2 Conditionals and Boolean Methods

Overview

This chapter introduces the Vic software to control electronic components. Vics provide a moderately realistic context in which you can learn about some new Java language features. These features let you ask your objects questions and then decide, based on the answers they give, what actions they should take.

- Sections 2.1-2.2 describe basic Vic commands and review main methods and subclassing.
- Sections 2.3-2.5 present conditional statements.
- Sections 2.6-2.8 discuss boolean operators, boolean variables, and methods that return boolean values.
- Sections 2.9-2.10 introduce additional useful topics including UML notation.

Some students like a more detailed preview of the language features they are about to learn. If you are such a student, you will find it helpful to skim the review at the end of each chapter before reading the chapter itself.

2.1 Using Vic Objects To Control Appliances

Your company sells a programmable machine that stores compact discs (CDs) and moves them around when the operator pushes certain buttons. The machine uses a computer chip to do this. Your job is to write the programs for this chip, using the Java programming language.

The primary component of this machine has a mechanical arm and one sequence of three or more slots for storing CDs (the higher-priced machines have more slots). At any given time, some slots contain a CD and some do not. The mechanical arm is positioned at one point in the sequence of slots, either right at a slot or just after all the slots.

Each programmable machine has one or more of these primary components. The machine also has a place where extra CDs can be stored, called its **stack** of CDs. This stack sometimes contains many CDs and sometimes contains none at all. When you put a CD on the stack, and later put another CD on the stack, then the later one is on top. That means that, the next time you take a CD off of the stack, you will get the later one, not the one that is underneath it.

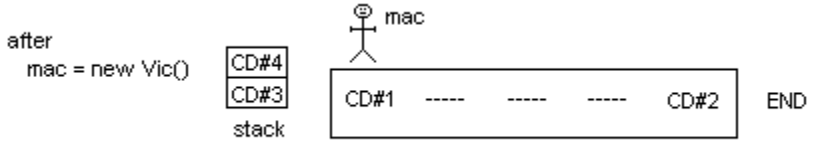
You can have one of these primary components perform one of four kinds of operations, named `takeCD`, `moveOn`, `putCD`, and `backUp`:

- The `takeCD` operation causes the mechanical arm to take a CD out of the slot at the current position and place it on top of the stack. This does not change the arm's position in the sequence. If there is no CD in the slot, the `takeCD` operation has no effect.
- The `moveOn` operation moves the mechanical arm down from its current position in the sequence of slots to the next position.
- The `putCD` operation causes the mechanical arm to remove a CD from the top of the stack and put it in the slot at the current position. This does not change the arm's position in the sequence. If a CD is already in the slot, or if no CD is in the stack, the `putCD` operation has no effect.
- The `backUp` operation moves the mechanical arm up from its current position in the sequence of slots to the position just before it.

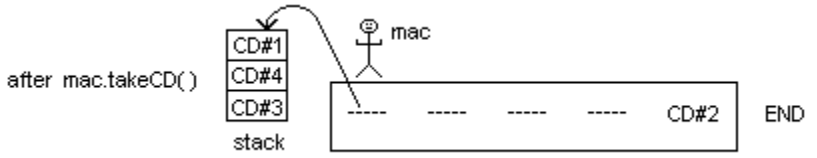
The phrase `Vic mac` in a program declares `mac` as the name for a variable, a part of the RAM's data area which can refer to a Vic object. The phrase `mac = new Vic()` in a program creates the object and puts a reference to the object in the variable named `mac`. Thereafter, a mention of `mac` in the program is an indirect mention of the object. The object specifies the component's sequence of slots and current position.

Figure 2.1 shows how the status of the Vic object changes after each operation. The stick figure indicates the position of the mechanical arm that shifts CDs. Initially, this particular Vic object has a sequence of five slots, with CDs in the first and last slots, as well as two CDs in the stack. The initial arrangement of CDs is whatever was left in the slots and stack when the machine was last turned off.

