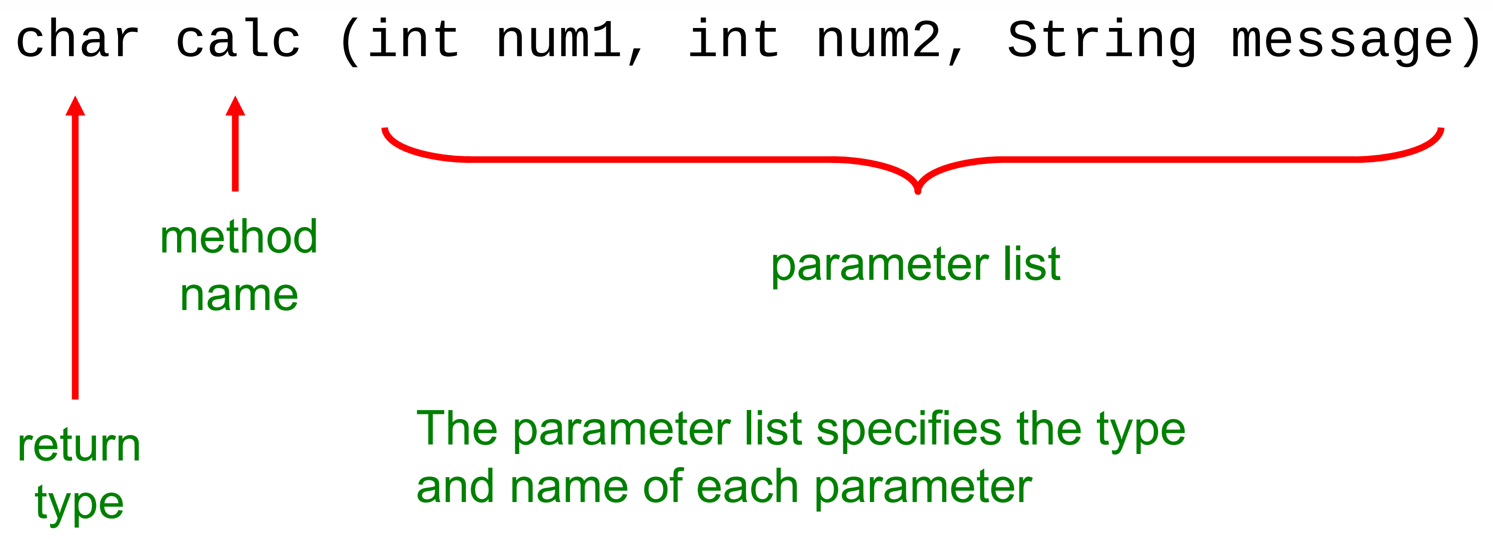
- The flow of control through methods:



Method Header

- A method declaration begins with a *method header*

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



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

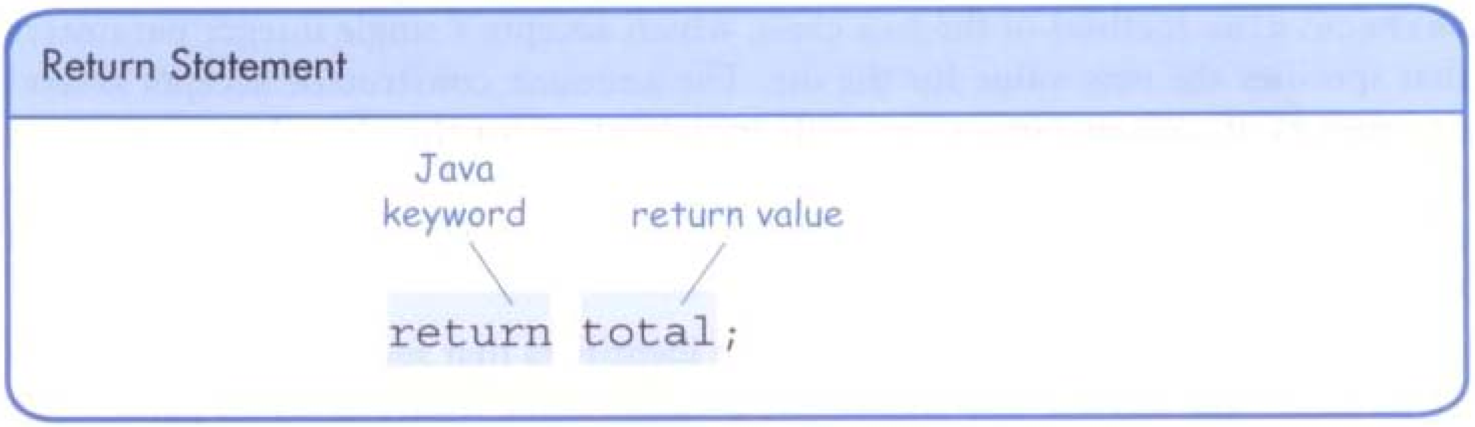
The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the caller
- 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

