5 Class Methods and Class Variables

Overview

This chapter shows you how to define and use methods and variables that do not need executors, such as the `Vic.say` method introduced in Chapter Two and the `JOptionPane.showInputDialog` method from the Sun standard library:

- Sections 5.1-5.3 describe and illustrate class methods, class variables, and final variables.
- Sections 5.4 and 5.5 develop a full implementation of a Vic simulator, which completes the top-down development of Vic software (first see how to use the methods, then learn how to define them). It uses two new String methods, `substring` and `length`. If you want to move on quickly to the material in later chapters, you may postpone or skip everything in Chapter Five after Section 5.4.
- Sections 5.6-5.8 describe software for working with Networks. The Network software illustrates the use of for-statements, class methods, class variables, and final variables. It does not involve any new language features.
- Section 5.9 explains and illustrates the use of recursion. This just means that you may have a method call itself as long as you avoid execution that never stops. The first half of the section is independent of Networks.

5.1 Defining Class Methods

A method to find an integer average of several integer values would be quite useful. If the `sum` of a group of 5 numbers is 49, the average is best approximated as 10, but if their `sum` is 46, the average is best approximated as 9. You cannot calculate this average as `sum / count`, because you would get 9 in either case; division of int values drops the fractional part. You could simply add half of the count before you divide and then have the method return that result:

```
return (sum + count / 2) / count;
```

This works for positive numbers. But if you try this when the `sum` is negative, it gives the wrong answer: `-49 + 5 / 2` is -47 and `-47 / 5` is -9; `-46 + 5 / 2` is -44 and `-44 / 5` is -8. Some more thought leads you to the correct answer, as follows:

```
public int average (int sum, int count)
{  if (sum >= 0)
      return (sum + count / 2) / count;
   else
      return (sum - count / 2) / count;
}  //======================
```

The `average` method now computes the correct value: `average(-49, 5)` returns the number -10 and `average(-46,5)` returns the number -9. However, something feels wrong here. This is an instance method, but there is no instance to act as the executor. The method deals with numbers only, no objects at all. You would have to create an object of the class to which the method belongs before you could use the `average` method, and then the object would be irrelevant to the calculation.

Class methods

Java provides a mechanism to handle such situations: The word `static` in the method heading means that it is a class method, i.e., you may call it with the name of the class in place of the executor. In such a method you cannot use `this`, either explicitly or by default, since there is of course no executor that `this` refers to. So the above method should have its heading begin `public static int average`.

Several different calculations come up from time to time involving numbers alone, so it is useful to have an entire class containing such utility methods. We will call it MathOp. So `x = MathOp.average(49,5)` stores 10 in `x`. Listing 5.1 describes the MathOp class, with the `average` method and another useful method for raising a number to a power of 2: `MathOp.powerOf2(5)` returns the number 32 and `MathOp.powerOf2(10)` returns the number 1024. Note that `powerOf2` illustrates a for-statement that does not have any initializer part (because `expo` is already initialized). It multiplies `power` by 2 once for each time through the loop.

Listing 5.1 The MathOp utilities class

```
public class MathOp
{
   /** Return the value of 2 to the power expo. */

   public static int powerOf2 (int expo)
   {  int power = 1;
      for (;  expo > 0;  expo--)
         power = power * 2;
      return power;
   }  //======================

   /** Return sum divided by count rounded to the nearest int. */

   public static int average (int sum, int count)
   {  if (sum >= 0)
         return (sum + count / 2) / count;
      else
         return (sum - count / 2) / count;
   }  //======================
}
```

Both of these MathOp methods could cause the program to fail in some situations. These defects are corrected in the exercises. Can you see what those situations are without peeking ahead to the exercises?

Utilities classes

We call a class with no instance methods or instance variables or main method a **utilities class**. MathOp is an example, and you will see more later. Since all of its methods are class methods, there is no point in creating objects of MathOp type. Some people call such a class a non-instantiable class. You could think of "MathOp" as being an "operative", a person who carries out certain math-related tasks for you. It does not have to be constructed because it has no state (i.e., instance variables that store information).

It is of course possible for outside classes to create MathOp objects (though pointless). Since MathOp does not define a constructor explicitly, outside classes can use `new MathOp()` for the default constructor supplied by the compiler. You can prevent this by adding a MathOp constructor and declaring it `private`, as in the following:

```
private MathOp()
{  super();
}  //=======================
```

Now the default constructor does not exist, since the compiler supplies it only when no other constructor is defined. And the actual constructor is not visible to outside classes, since it is private. Some people feel it is a good idea to do this. Reminder: Marking a constructor or other method as `private` means your coding inside its class can mention it, but no outside class can mention it.

Four categories of methods

The phrase that precedes the return type (or void) in a method's heading tells which of several categories it is in (X denotes the class in which the method is defined):

- `public`: Callable from any class with an executor of class X.
- `private`: Callable only within class X with an executor of class X.
- `public static`: Callable from any class with X in place of the executor.
- `private static`: Callable only within class X with X in place of the executor.

If you call the method from within class X, you may omit the executor or class name before the dot -- it defaults to the executor of the method it is in or to the class it is in, respectively. A class method can be called with an executor if you wish, a variable that can refer to an object of the class, but there is no advantage in doing this.

Independent class methods

The `average` and `powerOf2` methods in Listing 5.1 could be put in any classes at all and they would work the same. They are **independent class methods**. Independent class methods could also be called utility methods. The `Vic.say` and `Vic.reset` introduced in Chapter Two are also class methods. They are defined in the Vic class. A key difference between them and the `MathOp.average` and `MathOp.powerOf2` methods is that the Vic methods can only be in the Vic class. This is because the Vic class methods access private parts of the Vic class that you cannot otherwise get to. So those two Vic class methods are not independent class methods.

Language elements
A declaration of a method can have the word static before its return type.
Such a method can be called using the class name in place of an executor.

Exercise 5.1 The `average` method causes the program to fail if `count` is zero, since division by zero does not make sense. Modify the method to return zero in such cases.
Exercise 5.2 The `powerOf2` method returns 1 whenever `expo` is negative, which is okay. But it produces the wrong answer if `expo` is more than 30, because the largest possible int value is $2^{31} - 1$. Modify the method to return 2^{30} in such cases.
Exercise 5.3 (harder) Write a method `public static int power (int base, int expo)` for MathOp: It returns `base` to the `expo` power. Return -1 in all cases in which the power cannot be computed using int values. But return zero if either parameter is negative. Hint: The largest int value is 2,147,483,647.
Exercise 5.4* Write a method `public static int gcd (int one, int two)` for MathOp: It returns the greatest common positive divisor of its two int parameters. Hint: If either is negative, multiply it by -1; then repeatedly replace the larger by the remainder from dividing the larger by the smaller until one goes evenly into the other, in which case that one will be the greatest common divisor. What do you do if either is zero?
Exercise 5.5* Under what circumstances can a call of a class method be polymorphic?
Exercise 5.6** Write a MathOp class method that finds the factorial of a given int value (e.g., `6!` is `6 * 5 * 4 * 3 * 2 * 1`). Watch out for negatives.

5.2 Declaring Class Variables; Encapsulation

You could put the following variable declaration in the Person class of Listing 4.4, outside of every method definition. This declaration means that the variable named `theNumPersons` is initially zero (i.e., when the program starts), though it is expected to change as the program executes:

```
   public static int theNumPersons = 0;
```

A local variable is declared inside a method definition. You can only use it inside that one method definition. Since `theNumPersons` is not a local variable, you can use it everywhere. The word `static` means that, if you mention it outside of the Person class, you may refer to it as `Person.theNumPersons`, i.e., with the name of the class. So it is called a **class variable**, analogous to a class method. That way, (a) the compiler knows where to look for its declaration, and (b) you can declare a different variable named `theNumPersons` in any other class if you want. Note that this is the same access rule Java has for class methods.

A **field variable** is any variable declared outside any method definition. If it is declared with the word `static`, it is a class variable (e.g., `theNumPersons` just described). The class itself contains the only copy of class variables such as `theNumPersons`.

If a field variable is not declared using the `static` keyword, it is an instance variable (e.g., `itsFirstName` for the Person class). Each instance (object) of a class contains its own separate copy of the instance variables. For example, if `sam` and `sue` are Person variables referring to Person objects, then `sam.itsFirstName` is a completely different variable from `sue.itsFirstname` but `sam.theNumPersons` and `sue.theNumPersons` are the same variable.

In general, use an instance variable to store information about an individual object, but use a class variable to store information about the class as a whole (information that is not specific to an individual instance).

You could add the statement `theNumPersons++;` to the constructor `public Person()`. Then `theNumPersons` keeps track of the number of completely new Person objects that have been created so far. Any outside class can find out from the Person class how many it has made by looking at the value of `Person.theNumPersons`.

Unfortunately, no outside class can trust that `Person.theNumPersons` truly does tell the number of Persons that have been made. After all, any other outside class could underhandedly change the value of `Person.theNumPersons`, perhaps doubling it to fool the other classes.

Problem: This way of making information available makes the information worthless.
Solution: Encapsulation, also known as information-hiding.

Encapsulation

Encapsulation basically requires that a class not let outside classes change its variables. So field variables should generally be declared as private. That way nothing outside of the class that owns the variable can sneak in and change the state of the variable without the owner class knowing about it. We will add to the Person class the following declaration of `theNumPersons` instead of the one given earlier:

```
   private static int theNumPersons = 0;
```

You can still make available to outside classes the value of the variable `theNumPersons` by having the following class method in the Person class:

```
public static int getNumPersons()
{  return theNumPersons;
}  //======================
```

Any outside class can then use `Person.getNumPersons()` to get the value of `theNumPersons`. But no outside class can change its value. In general, any method or field variable that does not need an executor should be declared as a class method.

Failure to encapsulate was the most pernicious cause of bugs in programs in the early decades of programming. With encapsulation, any outside classes that modify these field variables must go through the class's methods to do so, if then. This makes bugs less likely. Listing 5.2 (see next page) contains the complete Person class as revised from Listing 4.4.

In the Nim class of Listing 4.8, each game object should have its own individual value of `itsNumLeft` and `itsMaxToTake`, but they can all share the same random number generator. So it could be declared outside of every method as follows:

```
private static java.util.Random randy
                       = new java.util.Random();
```

Initial values of class variables

A class variable exists independently of any instance of the class. You should almost always give it an initial value where it is declared. If the initial value is not given there, the compiler gives it a **default initial value**: zero for a numeric variable, null for an object variable, and false for a boolean variable.

Initializations of class variables at runtime are done when the class is first loaded, before any instances of the class are created, and in the order they are listed in the class definition. So the initialization of one class variable should not refer to the value of a class variable declared later in the class definition.

Programming Style It is good style to explicitly state in your classes the initial value of a class variable when your logic requires it to have one, rather than relying on the default value. Note that Listing 5.2 does this. That way a reader of the class does not have to stop and think what the default value is.

Typical structure of an information facility

Listing 5.2 illustrates a very common way of equipping a class to provide certain information about itself. In this case, the information to be provided is the number of Person objects created so far in the program. Listing 5.2 has three relevant parts:

(a) A class query method so that others can access the information, e.g., the method call `Person.getNumPersons()`. This kind of method usually just returns the value of a class variable (`theNumPersons` in this case).
(b) The declaration of the class variable with its initial value specified.
(c) Statements to update the value of the class variable in each method that modifies the information to be provided (only the constructor in Listing 5.2).