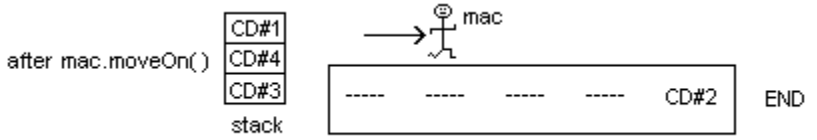
`mac = new Vic()`
creates the internal description for the Vic object, positioned at the beginning of the sequence of slots.



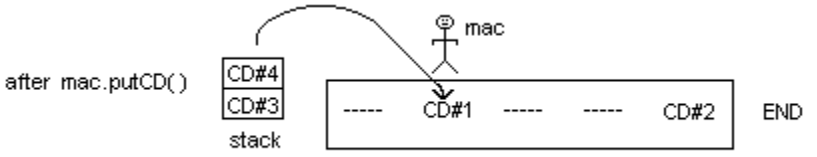
`mac.takeCD()`
asks the Vic to take a CD out of its current slot and put it on top of the stack of CDs for the machine.



`mac.moveOn()`
asks the Vic to move to the next slot after the one at its current position.



`mac.putCD()`
asks the Vic to put a CD in its current slot. This CD comes from the top of the stack of CDs.



`mac.backUp()`
asks the Vic to back up to the slot just before the one at its current position.

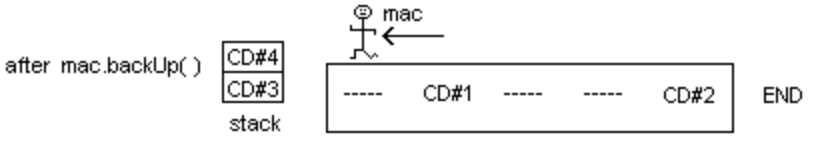


Figure 2.1 The meaning of the four basic Vic commands

Objects in Vic programs

When a program that operates the Programmable CD Organizer machine begins its work, it creates an internal description of each of these primary components and puts that data in RAM. This internal description is an object. An object is a virtual mechanism when it is a computer model of a physical mechanism. Since this particular object stores Virtual CDs, we can call it a Vic for short.

The four basic Vic operations of `putCD`, `takeCD`, `moveOn`, and `backUp` are what actually move the springs, gears, and grippers in the physical machine. You write a program to perform a complex task by performing these simple physical actions in an appropriate sequence. When `mac` refers to some Vic, `mac.takeCD()`, `mac.moveOn()`, `mac.putCD()`, and `mac.backUp()` are the ways you express these actions in a Java program.

An application program

Suppose you want a Vic object to take a CD from its third slot and put the CD in its second slot. You could have the computer chip execute the application program in Listing 2.1. The comments at the end of certain lines (signaled by the `//` symbol) explain what is happening. The class heading and the method heading (the two boldfaced lines) plus the corresponding two pairs of braces are what this book always uses for application programs, except that the name MoveOne is chosen to reflect what the specific program does.

Listing 2.1 An application program using one Vic object

```
public class MoveOne
{
   // Take a CD from the third slot; put it in the second slot.

   public static void main (String[ ] args)
   {  Vic sue;          // 1
      sue = new Vic();  // 2

      sue.moveOn();     // 3  move to the second slot
      sue.moveOn();     // 4  move to the third slot
      sue.takeCD();     // 5  take CD from slot 3, put on stack

      sue.backUp();     // 6  move back to the second slot
      sue.putCD();      // 7  put CD in slot 2, taken from stack
   }  //=====================
}
```

Figure 2.2 shows the status of the Vic object at three points during one execution of the MoveOne program. The first two statements of the main method create the Vic object, position it at the first slot in the first sequence of slots, and make `sue` a reference to that object. The top part of Figure 2.2 shows what things might look like at this point.

After the Vic object is created, the next three statements move the mechanical arm to the third slot in the sequence and then take CD#1 out of that slot. The middle part of Figure 2.2 shows the current status. Note that CD#1 is no longer in the slot; it is on the stack.

The last two statements of the main method move the mechanical arm back up to the second slot in the sequence and then have it transfer CD#1 from the stack into that second slot. The bottom part of Figure 2.2 shows the final status of the machine.