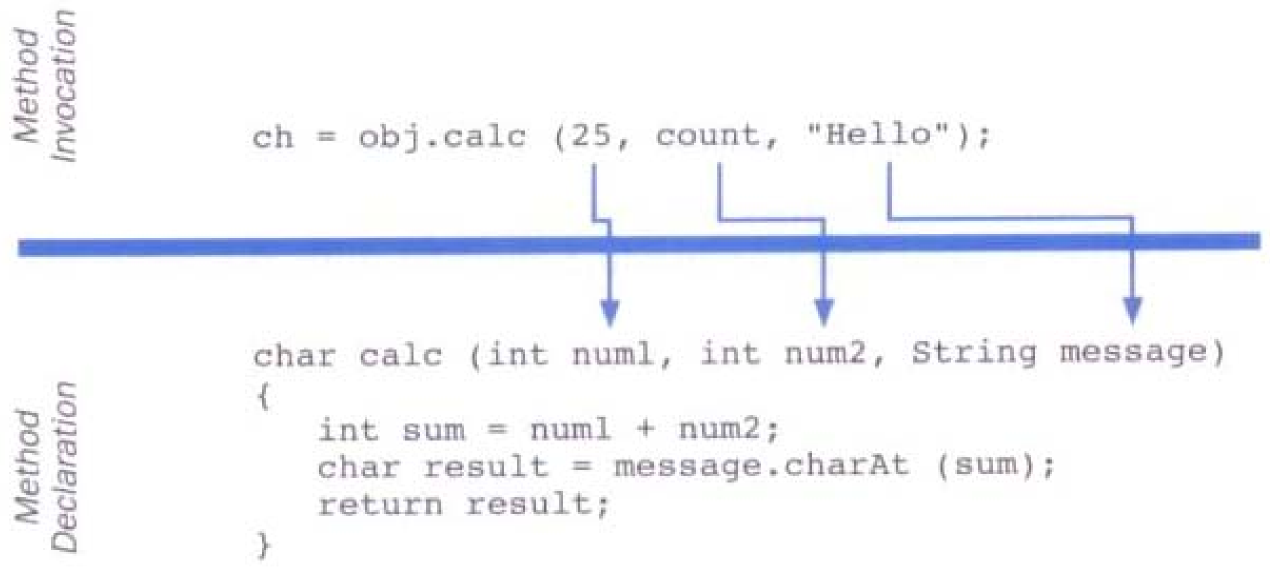
The return Statement


Return Statement

```
return total;
```

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 become *automatic local variables* in the method

- 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

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

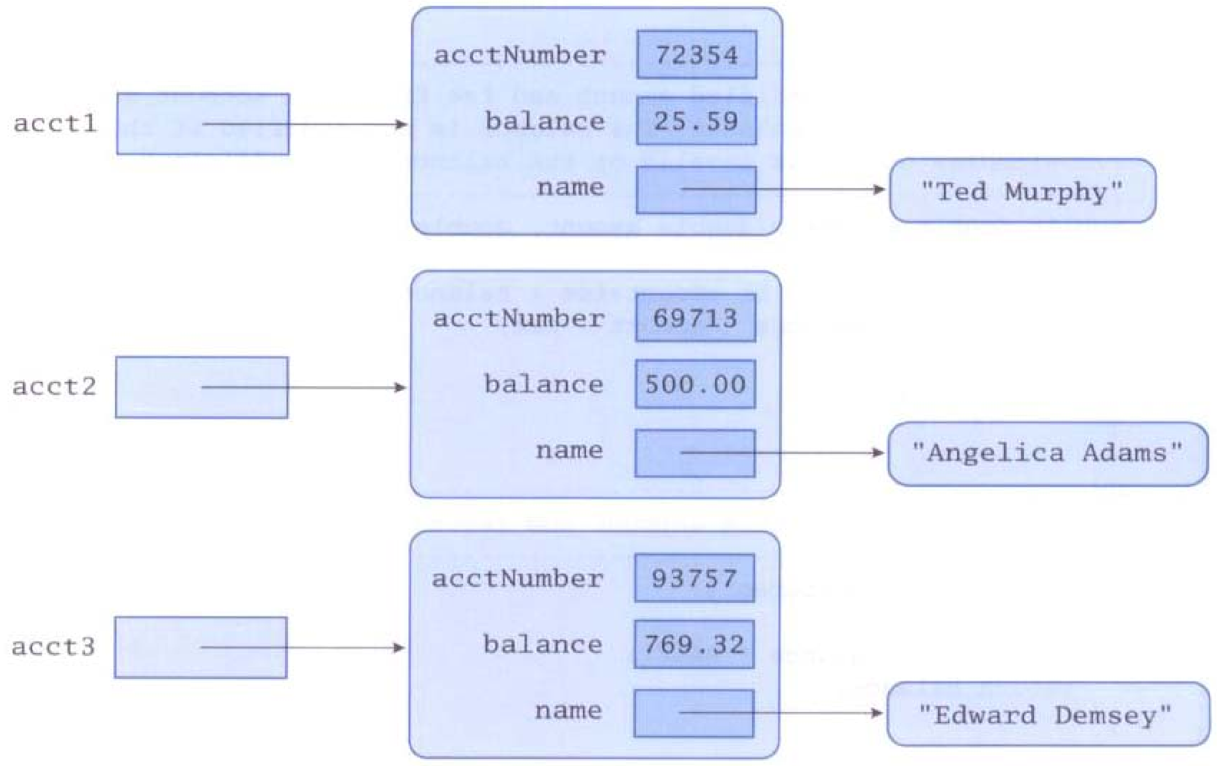
Driver Programs

- 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

```java
//********************************************************************
//  Transactions.java       Java Foundations
//
//  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, 25.59);
      Account acct2 = new Account("Angelica Adams", 69713, 500.00);
      Account acct3 = new Account("Edward Demsey", 93757, 769.32);

      acct1.deposit(44.10);  // return value ignored

      double adamsBalance = acct2.deposit(75.25);
      System.out.println("Adams balance after deposit: " +
                         adamsBalance);

      System.out.println("Adams balance after withdrawal: " +
                         acct2.withdraw (480, 1.50));
      acct3.withdraw(-100.00, 1.50);  // invalid transaction

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

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

Bank Account Example

- The objects just after creation could be depicted as follows:



```java
//********************************************************************
//  Account.java       Java Foundations
//
//  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 String name;
   private long acctNumber;
   private double balance;

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

```java
//-----------------------------------------------------------------
//  Deposits the specified amount into this account and returns
//  the new balance. The balance is not modified if the deposit
//  amount is invalid.
//-----------------------------------------------------------------
public double deposit(double amount)
{
   if (amount > 0)
      balance = balance + amount;

   return balance;
}

//-----------------------------------------------------------------
//  Withdraws the specified amount and fee from this account and
//  returns the new balance. The balance is not modified if the
//  withdraw amount is invalid or the balance is insufficient.
//-----------------------------------------------------------------
public double withdraw(double amount, double fee)
{
   if (amount+fee > 0 && amount+fee < balance)
      balance = balance - amount - fee;

   return balance;
}
```

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

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

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

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

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

Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the `static` modifier

- It associates the method or variable with the class rather than with an object of that class

- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*

- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

```
class Helper
{
   public static int cube (int num)
   {
      return num * num * num;
   }
}
```

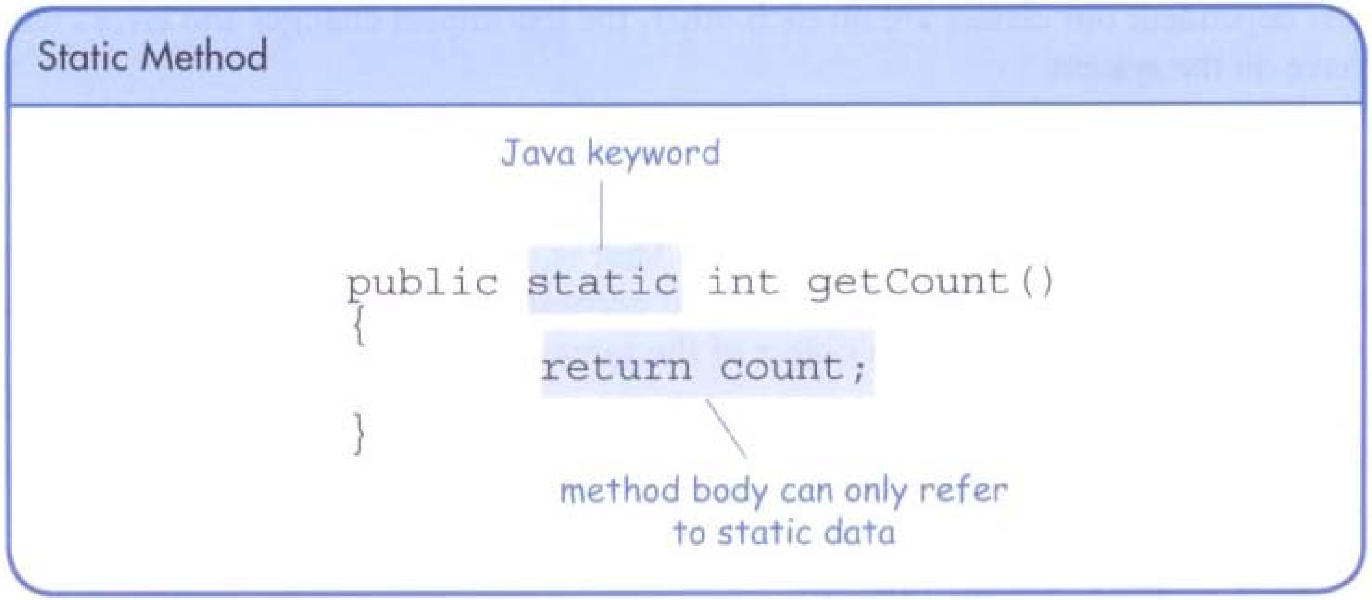
Because it is declared as static, the method can be invoked as

```
value = Helper.cube(5);
```

Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

Static Class Members



```
public static int getCount()
{
        return count;
}
```

Static Class Members

- Static methods and static variables often work together

- The following example keeps track of how many `Slogan` objects have been created using a static variable, and makes that information available using a static method

```java
//********************************************************************
//  SloganCounter.java       Java Foundations
//
//  Demonstrates the use of the static modifier.
//********************************************************************

public class SloganCounter
{
   //-----------------------------------------------------------------
   //  Creates several Slogan objects and prints the number of
   //  objects that were created.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Slogan obj;

      obj = new Slogan("Remember the Alamo.");
      System.out.println(obj);

      obj = new Slogan("Don't Worry. Be Happy.");
      System.out.println(obj);

      obj = new Slogan("Live Free or Die.");
      System.out.println(obj);

      obj = new Slogan("Talk is Cheap.");
      System.out.println(obj);
      obj = new Slogan("Write Once, Run Anywhere.");
      System.out.println(obj);

      System.out.println();
      System.out.println("Slogans created: " + Slogan.getCount());
   }
}
```

```java
//********************************************************************
//  Slogan.java       Java Foundations
//
//  Represents a single slogan or motto.
//********************************************************************

public class Slogan
{
   private String phrase;
   private static int count = 0;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the slogan and increments the number of
   //  instances created.
   //-----------------------------------------------------------------
   public Slogan(String str)
   {
      phrase = str;
      count++;
   }

   //-----------------------------------------------------------------
   //  Returns this slogan as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return phrase;
   }
   //-----------------------------------------------------------------
   //  Returns the number of instances of this class that have been
   //  created.
   //-----------------------------------------------------------------
   public static int getCount()
   {
      return count;
   }
}
```