Listing 5.2 The Person class of objects

```
public class Person extends Object
{
   private static int theNumPersons = 0;  // initialize num
   private String itsFirstName;
   private String itsLastName;
   private int itsBirthYear;

   public Person (String first, String last, int year)
   {  super();
      theNumPersons++;                    // update num
      itsFirstName = first;
      itsLastName = last;                 // initialize last name
      itsBirthYear = year;
   }  //=======================

   /** Tell how many different Persons exist. */

   public static int getNumPersons()      // access num
   {  return theNumPersons;
   }  //======================

   /** Return the birth year. */

   public int getBirthYear()
   {  return itsBirthYear;
   }  //=======================

   /** Return the first name. */

   public String getFirstName()
   {  return itsFirstName;
   }  //=======================

   /** Return the last name. */

   public String getLastName()            // access last name
   {  return itsLastName;
   }  //=======================

   /** Replace the last name by the specified value. */

   public void setLastName (String name)  // update last name
   {  itsLastName = name;
   }  //=======================
}
```

You have seen a similar structure for equipping an object to provide certain information about itself. An example is the last name of a Person. Listing 5.2 has three relevant parts:

(a) An instance query method so that others can access the information, e.g., the method call `sue.getLastName()`. This kind of method usually just returns the value of an instance variable (`itsLastName` in this case).
(b) The declaration of the instance variable, sometimes with its initial value specified. However, the initial value may be specified in the constructors instead (as is done for `itsLastName` in Listing 5.2).
(c) Statements to update the value of the instance variable in each instance method that modifies the information to be provided, e.g., `sue.setLastName("Jones")`.

Scope and positioning of a field variable declaration

The declaration of an instance variable or class variable can be put anywhere in the class. Its position in the listing does not affect where it can be used within methods. Some people like to put all instance variables at the end of the class, and some like to put them at the beginning. The **scope of a variable** is where it can be used without being directly preceded by a dot and an object or class reference:

- The scope of a class variable is its entire class.
- The scope of an instance variable is all instance methods and constructors in its class.
- The scope of a formal parameter is its method's body. (Reminder: Variables declared in a method heading are called formal parameters; the corresponding values in the parentheses of a method call are called arguments).
- The scope of a variable declared in the initializer part of a for-statement is only the entire for-statement.
- The scope of any other local variable (declared in a method) is from the point where it is declared to the end of the innermost pair of braces within which it is declared.

Similarly, the placement of methods within the class definition does not affect how they are called within other methods. But if one method calls another in the class, most people find it easier to understand if the method doing the calling comes before the method being called.

Caution Do not use the words private or public inside a method definition. If you make this mistake, the message that the compiler gives you can be baffling. Local variables are by nature private, since you can only mention them inside their methods. But only class variables or instance variables are explicitly stated to be public or private.

Technical note If you declare a variable in e.g. the third statement of a method, you are not allowed to refer to that variable in the first two statements of that method. It is as if the creation of the variable does not occur until that third statement. However, the bytecode that the compiler produces creates all of the local variables of a method when the method begins execution, regardless of where they are declared. The point of declaration does not determine when the variable is created at runtime, it only determines where the variable can be used.

Language elements
A variable declaration that is outside of any method can have the word static before its type. Such a variable can be used with the class name in place of its executor.

Exercise 5.7 Change the Person class of Listing 5.2 so that any outside class can find out the first name of the Person who was most recently created. Use "none so far" for the answer if no Persons have yet been created.
Exercise 5.8 (harder) Write a method `public static int range (int one, int two)` for the MathOp class in the earlier Listing 5.1: The method returns an integer chosen at random within the range of the two parameters (i.e., between or equal to the two parameters). Use a Random class variable. Allow for `one` being greater than `two`.
Exercise 5.9* Change the Person class of Listing 5.2 so that any outside class can find out the smallest birth year of all the Persons who have been created so far. Use -1 for the answer if no Persons have yet been created.
Exercise 5.10* Change the Person class of Listing 5.2 so that any outside class can find out the average birth year of all the Persons who have been created so far. Use 0 for the answer if no Persons have yet been created.

5.3 Final Local, Instance, And Class Variables

If a variable declaration has the word `final` immediately before the name of the type, the compiler will not let any statement change the value any time after you give it its initial value. Such variables are called constants in most programming languages, but Java programmers tend to call them **final variables**.

Final class variables

The Time class constructor in Listing 4.5 has the phrase `itsMin = itsMin + 60` in one method. Other methods would also mention 60, since that is the number of minutes in an hour. The logic of the Time class would be clearer if you declare a final class variable with the value 60 and use it instead. By convention, we write the variable name all in capital letters if it is a final class variable. So the declaration in the Time class, and the corresponding change in the Time constructor, could be as follows:

```
   public static final int MIN_PER_HOUR = 60;  // class variable
   itsMin = itsMin + MIN_PER_HOUR;
```

The BasicGame class in Listing 4.3 has an instance variable `itsSecretWord`. Every BasicGame object has exactly the same secret word, though they may have different user's words (depending on what the user chose for that game). It would make more sense to have just one copy of this value for the whole class, rather than one for each BasicGame object. Since the value of this variable never changes, you could write the class variable declaration in the BasicGame class, and the revised statement in the `shouldContinue` method, as follows:

```
   public static final String SECRET = "duck";  // class variable
   return ! SECRET.equals (itsUsersWord);
```

Programming Style Any constant value that is used in two or more methods in a class should be declared as a final class variable and that variable used in place of the constant value. The name of the variable should be all in capital letters. It may or may not be public. Some exceptions: 2, 1, 0, and "".

This book normally lists field variables in the order `public static final`, then `private static`, and then the instance variables. This book also puts a prefix of "the" on almost all names of non-final class variables, and never anywhere else. This hallmark, along with the prefix "its" on almost all names of instance variables, makes bugs less likely. If you do not do this in your own definitions, at least obey the following safety principle: Never name a parameter or local variable the same as a class variable or an instance variable.

Final instance variables and final local variables

In Listing 5.2, no provision is made for changing two of the values of the Person instance variables once they have been assigned. This should be made clear in the declarations:

```
   private final String itsFirstName;
   private final int itsBirthYear;
```

They could be made publicly visible if there is a good reason to do so, because that would not violate the encapsulation principle: No outside class could change the value of any instance variable of any Person object. But as it is, we have the getXXX methods to retrieve any one of the values, so we need not make them public. In general, it is preferable to access even final instance variables through getXXX methods rather than directly -- it makes it easier to upgrade the software in the future.

The value of a final instance variable must be assigned in the declaration or else in every constructor of the class. You can see that the Person class does the latter, so the addition of the word `final` is the only change needed. When all of an object's instance variables are final, the object is said to be **immutable** (because you cannot change its attributes once it has been created). String values, for instance, are immutable.

You may declare a variable that is local to a method as `final`, in which case you should immediately assign its final value at the point where you declare it. This should usually be done if you use a constant value in two or more places within the method. Some people like to make the name all in capital letters; others reserve that for class variables.

Some people go so far as to say that every constant value used in a method should be a named final variable, other than perhaps 0, 1, and 2 and the empty String. This book does not go that far. That principle would require that every string literal be named.

Language elements
A variable declaration can have the word final before its type.
The value of such a variable cannot be changed once it has been assigned.

Exercise 5.11 Rewrite Listing 4.5 to store 10 in a named local final variable, then use it wherever it is appropriate.
Exercise 5.12* Rewrite Listing 4.6 to store both 1 and 100 in named public final variables standing for the lower and upper limits, then use them wherever appropriate.
Exercise 5.13* Rewrite the Nim constructor in Listing 4.8 to use local final variables for the numbers 21 and 3, then use them wherever it is appropriate.

5.4 Two New String Methods

You now have enough background to understand a complete simulation of the Vic's Programmable CD Organizer (described in Chapters 2 and 3). Of course, no one can write the actual program in Java; since it moves armatures and gears and springs, the Vic engineers have to write it in native code. If you were to look at their implementation, you would see almost all the methods with the notation `native` and no method bodies.

The Vic simulation developed in these two sections will not involve graphics; it is too early for that. But the simulation will produce **tracing output** to the terminal window such that, each time one of the four action instance methods is executed, you will see a full description of that sequence of slots. The simulation makes heavy use of class variables and final variables. And it introduces two new String methods from the standard library.

The data structure

Each sequence of slots is represented by a string of non-blank characters, where 0 signals an empty slot and any other character signals a slot that contains a CD whose name is that character. The instance variables are `itsSequence` (the String), `itsPos` (a position in the sequence) and `itsID` (a positive int). So `getPosition` only needs one statement to report a string of characters containing the current position and ID:

```
   return itsPos + "," + itsID;
```

The value of `itsID` is also used in the tracing output, to tell you which Vic object is being described. The `trace` method each action instance method calls has the following single statement. The method call `System.out.println` has a single String parameter that it prints to the terminal window:

```
   System.out.println ("Vic# " + itsID + ": " + action
         + itsPos + "; sequence= " + itsSequence);
```

The `Vic.say` method can use this output statement as well, having just one statement:

```
System.out.println ("SAYS: " + message);
```

The `backUp` method first checks that `itsPos` is greater than 1, since otherwise the program is to terminate immediately. Then it decrements `itsPos` and calls the trace method in an obvious way. The logic for `backUp` and the three one-liner Vic methods discussed so far is in the upper part of Listing 5.3 (see next page). `System.exit(0)` is not a graceful way of terminating; adding an explanatory message is an exercise.

String methods

Before you go further, you need to know about the two methods in the String class that this software uses. One is the `length` method: `s.length()` returns the number of characters in the String `s`. The other is the `substring` method, which returns a new String value that is a portion of the executor. For instance, `s.substring(2,4)` returns the String value consisting of the characters numbered starting from 2 and going up to but not including 4. That is, you get the characters numbered 2 and 3, in that order. Figure 5.1 describes the two new String methods.

s.length()
is the number of characters in the String object referred to by s.

s.substring (start, end)
returns a new String object consisting of the characters of s in positions start through end-1. The program can crash unless 0 <= start <= end <= s.length().

Figure 5.1 Two String methods

You also need to know that Java numbering of the positions in a String is **zero-based**: The first character is numbered 0, the second is numbered 1, the third is numbered 2, etc. Therefore, `"abcdef".substring(2,5)` is the string `"cde"`. This all means that the `seesSlot` method can be coded as just one statement, as follows.

```
return itsPos < itsSequence.length();
```

For instance, if `itsSequence` has length 6, and thus numbers the characters 0 through 5, `itsPos` is beyond the end of the sequence of slots if `itsPos` is 6. For the `seesCD` method, you must look at the substring consisting of one character starting at position `itsPos`, which is expressed in Java as follows:

```
itsSequence.substring (itsPos, itsPos + 1);
```

In Listing 5.3, a named constant String value NONE represents the absence of a CD, thus has the value `"0"`. The coding for `moveOn` parallels `backUp`.