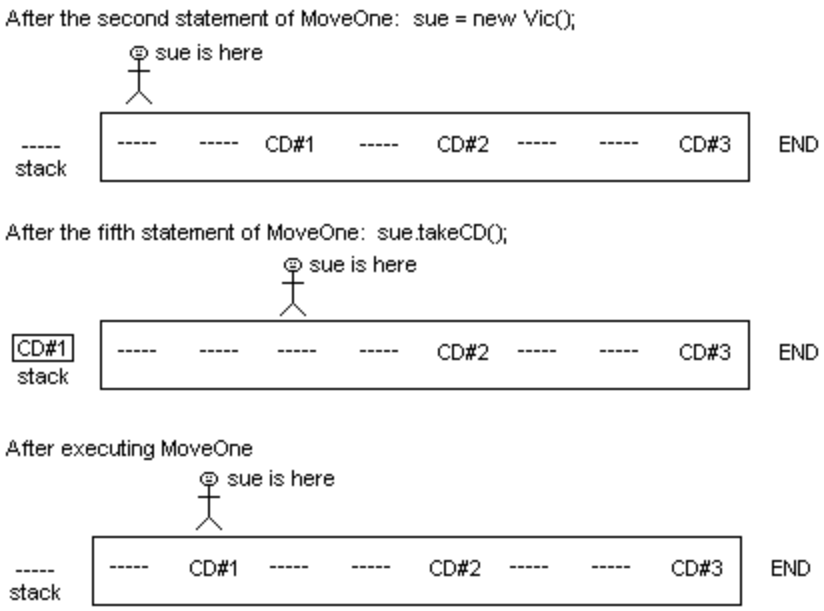


Figure 2.2 Stages of execution for MoveOne

The Vic simulator

The company engineers have not finished building the physical machine, so they have provided you with a simulation in software. It is the Vic class, available on this book's website (or you could use the simpler one in Chapter Five). It lets you test the programs you write.

When you run a program with this simulation, a graphical representation of the entire machine appears on the screen with CDs and slots. It carries out the commands you gave it, slowly enough for you to follow.

To run an application program such as MoveOne in Listing 2.1, you must already have the Vic class compiled. You type the lines of the MoveOne program into a plain-text file named `MoveOne.java`, then compile it to produce the executable file named `MoveOne.class`. You enter `java MoveOne` at the prompt in the terminal window to run the program. The runtime system then executes the commands in the executable file.

For this particular Vic simulator, the graphics display shows one to four sequences of slots, representing actual components of the physical machine. Each sequence has at most eight slots. Figure 2.3 shows roughly what the graphics display looks like. The Vic software provides faked names for CDs consisting of a letter and a digit, so you can tell which CDs were originally in which slots. The CD names along the left side (b1 and a2 in the figure) are the CDs that are on the stack (b1 is on top of the stack).

Figure 2.3 shows three sequences with four slots in the first two and three slots in the third. The stick figure indicates that only one Vic object has been created (i.e., `new Vic()` has been executed one time), and it is currently positioned at the second slot.

Vic methods that perform an action print a record of the action in the terminal window. If the graphics window covers the terminal window, move the terminal window around to make at least its lower part visible so you can see this record.