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
  - Dependency: A *uses* B
  - Aggregation: A *has-a* B
  - Inheritance: A *is-a* B
- Let's discuss dependency and aggregation further
- Inheritance is discussed in detail in Chapter 8

Dependency

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- We've seen dependencies in many previous examples
- We don't want numerous or complex dependencies among classes
- Nor do we want complex classes that don't depend on others
- A good design strikes the right balance

Dependency

- Some dependencies occur between objects of the same class
- A method of the class may accept an object of the same class as a parameter
- For example, the `concat` method of the `String` class takes as a parameter another `String` object

```
str3 = str1.concat(str2);
```

- This drives home the idea that the service is being requested from a particular object

Dependency

- The following example defines a class called `RationalNumber` to represent a rational number

- A rational number is a value that can be represented as the ratio of two integers

- Some methods of the `RationalNumber` class accept another `RationalNumber` object as a parameter

```
//********************************************************************
//  RationalTester.java       Java Foundations
//
//  Driver to exercise the use of multiple Rational objects.
//********************************************************************

public class RationalTester
{
   //-----------------------------------------------------------------
   //  Creates some rational number objects and performs various
   //  operations on them.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      RationalNumber r1 = new RationalNumber(6, 8);
      RationalNumber r2 = new RationalNumber(1, 3);
      RationalNumber r3, r4, r5, r6, r7;

      System.out.println("First rational number: " + r1);
      System.out.println("Second rational number: " + r2);

      if (r1.isLike(r2))
         System.out.println("r1 and r2 are equal.");
      else
         System.out.println("r1 and r2 are NOT equal.");
      r3 = r1.reciprocal();
      System.out.println("The reciprocal of r1 is: " + r3);

      r4 = r1.add(r2);
      r5 = r1.subtract(r2);
      r6 = r1.multiply(r2);
      r7 = r1.divide(r2);

      System.out.println("r1 + r2: " + r4);
      System.out.println("r1 - r2: " + r5);
      System.out.println("r1 * r2: " + r6);
      System.out.println("r1 / r2: " + r7);
   }
}
```

```
//********************************************************************
//  RationalNumber.java       Java Foundations
//
//  Represents one rational number with a numerator and denominator.
//********************************************************************

public class RationalNumber
{
   private int numerator, denominator;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the rational number by ensuring a nonzero
   //  denominator and making only the numerator signed.
   //-----------------------------------------------------------------
   public RationalNumber(int numer, int denom)
   {
      if (denom == 0)
         denom = 1;

      // Make the numerator "store" the sign
      if (denom < 0)
      {
         numer = numer * -1;
         denom = denom * -1;
      }

      numerator = numer;
      denominator = denom;

      reduce();
   }
```

```
//-----------------------------------------------------------------
//  Returns the numerator of this rational number.
//-----------------------------------------------------------------
public int getNumerator()
{
   return numerator;
}

//-----------------------------------------------------------------
//  Returns the denominator of this rational number.
//-----------------------------------------------------------------
public int getDenominator()
{
   return denominator;
}

//-----------------------------------------------------------------
//  Returns the reciprocal of this rational number.
//-----------------------------------------------------------------
public RationalNumber reciprocal()
{
   return new RationalNumber(denominator, numerator);
}
```

```java
//-----------------------------------------------------------------
//  Adds this rational number to the one passed as a parameter.
//  A common denominator is found by multiplying the individual
//  denominators.
//-----------------------------------------------------------------
public RationalNumber add(RationalNumber op2)
{
   int commonDenominator = denominator * op2.getDenominator();
   int numerator1 = numerator * op2.getDenominator();
   int numerator2 = op2.getNumerator() * denominator;
   int sum = numerator1 + numerator2;

   return new RationalNumber(sum, commonDenominator);
}

//-----------------------------------------------------------------
//  Subtracts the rational number passed as a parameter from this
//  rational number.
//-----------------------------------------------------------------
public RationalNumber subtract(RationalNumber op2)
{
   int commonDenominator = denominator * op2.getDenominator();
   int numerator1 = numerator * op2.getDenominator();
   int numerator2 = op2.getNumerator() * denominator;
   int difference = numerator1 - numerator2;

   return new RationalNumber(difference, commonDenominator);
}
```

```
//-----------------------------------------------------------------
//  Multiplies this rational number by the one passed as a
//  parameter.
//-----------------------------------------------------------------
public RationalNumber multiply(RationalNumber op2)
{
   int numer = numerator * op2.getNumerator();
   int denom = denominator * op2.getDenominator();

   return new RationalNumber(numer, denom);
}

//-----------------------------------------------------------------
//  Divides this rational number by the one passed as a parameter
//  by multiplying by the reciprocal of the second rational.
//-----------------------------------------------------------------
public RationalNumber divide (RationalNumber op2)
{
   return multiply(op2.reciprocal());
}

//-----------------------------------------------------------------
//  Determines if this rational number is equal to the one passed
//  as a parameter.  Assumes they are both reduced.
//-----------------------------------------------------------------
public boolean isLike(RationalNumber op2)
{
   return ( numerator == op2.getNumerator() &&
            denominator == op2.getDenominator() );
}
```