Chaining

If a method call returns an object value, you may use the method call for the executor of another method call. This is called **chaining** of method calls, or sometimes cascading. For instance, the following statements are legal:

```
// replace spot.equals (getPosition()) by:
getPosition().equals (spot);
// replace the last two statements of seesCD by:
return ! itsSequence.substring(itsPos,itsPos+1).equals(NONE);
// replace the first three statements in Listing 4.1 by:
new BasicGame().playManyGames();
```

Listing 5.3 The Vic class of objects, part 1 of a String-based simulation

```
public class Vic extends Object
{
   private static final String NONE = "0";
   /////////////////////////////////
   private String itsSequence = "";
   private int itsPos = 1;
   private final int itsID;  // assigned by the constructor


   public String getPosition()
   {  return itsPos + "," + itsID;
   }  //======================

   public static void say (String message)
   {  System.out.println ("SAYS: " + message);
   }  //======================

   private void trace (String action)
   {  System.out.println ("Vic# " + itsID + ": " + action
            + itsPos + "; sequence= " + itsSequence);
   }  //======================

   public void backUp()
   {  if (itsPos == 1)
         System.exit (0);
      itsPos--;
      trace ("backUp to slot ");
   }  //======================

   public void moveOn()
   {  if ( ! seesSlot())
         System.exit (0);
      itsPos++;
      trace ("moveOn to slot ");
   }  //======================

   public boolean seesSlot()
   {  return itsPos < itsSequence.length();
   }  //======================

   public boolean seesCD()
   {  if ( ! seesSlot())
         System.exit (0);
      String s = itsSequence.substring (itsPos, itsPos + 1);
      return ! s.equals (NONE);
   }  //======================
}
```

Exercise 5.14 Write a private class method in the Vic class that has a String parameter and an int parameter and returns the one-character substring at the given position. Then replace more complex coding by a call of that method in Listings 5.3, 5.4, and 5.5.
Exercise 5.15* An attempt to `moveOn` or `backUp` or to evaluate `seesCD` when it is illegal causes an abrupt `System.exit(0)` without explanation. The user would appreciate a tracing output in such cases. Revise these three methods to call a private method that explains the problem (with `showMessageDialog`) and then terminates.

Part B Enrichment And Reinforcement

5.5 Complete String Implementation Of A Vic Simulator

The stack operations

Since Vics all share the same stack, we begin by declaring a class variable `theStack`, initially an empty string of characters. The `stackHasCD` class method then tells whether the string is not empty, i.e., it tells whether `theStack.length()` is positive.

The coding for `takeCD` first checks that `seesSlot()` is `true`; if not, the program terminates. If `seesCD()` is `false`, nothing happens, otherwise `theStack` appends the substring `itsSequence.substring(itsPos, itsPos+1)`.

Next, `itsSequence` puts NONE in place of that substring. The way that `itsSequence` puts NONE at position `itsPos` is to make the new value of `itsSequence` be (a) the substring from position 0 up to `itsPos`, followed by (b) NONE, followed by (c) the substring running from position `itsPos + 1` up to the end. The last thing that `takeCD` does is call the `trace` method.

The upper part of Listing 5.4 (see next page) contains the Java coding for the `stackHasCD` method and the `takeCD` method just described. The logic for the `putCD` method is rather more complex; a reasonable plan is in the accompanying design block.

```
STRUCTURED NATURAL LANGUAGE DESIGN for putCD
1.  Exit the program if there is no slot at the current position.
2.  If the current slot does not have a CD and the stack does, then...
        2a. Change itsSequence to be a new string consisting of three parts:
                (a) all its characters up to but not including the current position;
                (b) the top value on the stack;
                (c) all its characters after the current position.
        2b. Remove the top value from the stack.
3.  Print out a tracing message.
```

The implementation of `putCD` in the lower part of Listing 5.4 differs somewhat from the design. The first thing the coding does is test `! seesCD()`, which will exit the program immediately if there is no slot at the current location. So Step 1 (testing `seesSlot()`) does not have to be coded explicitly.

The reset method

The value passed in to the `reset` method has the type description `String[]`, which has not yet been discussed in this book. A full explanation of these array variables has to wait until Chapter Seven, but the foretaste you get here should be manageable.

The Vic class has a class variable named `theTableau` where it keeps what are initially the three empty String values it allows. The type description of this variable is `String[]`, which means it can hold several String values. The class variable `theTableau` can be initialized to be three empty Strings with this coding:

```
   private static String[] theTableau = { "", "", "" };
```

Listing 5.4 The Vic class of objects: the parts involving theStack

```
// public class Vic continued: using the stack

   private static String theStack = "";   // initially empty


   public static boolean stackHasCD()
   {  return theStack.length() > 0;
   }  //======================


   public void takeCD()
   {  if (seesCD())
      {  theStack = theStack
                  + itsSequence.substring (itsPos, itsPos + 1);
         itsSequence = itsSequence.substring (0, itsPos) + NONE
                  + itsSequence.substring (itsPos + 1,
                                 itsSequence.length());
      }
      trace ("takeCD at slot ");
   }  //======================


   public void putCD()
   {  if ( ! seesCD() && stackHasCD())
      {  int atEnd = theStack.length() - 1;
         itsSequence = itsSequence.substring (0, itsPos)
                     + theStack.substring (atEnd, atEnd + 1)
                     + itsSequence.substring (itsPos + 1,
                                 itsSequence.length());
         theStack = theStack.substring (0, atEnd);
      }
      trace ("putCD  at slot ");
   }  //======================
```

The `reset` method simply assigns its `args` parameter to this `theTableau` variable, replacing the current value of three empty Strings by however many non-empty Strings the user gives as input in `args`. However, in accordance with the specifications for `reset` in Chapter Two, it does not make the assignment if `args` has no strings at all or if some Vic object has already been constructed.

The class variable `theNumVics` keeps track of the number of Vic objects created so far. This is done just as you saw for `theNumPersons` in the earlier Listing 5.2: Initialize it to zero in the declaration, then increment it in the constructor. This variable is used to determine the `itsID` value for each Vic, and it is also used by the `reset` method: If `theNumVics` is not zero, the `reset` method has no effect.

Each object whose type is `String[]` has a public final instance variable named `length` that you can test to find out how many elements are in the array. So the `reset` method makes sure that `args.length` is positive before it replaces `theTableau` by `args`. The full coding for the `reset` method is therefore as follows:

```
   public static void reset (String[ ] args)
   {  if (theNumVics == 0 && args.length > 0)
         theTableau = args;
   }  //======================
```

The Vic constructor

We saved the hardest method for last. To help the constructor do its job, we have two class variables: One is a random number generator and the other is a named constant that contains the first 12 letters of the alphabet. These letters are at positions 1 through 12 of the LETTERS string, with the character at position 0 left blank. The value of `itsPos` will always be at least 1 for any Vic object. In effect, we switch to a one-based numbering of characters in a String in spite of Java's preference for zero-based. You should compare the following narrative development of the constructor with its coding in Listing 5.5, where each line of the method is numbered.

The first thing the Vic constructor does is create the object (line 1) and then check that `theNumVics` is less than the number of strings in `theTableau` (which will be three unless they were replaced by the `reset` method, in which case it could be dozens, depending on the user's choice; see line 2). If `theNumVics` is too large, we already have all the objects we are allowed, so we do nothing but increment `theNumVics` (line 15), set `itsID` to that value (line 16), and print a tracing message (line 17). These three actions are what we do at the end of the construction process for regular Vics too. So the question is, what do we do if `theNumVics` is not too large?

Listing 5.5 The Vic class of objects, completed

```
// public class Vic completed: constructor and reset

   private static final java.util.Random randy
                      = new java.util.Random();
   private static final String LETTERS = " abcdefghijkl";
   private static int theNumVics = 0;
   private static String[ ] theTableau = {"", "", ""};


   public static void reset (String[ ] args)
   {  if (theNumVics == 0 && args.length > 0)
         theTableau = args;
   }  //======================


   public Vic()
   {  super();                                               // 1
      if (theNumVics < theTableau.length)                    // 2
      {  itsSequence = theTableau[theNumVics];               // 3
         if (itsSequence.length() == 0)                      // 4
         {  for (int k = 3 + randy.nextInt (6); k >= 1; k--) // 5
            {  if (randy.nextInt (2) == 0)                   // 6
                  itsSequence = NONE + itsSequence;          // 7
               else                                          // 8
                  itsSequence = LETTERS.substring (k, k + 1) // 9
                              + itsSequence;                // 10
            }                                               // 11
         }                                                  // 12
         itsSequence = " " + itsSequence;                   // 13
      }                                                     // 14
      theNumVics++;                                         // 15
      itsID = theNumVics;                                   // 16
      trace ("constructed ");                               // 17
   }  //======================
```

First we need to make `itsSequence` equal to the corresponding array element (line 3). That is done with the following statement (fully explained in Chapter Seven; we do not say anything more about arrays here):

```
   itsSequence = theTableau[theNumVics];
```

If this is not the empty String, it must be the one that came from `reset`, so we are done with the construction process (except putting a blank on the front and doing the three actions mentioned earlier). But if it is the empty String (line 4), we have a loop that executes from 3 to 8 times (randomly chosen in line 5), each time adding a 1-character string to the front of `itsSequence`. The string it adds is NONE half the time (again chosen at random; lines 6-7) and is otherwise the corresponding letter from the LETTERS value (lines 8-10). For instance, an object with six slots and only the second and fourth slots having no CD has `"a0c0ef"` for `itsSequence`. Then a blank is put on front (to make it one-based; line 13) and the three actions mentioned earlier are done.

Caution Any time you work out a complex logic such as this, you need to go over it very carefully to make sure that there are no errors. This Vic object class crashed on its fourth test run because the constructor does not always assign a value to `itsSequence`. After this bug was found, the declaration of `itsSequence` was changed to initialize it to the empty String. The bug was caused by a failure to obey a simple principle: Initialize every instance variable in its declaration unless you are absolutely sure it is initialized in every constructor.

Exercise 5.16 If the user calls the `reset` method twice before any Vic is constructed, what happens the second time?
Exercise 5.17 Modify the simulation by having each tracing statement begin with a list of the elements in `theStack`.
Exercise 5.18 Describe the consequence of forgetting the phrase `! seesCD() &&` in the coding of `putCD`.
Exercise 5.19* Describe the consequence of reversing the order of indexing in the constructor to have `for (int k = 1; k <= 3 + theRandy.nextInt(6); k++)`.
Exercise 5.20* Describe the consequence of forgetting the phrase `&& stackHasCD()` in the coding of `putCD`.

5.6 Case Study: Introduction To Networks

This Network material presents a completely different software situation from the Vic software, so you can see more examples of how to use the language elements you have learned. No new language features are introduced in these three sections.

Some situations where networks arise

One relevant situation is the network of nodes on the World Wide Web. Each node can send messages directly to several other nodes. A message can be routed from one node to any other node by having it pass through several direct node-to-node connections along the way.

Another relevant situation is a group of students who are registered for a group of courses. Each student has registered for several courses and each course has several students registered for it.

A third relevant situation is the set of airports served by a particular airline. Each airport provides direct flights to only a few other airports. But one can get to any other airport (hopefully) by taking a series of direct flights.

Networks and nodes

These situations have common elements you can model in software. A Network object corresponds to the WWW or the college or the airline. Each Network object has a (usually long) list of all the Node objects in the network. Nodes correspond to internet nodes or students or courses or airports. Each Node object has a (relatively short) list of other Node objects that it connects to. Figure 5.2 shows an example of a network and gives the node-list for the entire network and for each node in it.