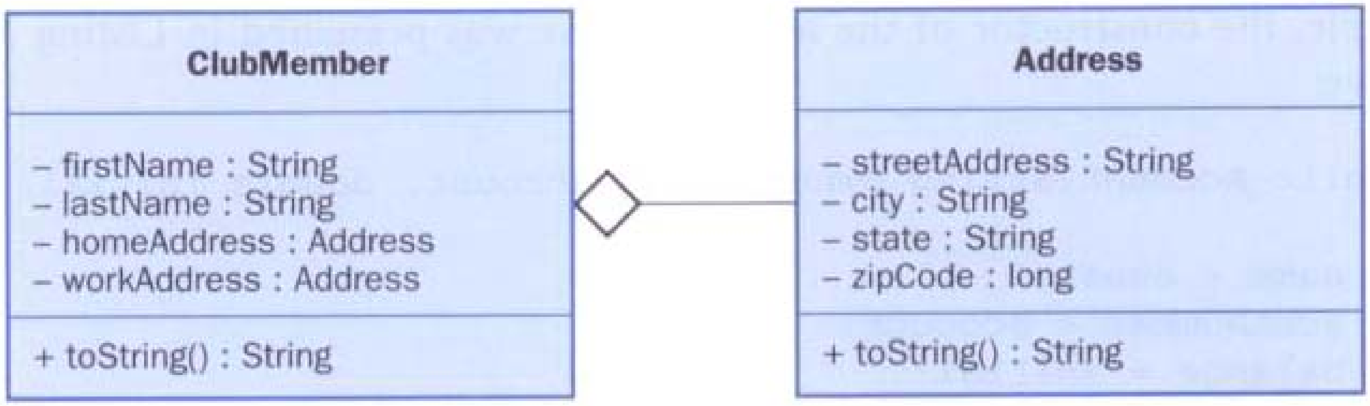
```
//-----------------------------------------------------------------
//  Returns this rational number as a string.
//-----------------------------------------------------------------
public String toString()
{
   String result;

   if (numerator == 0)
      result = "0";
   else
      if (denominator == 1)
         result = numerator + "";
      else
         result = numerator + "/" + denominator;

   return result;
}
```

```
   //-----------------------------------------------------------------
   //  Reduces this rational number by dividing both the numerator
   //  and the denominator by their greatest common divisor.
   //-----------------------------------------------------------------
   private void reduce()
   {
      if (numerator != 0)
      {
         int common = gcd(Math.abs(numerator), denominator);

         numerator = numerator / common;
         denominator = denominator / common;
      }
   }

   //-----------------------------------------------------------------
   //  Computes and returns the greatest common divisor of the two
   //  positive parameters. Uses Euclid's algorithm.
   //-----------------------------------------------------------------
   private int gcd(int num1, int num2)
   {
      while (num1 != num2)
         if (num1 > num2)
            num1 = num1 - num2;
         else
            num2 = num2 - num1;

      return num1;
   }
}
```

Aggregation

- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A car *has a* chassis
- In software, an aggregate object contains references to other objects as instance data
- The aggregate object is defined in part by the objects that make it up
- This is a special kind of dependency – the aggregate usually relies on the objects that compose it

Aggregation in UML



The this Reference

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- In the first invocation, the this reference refers to `obj1`; in the second it refers to `obj2`

The `this` reference

- The `this` reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names

- The constructor of the `Account` class could have been written as follows:

```
public Account (String name, long acctNumber,
                double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

Method Design

- As we've discussed, high-level design issues include:
  - identifying primary classes and objects
  - assigning primary responsibilities
- After establishing high-level design issues, its important to address low-level issues such as the design of key methods
- For some methods, careful planning is needed to make sure they contribute to an efficient and elegant system design

Method Design

- An *algorithm* is a step-by-step process for solving a problem
- Examples: a recipe, travel directions
- Every method implements an algorithm that determines how the method accomplishes its goals
- An algorithm may be expressed in *pseudocode*, a mixture of code statements and English that communicate the steps to take

Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A public service method of an object may call one or more private support methods to help it accomplish its goal
- Support methods might call other support methods if appropriate

Method Decomposition

- Let's look at an example that requires method decomposition – translating English into Pig Latin
- Pig Latin is a language in which each word is modified by moving the initial sound of the word to the end and adding "ay"
- Words that begin with vowels have the "yay" sound added on the end

book	➡	ookbay	table	➡	abletay
item	➡	itemyay	chair	➡	airchay

Method Decomposition

- The primary objective (translating a sentence) is too complicated for one method to accomplish
- Therefore we look for natural ways to decompose the solution into pieces
- Translating a sentence can be decomposed into the process of translating each word
- The process of translating a word can be separated into translating words that
  - begin with vowels
  - begin with consonant blends (sh, cr, th, etc.)
  - begin with single consonants