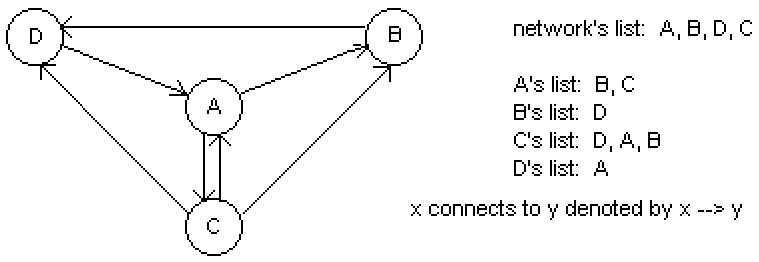

D
B
A
C
network's list: A, B, D, C
A's list: B, C
B's list: D
C's list: D, A, B
D's list: A
x connects to y denoted by x --> y

Figure 5.2 Example of a network with four nodes

"Node x connects to Node y" models any of these relations:

- Internet Node x can send a message directly to Node y.
- College Student x is registered for Course y.
- College Course x has registered in it Student y.
- The Airline has a direct flight from Airport x to Airport y.

The Network and Position classes

The only two methods for the Network class are as follows:

- `net = new Network()` constructs a Network object that represents an actual network. In other words, it creates a virtual network. Each use of this constructor normally gives a different network.
- `net.nodes()` produces the list of Nodes in the Network object `net`, starting with the first node.

The `nodes` method is the key method. It produces a new Position object. That Position object iterates through the list of all the Nodes in the Network one at a time. If you get a Position object from a Network `net` using e.g. `Position pos = net.nodes()`, you can use the following three instance methods:

- `pos.moveOn()` changes the position of `pos` to the next Node on the list of nodes (like Vic's `moveOn`).
- `pos.hasNext()` tells whether there is a Node at the current position of `pos` in its list (like Vic's `seesSlot`).
- `pos.getNext()` returns the Node object at the current position of `pos` in its list (<u>not</u> like Vic's `takeCD`, since `getNext` does not remove or change the Node at the position, it only lets you look at it).

The application program in Listing 5.6 illustrates all of these commands, as well as one for Node objects: `current.getName()` returns the name of the node referred to by `current`. This coding also illustrates the basic counting logic: If you initialize a variable to zero and increment it once each time through a loop, then when you exit the loop that variable will contain the number of iterations of the loop.

Listing 5.6 A program using a Network object

```
import javax.swing.JOptionPane;

class NetApp
{
   /** List all nodes and tell how many there are. */

   public static void main (String[ ] args)
   {  Network school = new Network();
      int count = 0;
      for (Position pos = school.nodes();  pos.hasNext();
                    pos.moveOn())
      {  Node current = pos.getNext();
         JOptionPane.showMessageDialog (null,
                   current.getName() + " is one of the nodes.");
         count++;
      }
      JOptionPane.showMessageDialog (null,
                   "The total number of nodes is " + count);
      System.exit (0);
   }  //======================
}
```

When a program executes, it almost always processes input and produces output. Typically, the output is the answer to a problem the program solves. For a Vic object, the input is the initial state of the mechanical components and the output is the final state of the mechanical components. For a Network object, the input is the initial state of the list of Nodes and their connections; the output is whatever you display for the user to see.

The Node class

The `pos.getNext()` method call produces a Node object, one of the Nodes in the Network. You can do several things with Nodes. For instance, some network situations involve two distinct groups of Nodes, as with colleges and students. Or they might represent people, some of whom are male and some female. To have a general model for such cases, we say some nodes are blue and some not.

For the college situation, blue nodes might represent students and non-blue nodes courses. Or for people, blue nodes might represent males and non-blue nodes females. For airports, we indicate that color does not matter by having all nodes blue. The method call `sam.isBlue()` tests the Node object that `sam` refers to to see if it is blue. The logic in Listing 5.6 could be modified to count the blue nodes and print the name of each by replacing the body of the for-statement by the following:

```
   Node current = pos.getNext();
   if (current.isBlue())
   {  JOptionPane.showMessageDialog (null,
                current.getName() + " is a blue node.");
      count++;
   }
```

You cannot change the state of a Network. But you can go down the list of nodes a given node connects to to find out things about those connections, as you will see next. The methods for Networks, Nodes, and Positions provide some very useful services for answering questions about a network.

The four Node methods

You have already seen two instance methods for Nodes. We introduce here two new ones, so you now have four altogether:

- `aNode.getName()` returns a String representation of the Node.
- `aNode.isBlue()` tells whether `aNode` belongs to one of two categories of Nodes.
- `aNode.equals(anotherNode)` tells whether two (possibly different) Node objects represent the same Node in the network, analogous to String `equals`.
- `aNode.nodes()` returns a Position object that iterates through the list of all Nodes that `aNode` connects to in the network.

A software suite to track water flow through the pipes of a municipal system would involve network operations. Figure 5.3 shows a network of water pipe connections. The arrows indicate the direction of flow through the pipes. If `pipes` refers to the Network here, the list that `pipes.nodes()` produces has all eight nodes of the network on it. However, the list that `x.nodes()` produces for various node values `x` has at most two nodes on it. For instance, as the figure implies, `E.nodes()` produces a Position object `pos` for which `pos.getNext()` is H, and `pos.moveOn(); pos.getNext()` is F.

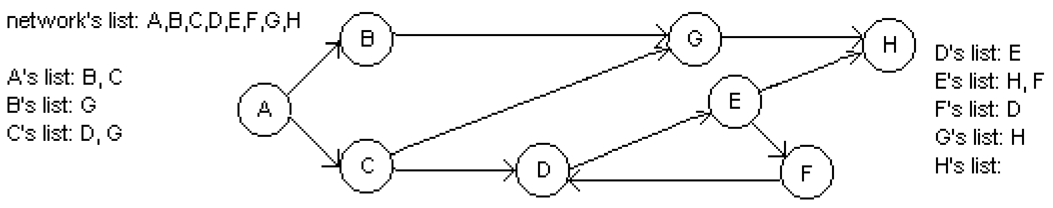


Figure 5.3 A network with 8 nodes

You might get two different descriptions of the same node when you use the `getNext` method. For instance, if `bee` and `cee` are Node variables that refer to nodes B and C in Figure 5.3, then `Node x = bee.nodes().getNext()` and `Node y = cee.nodes().getNext()` may both give you an object that refers to node G, but they may be different objects (i.e., stored in different places in RAM, so `x == y` is `false`). However, `x.equals(y)` will be true since they both represent node G.

A utilities class for Nodes

No constructor has been specified for Positions or Nodes, so you cannot make a subclass that gives additional abilities to Positions or Nodes. But you can create a NodeOp class to hold various class methods that deal with Nodes. These methods are declared using the word `static`, meaning you call them with the class name in place of the executor. Listing 5.7 (see next page) is a start on this utilities class.

The seesOnlyBlue method

An example of the use of a NodeOp method is the following statement:

```
if (NodeOp.seesOnlyBlue (sam))
   JOptionPane.showMessageDialog (null, "only blue nodes");
```

Listing 5.7 The NodeOp class

```
/** Answer queries about one or two given nodes.
 *  Precondition for all methods:  no parameter is null. */

public class NodeOp
{
   /** Tell whether par connects only to blue nodes. */

   public static boolean seesOnlyBlue (Node par)
   {  for (Position p = par.nodes();  p.hasNext();  p.moveOn())
      {  if ( ! p.getNext().isBlue())
            return false;
      }
      return true;
   }  //=======================

   /** Tell whether from connects to target.  */

   public static boolean connected (Node from, Node target)
   {  for (Position p = from.nodes();  p.hasNext();  p.moveOn())
      {  if (p.getNext().equals (target))
            return true;
      }
      return false;
   }  //=======================

   /** Tell whether par connects to any node of the same color.*/

   public static boolean seesSameColor (Node par)
   {  return (par.isBlue() && ! seesOnlyNonBlue (par))
            || (! par.isBlue() && ! seesOnlyBlue (par));
   }  //=======================

   /** Tell whether par connects to no blue node. */

   public static boolean seesOnlyNonBlue (Node par)
   {} // left as exercise
}
```

The purpose of the `seesOnlyBlue(Node)` class method in Listing 5.7 is to tell whether the parameter `par` connects to nothing but blue nodes. A good design is: You run down the list of nodes `par` connects to. If you see a node that is not blue, the answer to the question, "Does `par` connect only to blue nodes?" is `false`. If you get to the end of the list without seeing a non-blue node, the answer to the question is `true`.

The method call `p.getNext()` returns a Node value, and the executor of the `isBlue` method must be a Node value. Therefore, `p.getNext().isBlue()` is a legal chain of method calls that asks whether the Node returned by `p.getNext()` is blue.

The connected method

The purpose of the `connected(Node, Node)` class method is to tell whether the first parameter `from` connects to the second parameter `target`. A good design is: You run down the list of nodes `from` connects to. If you see the `target` node, the answer to the question "Does `from` connect to `target`?" is `true`. If you get to the end of the list without seeing the `target` node, the answer to the question is `false`.

Note that these two methods have almost exactly the same structure, except the `true` and `false` values are switched. You will see these two looping patterns very frequently. The first is typical of All-A-are-B conditions (specifically, "All nodes connected to `par` are blue"), so we call it the **All-A-are-B looping action**. The second is typical of Some-A-are-B conditions (specifically, "Some node connected to `from` equals `target`"), so we call it the **Some-A-are-B looping action**.

The seesSameColor method

The purpose of the `seesSameColor(Node)` method is to tell whether some node the parameter connects to is the same color as the parameter. It is logical that a node is connected to another node of the same color if and only if it is blue but not connected only to non-blues, or it is non-blue but not connected only to blues.

All three of the methods in Listing 5.7 are query methods because, after the executor returns `true` or `false` from the method call, every object is in the same state it had when you made the call. You may protest that the object obtained by the statement `p = par.nodes()` has changed, and you would be right. But that object is irrelevant, since:

1. That object did not exist when you called the method,
2. That object cannot be used after you return from the method, and
3. The fact that it was created and modified has no effect on anything after you return from the method. This is because each future call of `nodes()` get a totally new Position object.

Exercise 5.21 Write the method `seesOnlyNonBlue` described in Listing 5.7.
Exercise 5.22 Write a method `public static int getNumNodes (Node par)` for NodeOp: It tells how many nodes its Node parameter connects to.
Exercise 5.23 (harder) Write a main method that only prints out the name of the last blue node, but prints "no blues" if there are none.
Exercise 5.24 (harder) Write a method `public static int bidirectional (Node par)` for NodeOp: It tells whether every Node that `par` connects to, connects to `par`.
Exercise 5.25* Revise Listing 5.6 to print the percentage of nodes that are blue.
Exercise 5.26* Draw the UML class diagram for Listing 5.6.
Exercise 5.27* Write a method `public static int hasA (Node par)` for NodeOp: It tells whether the name of any Node that `par` connects to begins with "A".

5.7 Extending The Network Class

The SmartNet class in Listing 5.8 (see next page) augments the Network class by adding three useful methods. To use this class, you need only start your program with a command such as `SmartNet net = new SmartNet()`. Figure 5.4 shows the UML class diagram for the SmartNet class.

The purpose of the `connectsToAll(Node)` method is to tell whether the Node parameter connects to every other node. The executor looks through the list of all nodes in the network until it sees one the given node does not connect to, then returns `false`. It returns `true` only when the given node connects to every other node. So this is another All-A-are-B looping action.

The purpose of the `getNumNodes()` method is to tell how many nodes are in the whole network. The executor initializes a counter to zero. Then it goes through the list of all nodes in the network and adds 1 to the counter each time it sees a node. So the final answer must be the number of nodes in the whole network.