```java
//********************************************************************
//  PigLatin.java       Java Foundations
//
//  Demonstrates the concept of method decomposition.
//********************************************************************

import java.util.Scanner;

public class PigLatin
{
   //-----------------------------------------------------------------
   //  Reads sentences and translates them into Pig Latin.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String sentence, result, another;

      Scanner scan = new Scanner(System.in);

      do
      {
         System.out.println();
         System.out.println("Enter a sentence (no punctuation):");
         sentence = scan.nextLine();
         System.out.println();
         result = PigLatinTranslator.translate(sentence);
         System.out.println("That sentence in Pig Latin is:");
         System.out.println(result);

         System.out.println();
         System.out.print("Translate another sentence (y/n)? ");
         another = scan.nextLine();
      }
      while (another.equalsIgnoreCase("y"));
   }
}
```

```java
//********************************************************************
//  PigLatinTranslator.java       Java Foundations
//
//  Represents a translator from English to Pig Latin. Demonstrates
//  method decomposition.
//********************************************************************

import java.util.Scanner;

public class PigLatinTranslator
{
   //-----------------------------------------------------------------
   //  Translates a sentence of words into Pig Latin.
   //-----------------------------------------------------------------
   public static String translate(String sentence)
   {
      String result = "";

      sentence = sentence.toLowerCase();

      Scanner scan = new Scanner(sentence);

      while (scan.hasNext())
      {
         result += translateWord(scan.next());
         result += " ";
      }

      return result;
   }
```

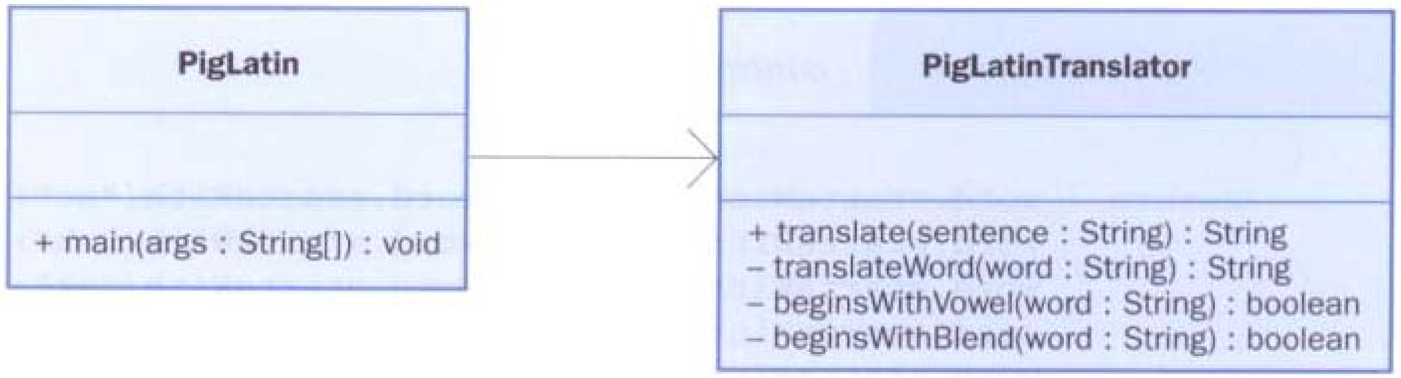
```
//-----------------------------------------------------------------
//  Translates one word into Pig Latin. If the word begins with a
//  vowel, the suffix "yay" is appended to the word.  Otherwise,
//  the first letter or two are moved to the end of the word,
//  and "ay" is appended.
//-----------------------------------------------------------------
private static String translateWord(String word)
{
   String result = "";

   if (beginsWithVowel(word))
      result = word + "yay";
   else
      if (beginsWithBlend(word))
         result = word.substring(2) + word.substring(0,2) + "ay";
      else
         result = word.substring(1) + word.charAt(0) + "ay";

   return result;
}
```

```
//-----------------------------------------------------------------
//  Determines if the specified word begins with a vowel.
//-----------------------------------------------------------------
private static boolean beginsWithVowel(String word)
{
   String vowels = "aeiou";

   char letter = word.charAt(0);

   return (vowels.indexOf(letter) != -1);
}
```

```
   //-----------------------------------------------------------------
   //  Determines if the specified word begins with a particular
   //  two-character consonant blend.
   //-----------------------------------------------------------------
   private static boolean beginsWithBlend(String word)
   {
      return ( word.startsWith ("bl") || word.startsWith ("sc") ||
               word.startsWith ("br") || word.startsWith ("sh") ||
               word.startsWith ("ch") || word.startsWith ("sk") ||
               word.startsWith ("cl") || word.startsWith ("sl") ||
               word.startsWith ("cr") || word.startsWith ("sn") ||
               word.startsWith ("dr") || word.startsWith ("sm") ||
               word.startsWith ("dw") || word.startsWith ("sp") ||
               word.startsWith ("fl") || word.startsWith ("sq") ||
               word.startsWith ("fr") || word.startsWith ("st") ||
               word.startsWith ("gl") || word.startsWith ("sw") ||
               word.startsWith ("gr") || word.startsWith ("th") ||
               word.startsWith ("kl") || word.startsWith ("tr") ||
               word.startsWith ("ph") || word.startsWith ("tw") ||
               word.startsWith ("pl") || word.startsWith ("wh") ||
               word.startsWith ("pr") || word.startsWith ("wr") );
   }
}
```

Method Decomposition

- This example depicted in a UML diagram:


PigLatin
+ main(args : String[]) : void
PigLatinTranslator
+ translate(sentence : String) : String
– translateWord(word : String) : String
– beginsWithVowel(word : String) : boolean
– beginsWithBlend(word : String) : boolean

- Notations can be used to indicate if a method is public (+) or private (-)

Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement

Passing Objects to Methods

- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

- What a method does with a parameter may or may not have a permanent effect (outside the method)

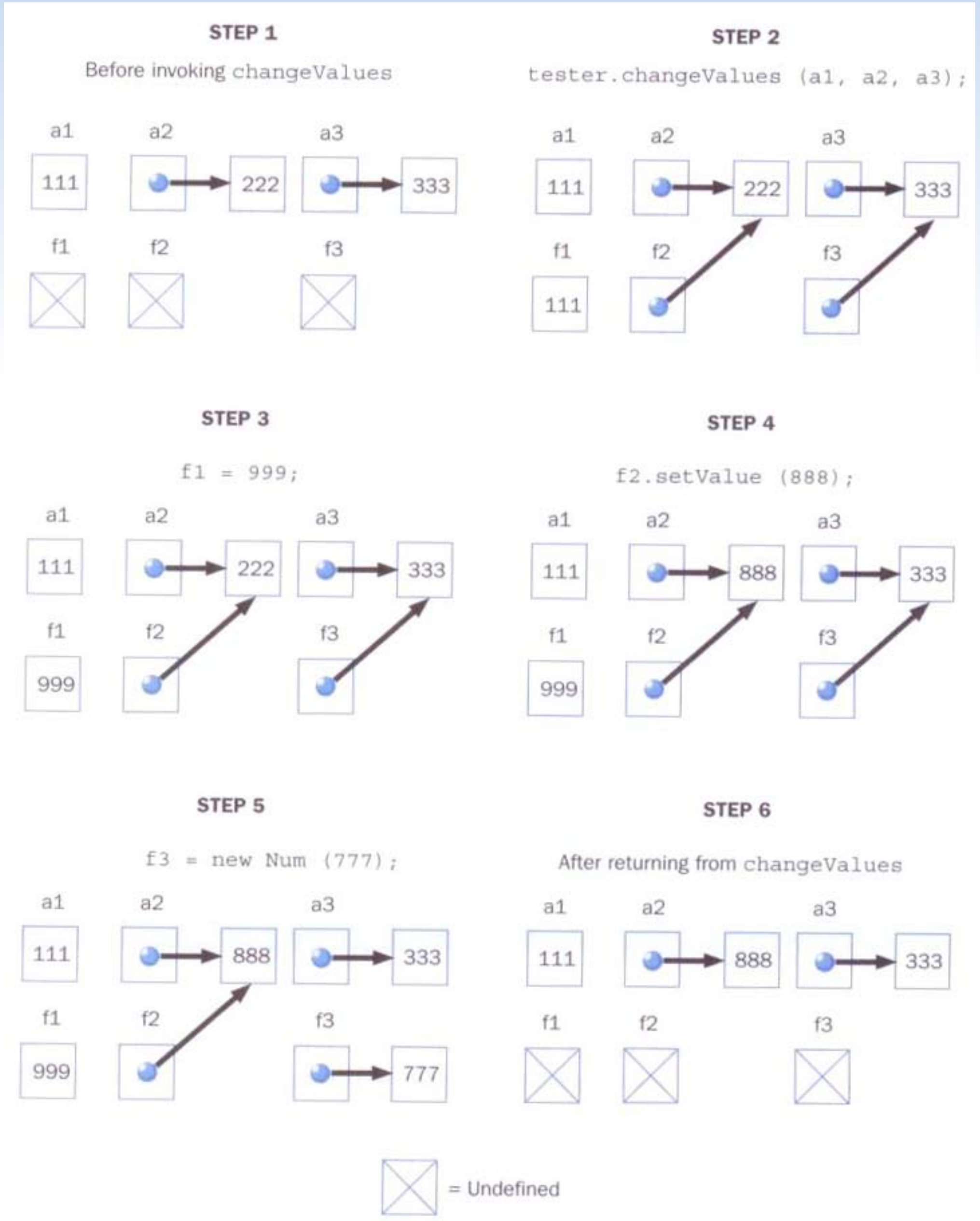
- Note the difference between changing the internal state of an object versus changing which object a reference points to

```java
//********************************************************************
//  ParameterTester.java       Java Foundations
//
//  Demonstrates the effects of passing various types of parameters.
//********************************************************************

public class ParameterTester
{
   //-----------------------------------------------------------------
   //  Sets up three variables (one primitive and two objects) to
   //  serve as actual parameters to the changeValues method. Prints
   //  their values before and after calling the method.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      ParameterModifier modifier = new ParameterModifier();

      int a1 = 111;
      Num a2 = new Num(222);
      Num a3 = new Num(333);

      System.out.println("Before calling changeValues:");
      System.out.println("a1\ta2\ta3");
      System.out.println(a1 + "\t" + a2 + "\t" + a3 + "\n");
      modifier.changeValues(a1, a2, a3);

      System.out.println("After calling changeValues:");
      System.out.println("a1\ta2\ta3");
      System.out.println(a1 + "\t" + a2 + "\t" + a3 + "\n");
   }
}
```