Listing 5.8 The SmartNet class

```
import javax.swing.JOptionPane;

public class SmartNet extends Network
{
   /** Tell whether par connects to all other nodes in this
    *  network. */

   public boolean connectsToAll (Node par)
   {  for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
      {  Node current = pos.getNext();
         if ( ! (current.equals (par)
                          || NodeOp.connected (current, par)))
            return false;
      }
      return true;
   }  //======================


   /** Return the total number of nodes in this network. */

   public int getNumNodes()
   {  int count = 0;
      for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
         count++;
      return count;
   }  //======================


   /** List all nodes in this network that connect to
    *  some node of the same color. */

   public void printSameColorConnections()
   {  for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
      {  Node current = pos.getNext();
         if (NodeOp.seesSameColor (current))
            JOptionPane.showMessageDialog (null,
                                           current.getName());
      }
   }  //======================
}
```

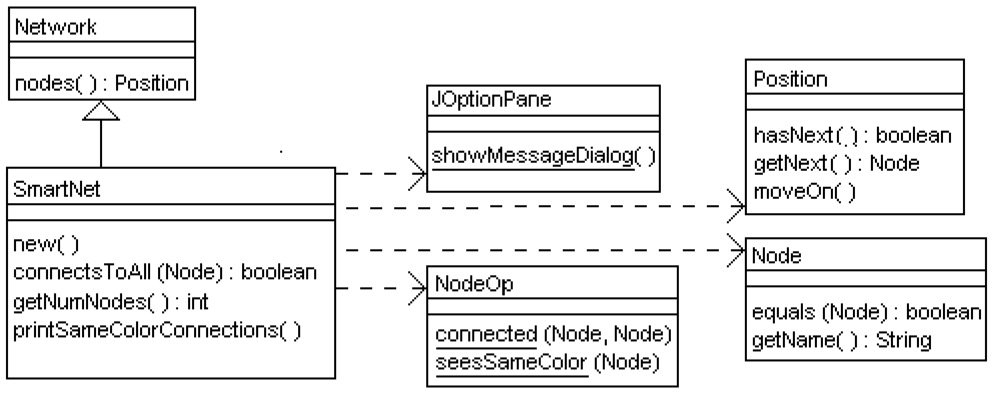


Figure 5.4 UML class diagram for the SmartNet class

The `printSameColorConnections()` method is used in a situation where you expect every node to connect to a node of a different color, such as students registered for courses. But you want to make sure this is true. The executor looks through the list of all nodes in the network to find those that are connected to a node of the same color (i.e., a blue node connected to a blue node or a non-blue node connected to a non-blue node). It prints all such nodes it sees.

You may wonder why this subclass of Network is called SmartNet. If you have a friend to whom you teach Spanish, is your friend still the same person? Answer: Yes, but a smarter person, since now your friend can speak Spanish. Similarly, making `net = new SmartNet()` instead of `net = new Network()` would produce the same network of nodes, but that network would be able to answer the question, "Does this particular Node connect to all other Nodes?", which a plain Network cannot answer. That is, SmartNet objects are smarter than plain Network objects.

Exercise 5.28 Rewrite the condition in SmartNet's `connectsToAll` method to use `&&` rather than `||`.
Exercise 5.29 Write a SmartNet method `public boolean noLoners()`: The executor tells whether each of its nodes is connected to at least one node.
Exercise 5.30 Write a SmartNet method `public boolean atLeastOneNonBlue()`: The executor tells whether at least one of its nodes is not blue.
Exercise 5.31* Write a SmartNet method `public boolean isBipartite()`: The executor tells whether every node connects only to nodes of the opposite color. Call on an existing NodeOp method to do most of the work.
Exercise 5.32* Write a SmartNet method `public boolean numBlues()`: The executor tells how many blue nodes it has.
Exercise 5.33* Write a SmartNet method `public boolean hasUniversalNode()`: Tell whether any node is connected to every node except possibly itself.
Exercise 5.34* Rewrite the SmartNet class so that `itsNumNodes` is an instance variable and `getNumNodes` returns its value rather than re-calculating it each time it is called.
Exercise 5.35* Draw the UML diagram for the SmartNet class.
Exercise 5.36** Write a SmartNet method `public boolean tellFirst (Node one, Node two)`: Tell which of the two given nodes comes first on the list of all network nodes. Return null if neither is on the list. Precondition: Neither is null.

5.8 Analysis And Design Example: The Reachability Problem

Marking nodes with numbers

Each Node has a color (blue or not) and a name; you cannot change them. But each Node also has an integer value you <u>can</u> change. This value is used to mark Nodes you have processed during execution of an algorithm, so you do not process them again.

- `x.setMark(5)` sets Node `x`'s marker number to 5 (you can use any int value here). The marker number is initially zero for all Nodes when the Network is created.
- `x.getMark()` returns the current value of `x`'s marker number.

The Reachability Problem

An important problem in the study of networks is to find out whether it is possible to send a message from a given starting point to every other node. The message can be routed through as many other nodes as is necessary, as long as it gets through eventually. For the Airline situation, the problem amounts to finding out whether the airline can get you to any airport from a given airport starting point, clearly a desirable quality in an airline.

This is the **Reachability Problem**. The `setMark` and `getMark` methods are intended to help solve this problem and many others. They are used to solve the Traveling Salesman Problem in the optional Section 5.9 on recursion.

Solution to the Reachability Problem

Solving the problem of whether all nodes are reachable from a given node is rather complex, so you need a plan. A first approach is as follows: Mark 1 on every node you can reach from the starting point. Then mark 1 on every node you can reach from one of those you marked. Then mark 1 on every node you can reach from one of those, etc. When you cannot mark any more, see if every node in the network has been marked.

How do you keep track of each node you have checked out (that is, you have marked the nodes you can reach from it)? You can use a different mark, say 2 instead of 1. So 1 means it is reachable but you have not yet checked out which nodes you can reach from it, and 2 means it is both reachable and checked out. This design can be refined as shown in the accompanying design block. Figure 5.5 traces the first few steps of this algorithm.

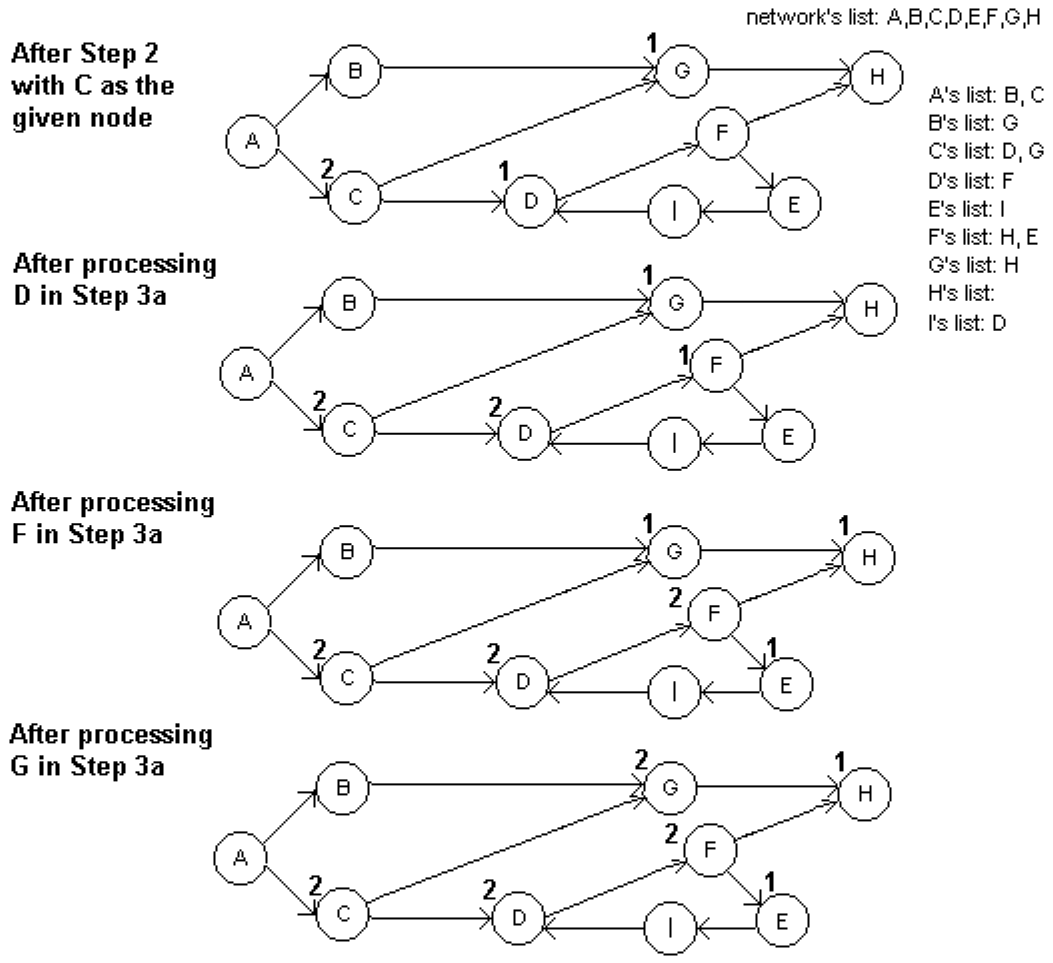

network's list: A,B,C,D,E,F,G,H
After Step 2 with C as the given node
A's list: B, C
B's list: G
C's list: D, G
D's list: F
E's list: I
F's list: H, E
G's list: H
H's list:
I's list: D
After processing D in Step 3a
After processing F in Step 3a
After processing G in Step 3a

Figure 5.5 Steps in the Reachability algorithm

How do you know when to repeat Step 3? Since this is a yes-no question, you could use a boolean variable. Set it `false` at the beginning of Step 3, then set it `true` if you find a node marked 1 at Step 3a. Repeat Step 3 if you find the boolean variable has turned `true` after going through the list of nodes. Listing 5.9 (see next page) contains the complete logic.

STRUCTURED NATURAL LANGUAGE DESIGN for the searching method

```
1. Mark 2 on the given node (since you are about to check it out).
2. Mark 1 on every node you can reach from the given node.
3. For each node current in the list of nodes of the network, do...
      3a. If current is marked 1 (reachable but not yet checked out), then...
            3aa. Mark 2 on current (since you are about to check current out).
            3ab. Mark 1 on every node you can reach from current unless it
                 already has a mark of 1 or 2.
4. Repeat Step 3 until you find no more nodes marked 1.
5. Return true if no nodes are marked 0; return false otherwise
   (since a 0 means it cannot be reached from the original node).
```

Listing 5.9 A SmartNet method for the Reachability Problem

```
/** Tell whether every node is reachable from the given node.
 *  Precondition:  source is not null.  */

public boolean allReachableFrom (Node source)
{  checkOut (source);  // marks 2 on source, 1 on some others
   boolean foundNodeToCheckOut;
   do
   {  foundNodeToCheckOut = false;
      for (Position pos= nodes(); pos.hasNext(); pos.moveOn())
      {  Node current = pos.getNext();
         if (current.getMark() == 1)
         {  foundNodeToCheckOut = true;
            checkOut (current);
         }
      }
   }while (foundNodeToCheckOut);
   return allNodesAreMarked();
}  //======================


/** Mark 2 on par; mark 1 on all nodes reachable from par
 *  except for those already marked 1 or 2.
 *  Precondition:  par is not null. */

private static void checkOut (Node par)
{  par.setMark (2);
   for (Position p = par.nodes();  p.hasNext();  p.moveOn())
   {  Node current = p.getNext();
      if (current.getMark() == 0)
         current.setMark (1);
   }
}  //======================

private boolean allNodesAreMarked()
{  boolean valueToReturn = true;
   for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
   {  if (pos.getNext().getMark() == 0)
         valueToReturn = false;
      else
         pos.getNext().setMark (0);
   }
   return valueToReturn;
}  //======================
```

Step 2 requires more than one or two statements to implement, so it is done as a call to a separate private helper method named `checkOut` which includes Step 1. Step 3ab uses the same method. Step 5 also requires more than one or two statements, so it has a separate method too, named `allNodesAreMarked`. Note: Listing 5.9 is rewritten in a much more efficient way in Section 5.9.

Programming Style You may well ask why the `checkOut` method is a class method instead of an instance method. The reason is, the Network object itself is not used at all (no explicit or implicit `this`). It would be deceptive to make it an instance method, and deception is not good style.

Implementing the Network class with a prototype

You are probably wondering why you do not get any Network software with which to test your programs, analogous to the Vic software. The reason is simple: This is a major programming project at the end of Chapter Seven (when you have learned about arrays).

For now, you can use the three classes in this section. They have overly simple logic, they are not adequate for realistic use of networks, and parts of the Node class are left as exercises. However, they are sufficient to allow you to test the methods you write. Study them carefully to reinforce your understanding of instance variables and integers.

A standard technique in software development is to develop a **prototype** of a system that sort of fakes the functionality of the real thing, for purposes of seeing how it looks and feels. Then you toss it when you write the real thing. These three classes are examples of such prototypes.

The Position methods

In this prototype implementation, a Position object keeps track of the id number of the Node at its current position in its list. When you call `getNext`, it returns a Node object with that id. The Node returned will be equal to any other Node object with the same id, because the `equals` method returns `true` if and only if the executor Node has the same id number as the parameter Node. Nodes are number from 0 to 99 inclusive, and `itsCurrent` may be greater than 99, so calculating the current node's number may require subtracting 100.

A Position object also keeps track of the id number of the last node in its list of nodes, so it knows when it has gone to far. So `getNext` returns null when its current node id is beyond its last node id, and `hasNext` simply verifies that its current node is not beyond its last node. The `moveOn` method adds 1 to the id number for its current node. These methods are coded in Listing 5.10 (see next page). Listing 5.11 provides the corresponding definition of the other two classes.

Exercise 5.37 Rewrite the `allReachableFrom` method to have just one statement subordinate to the for-loop, namely, an if-statement.
Exercise 5.38 Continue the trace of the algorithm in Figure 5.5 for two more executions of Step 3a.
Exercise 5.39 Write the `isBlue` Node method (an odd `itsID` means it is blue) and the `getName` Node method (every Node is named "Darryl") for Listing 5.11.
Exercise 5.40 Write the `setMark` and `getMark` Node methods for Listing 5.11. Add an extra instance variable named `itsMark` to do this.
Exercise 5.41 In the Node class of Listing 5.11, which Nodes does Node #6 connect to? Node #42? Node #97?
Exercise 5.42* Revise Listings 5.10 and 5.11 so each Node connects to five other Nodes of the opposite color (an odd `itsID` means it is blue). Hint: Have #18 connect to #19, #21, #23, #25, #27.

Listing 5.10 Prototype Position class of objects

```
public class Position extends Object
{
   private int itsCurrent; // Node at current position on list
   private int itsLast;    // Node at last position on list

   public Position (int first, int last)
   {  super();
      itsCurrent = first;
      itsLast = last;
   }  //======================

   public Node getNext()
   {  if (itsCurrent > itsLast)
         return null;
      else
         return new Node (itsCurrent % Network.NUM_NODES);
   }  //======================

   public boolean hasNext()
   {  return itsCurrent <= itsLast;
   }  //======================

   public void moveOn()
   {  itsCurrent++;
   }  //======================
}
```

Listing 5.11 Prototype Network and Node classes of objects

```
public class Network extends Object
{
   public static final int NUM_NODES = 100;
   ///////////////////////////////////////

   public Position nodes()
   {  return new Position (0, NUM_NODES - 1);
   }  //======================
}
//################################################################

public class Node extends Object
{
   private int itsID;     // ranges from 0 to NUM_NODES - 1

   public Node (int index)
   {  super();
      itsID = index;
   }  //======================

   public Position nodes()
   {  return new Position (itsID + 1, itsID + 4);
   }  //======================

   public boolean equals (Node par)
   {  return par != null && this.itsID == par.itsID;
   }  //======================
}
```

*5.9 Recursion (*Enrichment)*

The following method definition is at the beginning of Chapter Three. It uses a while-statement to put a CD in each slot of the executor's sequence of slots, by moving one slot forward each time until there are no more slots to fill:

```
public void fillSlots()
{  while (seesSlot())
   {  putCD();
      moveOn();
   }
}  //======================
```

When you think about what a while-statement means, you can see this is the same logic as the following. Of course, that last line is a comment instead of a Java statement, so the effect is not the same. But it describes what the while-statement does.

```
public void fillSlots()
{  if (seesSlot())
   {  putCD();
      moveOn();
      // repeat this if-statement
   }
}  //======================
```

This logic can be expressed a third way. The comment has been replaced by a call of the method that contains the if-statement. This is called **recursion**.

```
public void fillSlots()
{  if (seesSlot())
   {  putCD();
      moveOn();
      fillSlots();  // i.e., repeat this if-statement
   }
}  //======================
```

Execution does not go on forever for any of these logics. For each of them, the executor moves one slot forward each time through the loop. Eventually it comes to the end of the sequence. Then the `seesSlot()` condition is `false` and execution stops.

Recursive version of fillSlots(int)

Consider this task: We want to fill in the first four slots in a sequence, or the first six slots or whatever is specified in a variable `numToFill`. Afterward we want to back up to the starting position. But if there are less than `numToFill` in the sequence, we just fill in all there are and then back up to the original position.

That logic can be written fairly clearly using recursion: When `numToFill` is positive and you see a slot, then you can fill `numToFill` slots if you (a) put a CD in the first slot, then (b) move on, then (c) fill `numToFill-1` slots, then (d) back up by one slot. The following is a line-for-line translation of this logic:

```
public void fillSlots (int numToFill)
{  if (seesSlot() && numToFill > 0)
   {  putCD();                           // (a)
      moveOn();                          // (b)
      fillSlots (numToFill - 1);         // (c)
      backUp();                          // (d)
   }
}  //======================
```

The recursive logic makes it unnecessary to keep track of the starting position in the sequence. The next-to-last statement in the method just means: Repeat this if-statement but with `numToFill` having a value 1 less than the previous time.

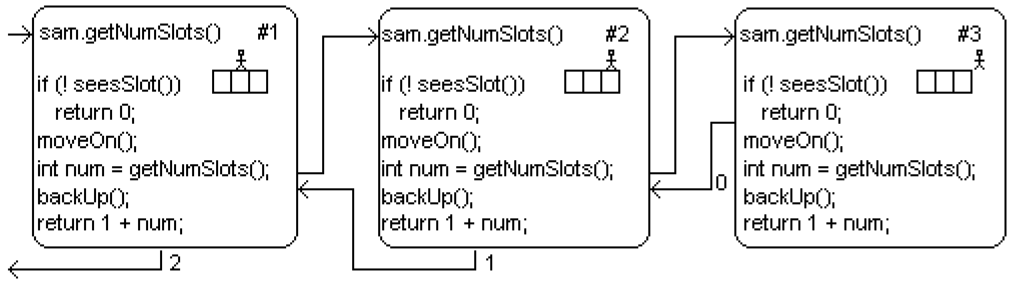
Recursive version of getNumSlots()

The `getNumSlots` method in the Interlude (before Chapter Four) had the executor move down its sequence, adding 1 to a counter for each slot it saw. When it came to the end, it backed up to the starting point (because it is a query method) and then returned the final value of the counter. A recursive solution to the request to count your slots and report how many you have is to (a) report zero if you have no slots, otherwise (b) move on to the next slot and count how many there are from that point on, then (c) back up one slot and report 1 more than you found in step (b). The coding is as follows:

```
public int getNumSlots()
{  if ( ! seesSlot())                    // (a)
      return 0;                          // (a)
   moveOn();                             // (b)
   int num = getNumSlots();              // (b)
   backUp();                             // (c)
   return 1 + num;                       // (c)
}  //======================
```

Figure 5.6 should give you some idea of how recursion works for the method call `sam.getNumSlots()` with two slots left in `sam's` sequence. The key is, each method call has the runtime system create a new Method object to carry out the process given by the method definition. <u>Useful metaphor</u>: The method definition for `getNumSlots` is a college course that Method objects can take to learn how to do something. Calling the method when `sam` has two slots left has the runtime system create a graduate of that course, labeled #1 in the figure, who is to carry out the process studied in the course.

Part of the process Method object #1 carries out is a method call that creates a totally different graduate of the course, labeled #2 in the figure, to carry out that same process when the sequence has one slot. Part of the process Method object #2 carries out is a method call that creates a third graduate of the course, labeled #3 in the figure, to carry out the same process when the sequence has no slots. So #3 returns 0 to #2, who stores 0 in his own variable `num`, thus returns 1 to #1, who stores the 1 in his own totally separate variable `num`, and thus returns 2 to the point where it was originally called.



```
sam.getNumSlots()
if (! seesSlot())
  return 0;
moveOn();
int num = getNumSlots();
backUp();
return 1 + num;
```

Figure 5.6 Call of getNumSlots with two additional recursive calls

People new to recursion sometimes ask, "How can a method call itself?" The answer is, "That doesn't happen; what happens is, one Method object calls a totally different Method object that graduated from the same course."

Turing Machine programs, as described at the end of Chapter Three, conventionally use recursion to the exclusion of while-statements. With recursion, methods in a subclass of the Tum class can always be coded as a single multi-selection statement.

Recursion applied to Networks

The `allReachableFrom` method in the earlier Listing 5.9 for Networks is much easier and much more efficient when you use recursion. You can eliminate all but the first and last statements of the `allReachableFrom` method if you just replace one statement in the `checkOut` method, as shown in Listing 5.12. Specifically, checking out a node `par` consists in marking it 2, then checking out every node marked 0 you can reach directly from `par`. In effect, to mark all the nodes you can reach from `par`, you first mark `par` and then you simply mark all the nodes you can reach from a node you can reach from `par`.

Listing 5.12 Recursive allReachableFrom method in SmartNet

```
   /** Rewrite of SmartNet's allReachableFrom for efficiency. */

   public boolean allReachableFrom (Node source)
   {  checkOut (source);
      // all of these lines have been deleted
      return allNodesAreMarked();
   }  //======================

   private static void checkOut (Node par)
   {  par.setMark (2);
      for (Position p = par.nodes();  p.hasNext();  p.moveOn())
      {  Node current = p.getNext();
         if (current.getMark() == 0)
            checkOut (current);  // this is the only line changed
      }
   }  //======================
```

The **Traveling Salesman Problem** for a network of airports is to answer the question of whether you can visit every airport without going through the same airport twice, starting from a given airport. A general solution can be obtained by calling `canTravelFrom (givenAirport, getNumNodes())` for the `canTravelFrom` method in Listing 5.13 (see next page). This method is based on the logic in the accompanying design block.

```
Question:  Can one travel through  n  different nodes marked zero starting from base?
Answer:    If  n  is 1, then...
               Of course it is possible, since base itself is the one.
           Otherwise...
               Mark 1 on base.
               If there is any node such that
               (a) you can reach it from base in one direct step, and
               (b) it is marked zero, and
               (c) you can travel through  n-1  different nodes marked zero
                   starting with it, then...
                       It is obviously possible.
               Otherwise...
                       It is not possible.
```

Listing 5.13 Recursive NodeOp method for the Traveling Salesman

```
/** Tell whether it is possible to travel through n nodes,
 *  all different and all marked 0, starting from Node base.
 *  Precondition:  base is marked 0.  */

public static boolean canTravelFrom (Node base, int n)
{  if (n <= 1)
      return true;
   base.setMark (1);
   for (Position p = base.nodes();  p.hasNext();  p.moveOn())
   {  Node current = p.getNext();
      if (current.getMark() == 0
                   && canTravelFrom (current, n - 1))
      {  base.setMark (0);   // restore original state
         return true;
      }
   }
   base.setMark (0);         // restore original state
   return false;
}  //======================
```

Language elements
A method can call any method in its class, even itself.