```
//********************************************************************
//  ParameterModifier.java       Java Foundations
//
//  Demonstrates the effects of changing parameter values.
//********************************************************************

public class ParameterModifier
{
   //-----------------------------------------------------------------
   //  Modifies the parameters, printing their values before and
   //  after making the changes.
   //-----------------------------------------------------------------
   public void changeValues(int f1, Num f2, Num f3)
   {
      System.out.println("Before changing the values:");
      System.out.println("f1\tf2\tf3");
      System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");

      f1 = 999;
      f2.setValue(888);
      f3 = new Num(777);

      System.out.println("After changing the values:");
      System.out.println("f1\tf2\tf3");
      System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");
   }
}
```

```
//********************************************************************
//  Num.java       Java Foundations
//
//  Represents a single integer as an object.
//********************************************************************

public class Num
{
   private int value;

   //-----------------------------------------------------------------
   //  Sets up the new Num object, storing an initial value.
   //-----------------------------------------------------------------
   public Num(int update)
   {
      value = update;
   }

   //-----------------------------------------------------------------
   //  Sets the stored value to the newly specified value.
   //-----------------------------------------------------------------
   public void setValue(int update)
   {
      value = update;
   }
   //-----------------------------------------------------------------
   //  Returns the stored integer value as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return value + "";
   }
}
```

- Tracing the parameter values:



Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions

- If a method is overloaded, the method name is not sufficient to determine which method is being called

- The *signature* of each overloaded method must be unique

- The signature includes the number, type, and order of the parameters

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
   return x + .375;
}

float tryMe(int x, float y)
{
   return x * y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```

Method Overloading

- The `println` method is overloaded

```
println(String s)
println(int i)
println(double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

Method Overloading

- The return type of the method is <u>not</u> part of the signature
- That is, overloaded methods cannot differ only by their return type
- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Testing

- *Testing*
  - The act of running a completed program with various inputs to discover problems
  - Any evaluation that is performed by human or machine to asses the quality of the evolving system
- Goal of testing: find errors
- Testing a program can never guarantee the absence of errors

Testing

- Running a program with specific input and producing correct results establishes only that the program works for that particular input
- As more and more test cases execute without revealing errors, confidence in the program rises
- Well-designed test cases are the key to thorough testing
- If an error exists, we determine the cause and fix it
- We should then re-run the previous test cases to ensure we didn’t introduce new errors – *regression testing*

Reviews

- Review – meeting of several people designed to examine a design document or section of code
- Presenting a design or code causes us to think carefully about our work and allows others to provide suggestions
- Goal of a review is to identify problems
- Design review should determine if the system requirements are addressed

Defect Testing

- Testing is also referred to as *defect testing*
- Though we don't want to have errors, they most certainly exist
- A *test case* is a set of inputs, user actions, or initial conditions, and the expected output
- It is not normally feasible to create test cases for all possible inputs
- It is also not normally necessary to test every single situation

Defect Testing

- Two approaches to defect testing
  - *black-box*: treats the thing being tested as a black box
    - Test cases are developed without regard to the internal workings
    - Input data often selected by defining *equivalence categories* – collection of inputs that are expected to produce similar outputs
    - Example: input to a method that computes the square root can be divided into two categories: negative and non-negative

Defect Testing

- Two approaches to defect testing
  - *white-box*: exercises the internal structure and implementation of a method.
    - Test cases are based on the logic of the code under test.
    - Goal is to ensure that every path through a program is executed at least once
    - *Statement coverage* testing – test that maps the possible paths through the code and ensures that the test case causes every path to be executed

Other Testing Types

- *Unit Testing* – creates a test case for each module of code that been authored. The goal is to ensure correctness of individual methods
- *Integration Testing* – modules that were individually tested are now tested as a collection. This form of testing looks at the larger picture and determines if bugs are present when modules are brought together
- *System Testing* – seeks to test the entire software system and how it adheres to the requirements (also known as *alpha* or *beta* tests)

Test Driven Development

- Developers should write test cases as they develop their source code
- Some developers have adopted a style known as *test driven development*
  - test cases are written first
  - only enough source code is implemented such that the test case will pass

Test Driven Development

- Test Driven Development Sequence
  1. Create a test case that tests a specific method that has yet to be completed
  2. Execute all of the tests cases present and verify that all test cases will pass except for the most recently implemented test case
  3. Develop the method that the test case targets so that the test case will pass without errors
  4. Re-execute all of the test cases and verify that every test case passes, including the most recently created test case
  5. Clean up the code to eliminate redundant portions (refactoring)
  6. Repeat the process starting with Step #1

Debugging

- *Debugging* is the act of locating and correcting run-time and logic errors in programs
- Errors can be located in programs in a number of ways
  - you may notice a run-time error (program termination)
  - you may notice a logic error during execution
- Through rigorous testing, we hope to discover all possible errors. However, typically a few errors slip through into the final program
- A *debugger* is a software application that aids us in our debugging efforts

Simple Debugging using `println`

- Simple debugging during execution can involve the use of strategic `println` statements indicating
  - the value of variables and the state of objects at various locations in the code
  - the path of execution, usually performed through a series of "it got here" statements
- Consider the case of calling a method
  - it may be useful to print the value of each parameter after the method starts
  - this is particularly helpful with recursive methods

Debugging Concepts

- Formal debuggers generally allow us to
  - set one or more *breakpoints* in the program. This allows to pause the program at a given point
  - print the value of a variable or object
  - step into or over a method
  - execute the next single statement
  - resume execution of the program