Exercise 5.43 (harder) Rewrite the `fillSlots` method in Listing 3.4 recursively, where the executor returns to its original position.
Exercise 5.44 (harder) Rewrite the `seesAllFilled` method in Listing 3.5 recursively.
Exercise 5.45* Rewrite the `clearSlotsToStack` method in Listing 3.4 recursively.
Exercise 5.46* Rewrite the `hasAsManySlotsAs` method in Listing 3.6 recursively.
Exercise 5.47** Rewrite the `lastEmptySlot` method in Listing 3.8 recursively.
Exercise 5.48** Write a recursive method `public boolean canFillAllSlots()` for a subclass of Vic: The executor tells whether there are enough CDs in the stack to fill all of its empty slots. When the method terminates, the executor must be in the same state it was in when the method began. Note that a non-recursive solution is too hard.

*5.10 More On JOptionPane (*Sun Library)*

This section describes methods from the `javax.swing.JOptionPane` class that can be quite useful for major projects or in later courses, though they are not mentioned elsewhere in this book. Look at your documentation to see additional possibilities, something like `jdk1.3\docs\api\javax\swing\JOptionPane.html` on your hard disk. Or on the web, go to the following Internet address, click on the package you want, then click on the class you want:

```
   http://java.sun.com/j2se/1.3/docs/api
```

The `showMessageDialog` method call has the following more general form:

```
   showMessageDialog (null, someMessage, "title at the top",
                      JOptionPane.someIntName)
```

The first parameter is a Component value; if it is not null, the dialog is displayed in the frame for that Component object and usually positioned directly below the Component.

The second parameter `someMessage` is typically a String to be displayed, on several lines if it includes the newline character `'\n'`. However, the parameter type is specified as Object, so you may make it any of several kinds of displayable objects. If it is null, no Exception is thrown, but nothing is displayed.

The third parameter is a String value to replace the default title "Confirm". For the fourth parameter, replace `someIntName` by one of the following to specify the icon:

- PLAIN_MESSAGE (no icon at all)
- ERROR_MESSAGE (a horizontal bar inside an octagon)
- WARNING_MESSAGE (an "!" inside a triangle)
- QUESTION_MESSAGE (a "?" inside a rectangle)
- INFORMATION_MESSAGE (an "i" inside a circle, which is the default icon)

The `showInputDialog` method call has the default title "Input", the "?" icon, and two buttons saying "OK" and "Cancel". It is also overloaded with a four-parameter version having the same four parameters with the same meaning as for `showMessageDialog`.

The `showConfirmDialog` can return any of several int values to tell what button the user clicked: YES_OPTION, NO_OPTION, OK_OPTION, CANCEL_OPTION, and CLOSED_OPTION (meaning that the user clicked the X-shaped closer icon in the top right of the window). The name `showConfirmDialog` is also overloaded; the more general form of it has two variants:

```
   showConfirmDialog (null, messageString, titleString,
                      JOptionPane.YES_NO_OPTION)
   showConfirmDialog (null, messageString, titleString,
                      JOptionPane.YES_NO_CANCEL_OPTION)
```

The first three parameters are as for `showMessageDialog` (the default title here is "Select an option"). The fourth parameter names an int value that specifies what buttons (either two or three of them) are displayed for the user to click. The default option (when only the first two parameters are used) is the YES_NO_CANCEL_OPTION.

5.11 Review Of Chapter Five

Listing 5.2 and Listing 5.3 illustrate almost all Java language features introduced in this chapter other than recursion.

About the Java language:

- You may call a **class method** (declared using `static`) with the name of its class in place of the executor. By contrast, an instance method requires a reference to an object of the class as the executor.
- You can use `this` inside an instance method as a reference to the executor of the method call. By contrast, a class method does not have an executor, so you cannot use `this` inside a class method.
- If you call a method without an executor and without a class name in place of the executor, the compiler uses the **default**. For calling a class method, the default is the class containing the method call. For calling an instance method, the default is `this` of the method containing the method call (i.e., it is `this` instance of the class).
- You may declare a variable in a class outside of any method, which makes it a **field variable**. Then each object of the class has its own value for the variable if it is an instance variable. But there is only one value for the whole class if it is a **class variable** (i.e., declared with the `static` modifier).
- If the declaration of a field variable assigns it a value, that takes effect for an object's instance variable when the constructor is called, for a class variable when the program begins. The runtime system supplies a default value if you do not: zero or null or false, as appropriate.
- The word **final** is allowed in any kind of variable declaration. It means that the first assignment of a value to that variable is the last one.
- If a method call returns an object, it can be used as the executor of another method call. This is a cascading or **chaining** of method calls.
- `someString.length()` returns the number of characters in the `someString`.
- `someString.substring(startInt, endInt)` returns the substring of `someString` running from position `startInt` to just before position `endInt`. Numbering is **zero-based** (starts from zero). You must have the following true:
  `0 <= startInt <= endInt <= someString.length()`
- In the grammar summary of Figure 5.7, the `Type` is int, boolean, or a ClassName; `ParameterList` is one or more `Type VariableName` combinations separated by commas, with no assignments to those parameters; and the `Modifier` is either `public` or `private`. Italicized words are optional elements.

`Modifier static` *`final`*
`      Type VariableName = Expression;`
	declaration of class variable with initial value

`Modifier static Type MethodName`
`   (`*`ParameterList`*`)  { StatementGroup }`
`Modifier static void MethodName`
`   (`*`ParameterList`*`)  { StatementGroup }`
	declaration of class method that accepts input initially assigned to its formal parameters

Figure 5.7 Declarations added in Chapter Five

Other vocabulary to remember:

- If a class does not have any instance methods or instance variables or main method, we call it an **utilities class**. If it has instance methods or instance variables and no main method, we call it an **object class**. The only other kind of class we use in this book is a class that has a main method and no other method, called an **application program**. A class method is **independent** if it could be in any class.
- String values are **immutable**, i.e., the attributes of these objects cannot be changed.

Answers to Selected Exercises

5.1 Insert before the first if: if (count == 0) return 0;

5.2 Insert before the for-statement: if (expo > 30) expo = 30;

5.3
```
public static int power (int base, int expo)
{	if (base <= 0 || expo < 0)
		return 0;
	int limit = 2147483647 / base;
	int power = 1;
	for ( ; expo > 0 && power <= limit;  expo--)
		power = power * base;
	if (expo == 0)
		return power;
	else
		return -1;
}
```

5.7 Put this outside of any method: private static String theLatestName = "none so far";
Put this statement at the end of the coding for the constructor: theLatestName = itsFirstName;
Add this method to the Person class:
```
public static String getNameOfLatestCreated()
{	return theLatestName;
}
```

5.8
```
java.util.Random randy = new java.util.Random();
public static int range (int one, int two)
{	if (one <= two)
		return one + randy.nextInt (two - one + 1);
	else
		return two + randy.nextInt (one - two + 1);
}
```

5.11 Put this declaration as the first statement of toString: final int BASE = 10;
Replace the phrase "itsHour < 10" by "itsHour < BASE".

5.14
```
private static char getSub (String sequence, int position)
{	return sequence.substring (position, position + 1);
}
```
Replace the following four parts of Listings 5.3-5.5:
```
return ! getSub (itsSequence, itsPos).equals (NONE);  // last 2 statements of seesCD
theStack = theStack + getSub (itsSequence, itsPos);  // statement in takeCD
+ getSub (itsStack, atEnd)  // fourth line of the body of putCD
itsSequence = getSub (LETTERS, k) + itsSequence;  // in the constructor; note the order
```

5.16 The new set of strings replaces the old set of strings provided by the previous call of reset.
This is no actual change in values, assuming that the same args was used each time.

5.17 Replace "Vic# " in the trace method by theStack + "; Vic# ".

5.18 If seesSlot() is false, the correct result happens -- the program terminates. Otherwise, when the stack is not empty, the "contract" for putCD is violated, because nothing is supposed to happen by calling putCD when a CD is already in that slot, but instead a CD is taken from the stack and put into the current slot in place of the CD that is already there.

5.21 It is the same coding as for seesOnlyBlue except remove the ! operator in the if-condition.

5.22
```
public static int getNumNodes (Node par)
{	int count = 0;
	for (Position p = par.nodes();  p.hasNext();  p.moveOn())
		count++;
	return count;
}
```

5.23
```
public static void main (String[ ] args)
{       Network airline = new Network();
        String lastBlue = "no blues";
        for (Position pos = airline.nodes();  pos.hasNext();  pos.moveOn())
        {       if (pos.getNext().isBlue())
                        lastBlue = pos.getNext().getName();
        }
        JOptionPane.showMessageDialog (null, lastBlue);
}
```
5.24
```
public static int bidirectional (Node par)
{       for (Position p = par.nodes();  p.hasNext();  p.moveOn())
        {       if ( ! connected (p.getNext(), par))
                        return false;
        }
        return true;
}
```
5.28 Replace the line beginning with "if" by:
```
if ( ! current.equals (par) && ! NodeOp.connected (current, par))
```
5.29
```
public boolean noLoners()
{       for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
        {       if ( ! pos.getNext().nodes().hasNext())
                        return false;
        }
        return true;
}
```
5.30
```
public boolean atLeastOneNonBlue()
{       for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
        {       if ( ! pos.getNext().isBlue())
                        return true;
        }
        return false;
}
```
5.37 Replace the for-statement by the following:
```
for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
        if (pos.getNext().getMark() == 1)
        {       foundNodeToCheckOut = true;
                checkOut (pos.getNext());
        }
```
5.38 Next after G, H will be checked out, which changes its 1 to 2. Next, the boolean variable is tested and the do-while repeats. E is checked out, which changes its 1 to 2 and I's 0 to 1.

5.39
```
public boolean isBlue()
{       return itsID % 2 == 1;
}
public String getName()
{       return "Darryl";
}
```
5.40
```
private int itsMark = 0;
public void setMark (int newMark)
{       itsMark = newMark;
}
public int getMark()
{       return itsMark;
}
```
5.41 Node #6 connects to Nodes 7, 8, 9, 10.
Node #42 connects to Nodes 43, 44, 45, 46.
Node #97 connects to Nodes 98, 99, 0, 1.

5.43
```
public void fillSlots()
{       if (seesSlot())
        {       putCD();
                moveOn();
                fillSlots();
                backUp();
        }
}
```
5.44
```
public boolean seesAllFilled()
{       if ( ! seesSlot())
                return true;
        else if ( ! seesCD())
                return false;
        moveOn();
        boolean valueToReturn = seesAllFilled();
        backUp();
        return valueToReturn;
}
```

Review: Overall Java Language So Far

This summary presents a description of all of the language elements seen so far except for the `String[] args` in a main method heading. It even includes some elements from the next chapter relating to `double`. Several special notations are used to make the descriptions compact yet reasonably clear:

- A phrase in italics is optional.
- Words beginning with a small letter are reserved words (the keywords plus `true`, `false`, and `null`); they must be written exactly as is.
- Words beginning with a capital letter and ending in "Name" can be replaced by any identifier of a class, method, or variable as indicated. An identifier is a name that the writer of the definition makes up. It can be made up of letters, digits, and underscores. It cannot start with a digit and it cannot be a reserved word.
- Other words beginning with a capital letter represent phrases defined elsewhere in this section to be any of several different things.

A **CompilationFile** is a file you may compile in Java, structured as shown below. If it has the optional `extends` phrase, then every public declaration in the superclass named is indirectly in the defined class by inheritance, except for constructors and those declarations that the subclass overrides by giving a new declaration with the same signature (i.e., the same name and the same parameter structure).

```
ImportDirectives
public  class  ClassName  extends SuperclassName
{  DeclarationGroup
}
```

You may have import directives in a compilable file, as long as they come before any class definition in the file. The **ImportDirectives** consist of one or more lines structured as shown below. The first allows the use of the named class from another package. The second allows the use of any class from that other package. The **PackageName** is several identifiers separated by dots, e.g., `javax.swing` and `java.awt.event`.

```
import PackageName . ClassName ;
import PackageName . * ;
```

A **DeclarationGroup** is any number of consecutive Declarations. A **Declaration** can be one of the five forms listed below (lines beginning with a left brace are a continuation of the preceding line). The first two are field variable declarations; the last three are method declarations. If the assignment to the variable is not present, the variable is initialized to zero, null, or false (whichever is appropriate). Methods defined in the same class can have the same MethodName if they have different signatures. For the constructor declaration (listed last), the ClassName has to be that of the class the constructor is in. You may omit `super(...);` which will then default to `super();` .

```
ModifierPhrase  Type  VariableName ;
ModifierPhrase  Type  VariableName = Expression ;
ModifierPhrase  Type  MethodName ( ParameterList )
   { StatementGroup }
ModifierPhrase  void  MethodName ( ParameterList )
   { StatementGroup }
public  ClassName ( ParameterList )
   { super ( ArgumentList ) ;  StatementGroup }
```

Examples An object of the following class represents a TV show that has a ranking on a scale of 1 to 10. The top line is an import directive whose PackageName is java.util (a package containing Random). The TVShow class has two declarations of VariableNames with assignments and two without. These variable declarations have three different ModifierPhrases: "public static final", "private static final", and "private". The Type is int in two cases; it is Random and String in the other cases. The TVShow class also has a declaration of a constructor with the ParameterList "String name, int rank" and two declarations of non-constructor methods.

```
import java.util.Random;

public class TVShow
{
   public static final int MAX_RANK = 10;  // two class variables
   private static final Random randy = new Random();
   ///////////////////////////////
   private String itsName;              // two instance variables
   private int itsRank;


   /** Create an object for the given name and rank. */

   public TVShow (String name, int rank)           // constructor
   {  super();
      itsName = name;
      if (rank >= 1 && rank <= MAX_RANK)
         itsRank = rank;
      else
         itsRank = 1;
   }  //======================

   /** Return a new TVShow object with the given name but a
    *  randomly chosen value for its rank. */

   public static TVShow randomShow (String name)  // class method
   {  return new TVShow (name, 1 + randy.nextInt (MAX_RANK));
   }  //======================


   /** Return the rank for this particular TVShow object. */

   public int getRank()                          // instance method
   {  return itsRank;
   }  //======================
}
```

A **ModifierPhrase** can have one of the following four forms. The modifier `public` says any other class can access the name; `private` says only definitions within the current class can access the name; `final` says it cannot be changed later (overridden or reassigned); and `static` says access can be with the class name in place of an instance of the class (thus it is a class method or a class variable). A non-constructor declaration without the `static` modifier is an instance method or instance variable.

```
   public static final
   private static final
   public final
   private final
```

The only **Type** discussed through the first part of Chapter Six is one of the following. The boolean type is for values that are either `true` or `false`. The int type is for numbers without decimal points. The double type is for numbers that have a decimal point and digits after it. The ClassName is an object type.

```
boolean
int
double
PackageName.  ClassName
```

An **ArgumentList**, used in method calls, consists of one or more Expressions with commas between two consecutive expressions. So it has one of the following two forms.

```
Expression
Expression , ArgumentList
```

Examples The following are statements containing method calls with 2, 1, and 0 arguments, respectively.

```
TVShow t1 = new TVShow ("Law & Order", 9);
TVShow t2 = TVShow.randomShow ("NYPD Blue");
System.out.println (t2.getRank());
```

A **ParameterList**, used in method headings, describes the kinds of arguments that must be passed as input to the execution of the method. It has one of the following two forms.

```
Type VariableName
Type VariableName , ParameterList
```

Examples The following method headings in the TVShow class have 2, 1, and 0 parameters, respectively.

```
public TVShow (String name, int rank)
public static TVShow randomShow (String name)
public int getRank()
```

A **StatementGroup** is any number of consecutive Statements. A **Statement** can be any of the following nine forms. If an `else` could be matched with more than one `if`, then the `else` is to be matched with the most recent such `if`. The parentheses shown after `if`, `while`, and `for` are not optional.

```
Type VariableName ;
MethodCall ;
Initializer ;
return Expression ;
{  StatementGroup  }
if ( Condition )  Statement
if ( Condition )  Statement  else  Statement
while ( Condition )  Statement
do  Statement  while ( Condition ) ;
for ( Initializer ; Condition ; Updater )  Statement
```

An **Initializer** is something that can be used in the first part of a for-statement, to assign a value to the loop control variable at the beginning of the looping process. It can have any of the following four forms.

```
   Type VariableReference = Expression
   VariableReference = Expression
   VariableReference ++
   VariableReference --
```

An **Updater** is something that changes the value of the loop control variable, as follows.

```
   MethodCall
   VariableReference = Expression
   VariableReference ++
   VariableReference --
```

Examples The following are possible for-statement headings.

```
   for (int k = 5;  k < size;  k++)
   for (k--;  k > 0;  k--)
   for (itsMin = min;  itsMin < 0;  itsMin = itsMin + 60)
   for (p = nodes;  p.hasNext();  p.moveOn())
```

An **Expression** is a phrase for which the runtime system can compute a value. It has one of the following three forms.

```
   ObjectExpression
   NumericExpression
   Condition
```

An **ObjectExpression** has one of the following forms. The `this` reference is only allowed within an instance method or constructor. The VariableReference must be an object type and the MethodCall must return an object type.

```
   this
   null
   VariableReference
   MethodCall
   StringValue
   new ClassName ( ArgumentList )
```

A **VariableReference** has one of the following forms. Option: You may omit `this.` or `ClassName.` at the beginning of a VariableReference, to have it default to the executor of the current instance method or to the class the VariableReference is in, respectively.

```
   VariableName
   PackageName.  ClassName . VariableName
   this . super . VariableName
   ObjectExpression . VariableName
```

Examples The following statements assign various kinds of ObjectExpressions to variables.

```
   TVShow a = null;                 // a refers to no object
   Object b = System.out;           // assign a VariableReference
   String c = sam.getPosition();    // assign a MethodCall
   String u = "Friends";            // assign a StringValue
   TVShow x = new TVShow ("24", 8); // assign a new value
```

A **MethodCall** has one of the following forms. Option: You may omit `this.` or `ClassName.` at the beginning of a MethodCall, to have it default to the executor of the current instance method or to the class the MethodCall is in, respectively.

```
PackageName.  ClassName . MethodName ( ArgumentList )
this . super . MethodName ( ArgumentList )
ObjectExpression . MethodName ( ArgumentList )
```

A **NumericExpression** has one of the following forms. The MethodCall must return an int or double type and the VariableReference must be an int or double type. The last form simply means that, if you already have a NumericExpression, you can put parentheses around it and it will still be a NumericExpression.

```
IntegerLiteral
DoubleLiteral
VariableReference
MethodCall
NumericExpression * NumericExpression
NumericExpression / NumericExpression
NumericExpression % NumericExpression
NumericExpression + NumericExpression
NumericExpression - NumericExpression
( NumericExpression )
```

A **Condition** is a kind of Expression. It has one of the following forms. The MethodCall must return a boolean type and the VariableReference must be a boolean type.

```
true
false
VariableReference
MethodCall
ComparisonOfTwoValues
! Condition
Condition && Condition
Condition || Condition
( Condition )
```

<u>Examples</u> The following assign various NumericExpressions and Conditions to variables.

```
int e = 47;                 // assign an IntegerLiteral
int f = e;                  // assign a VariableReference
int k = x.getRank();        // assign a MethodCall
int m = f * (e - 5);        // assign result of operator
boolean p = true;           // assign a boolean literal
boolean q = s.equals (t);   // assign a MethodCall
boolean r = e <= f;         // assign a ComparisonOfTwoValues
boolean ok = p && ! q;      // assign result of operator
```

A **ComparisonOfTwoValues** is a special kind of Condition. It has one of the following eight forms.

```
NumericExpression <  NumericExpression
NumericExpression <= NumericExpression
NumericExpression >  NumericExpression
NumericExpression >= NumericExpression
NumericExpression == NumericExpression
NumericExpression != NumericExpression
ObjectExpression == ObjectExpression
ObjectExpression != ObjectExpression
```

A **StringValue** has one of the following foms. The only operator that can be used with object references is the plus sign, where at least one of the operands is a String value. It concatenates the two String values. If the other operand is a numeric value, the plus sign concatenates the string of characters that form the numeral with the string of characters in the String value.

```
   StringLiteral
   VariableReference
   MethodCall
   StringValue + StringValue
   StringValue + NumericExpression
   NumericExpression + StringValue
```

A **StringLiteral** is a pair of quotes containing any characters other than a backslash or a quote, except you may use one of the \n or \\ or \b or \t combinations.

An **IntegerLiteral** is a sequence of one or more digits, optionally preceded by a negative sign.

A **DoubleLiteral** is a sequence of one or more digits, then a decimal point, then a sequence of one or more digits. The whole is optionally preceded by a negative sign.

The twenty **reserved words** seen so far in this book are the following (`true`, `false`, and `null` are technically not keywords). Those in the first line can only be used outside of a method body, and those in the last line can only be used inside a method body.

```
private  public  class  extends  static  void
int  boolean  new  true  false  null
if  else  while  do  for  return  this  super
```

The thirteen **Sun standard library** methods seen so far in this book are the following.

```
Object:       equals (someObject),
              toString().
String:       equals (someString),
              length(),
              substring (start, end).
Random:       new Random(),
              nextInt (limitInt).
System:       System.exit (0),
              System.out.println (someString).
Integer:      Integer.parseInt (someString).
JOptionPane:  JOptionPane.showMessageDialog (null, someString),
              JOptionPane.showInputDialog (promptString),
              JOptionPane.showConfirmDialog (null, someString).
```

Examples The main method of the following class can be executed from the command line using `java Puzzler`. It illustrates the use of some of these library methods.

```
   public class Puzzler
   {
      public static void main (String[] args)
      {  String s = JOptionPane.showInputDialog ("How many?");
         int number = Integer.parseInt (s);
         for (int k = 0;  k < s.length();  k++)
            System.out.println (s.substring (0, k));
         JOptionPane.showMessageDialog (null, "all done");
         System.exit (0);
      }  //======================
   }
```