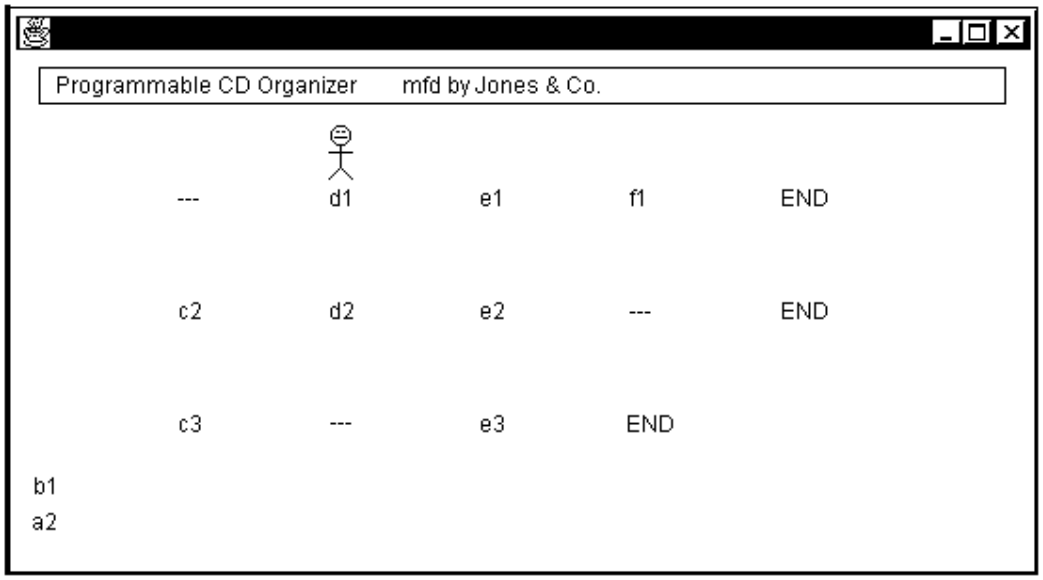


Figure 2.3 The graphics display for the Vic software

The reset command

You can test your programs by setting up the CDs in a particular order before a test run. The Vic simulator provides a `reset` command that puts CDs in whatever slots you choose at runtime; simply have `Vic.reset(args);` as the first statement of the main logic. For instance, if you want to have the arrangement of CDs shown in Figure 2.3 when you run the program in Listing 2.1, type in `Vic.reset(args);` as the first statement of the program and execute the program by giving the command

```
java MoveOne 0111 1110 101
```

in the terminal window. The simulator then creates three sequences (because you gave three "words" after the basic command) with no CD wherever you have a 0 in a string and a CD wherever you have something other than a 0.

If you execute the same program using the command `java MoveOne 010 01100`, the simulator creates a machine with two sequences: The first sequence has three slots with a CD only in the second slot, and the second sequence has five slots with CDs only in the second and third slots. [If you use BlueJ instead of the terminal window, put the following in the "main" text box: `{"010", "01100"}`].

Those extra words in the command line are called **command-line arguments**. The runtime system puts those words in `args`, and the `reset` command uses the information it receives in `args` to initialize the simulation. If no words are entered, or if the `reset` command is not used before any Vic object is created, the simulation creates an initial arrangement of CDs, slots, and sequences for you at random.

Language elements

A CompilableUnit can be: public class ClassName { Declaration }
A Declaration can be: public static void main (String [ ] args) { StatementGroup }
A StatementGroup is any number of Statements.
A Statement can be: ClassName VariableName ; e.g., Vic sam;
or: VariableName = new ClassName ( ) ; e.g., sam = new Vic();
or: VariableName . MethodName ( ) ; e.g., sam.backUp();
or: ClassName . MethodName (Expression) ; e.g., Vic.reset(args);
Anything after // on a line in a program has no effect on the program.

Exercise 2.1 Write an application program that creates a Vic object and then moves a CD out of its third slot into its first slot. Assume the first slot is empty and the third is not.
Exercise 2.2 Write an application program that creates a Vic object and then puts a CD in each of its first three slots. Assume the stack has enough CDs.
Exercise 2.3 Write an application program that creates a Vic object and then takes a CD out of its second slot and its fourth slot. Use `reset` to be sure you have enough slots.
Exercise 2.4* Write an application program that creates a Vic object and then takes a CD from each of its second and fourth slots and puts one of them in its fifth slot. Use `reset` to be sure you have enough slots. Which one ends up in the fifth slot?
Exercise 2.5* Write an application program that creates a Vic object and then swaps the CD in its third slot with the CD in its second slot. Assume it has CDs in both slots.

2.2 Defining A Subclass Containing Only Instance Methods

When a command in a program refers to a Vic object before the dot, it is a message sent to the Vic object. For instance, `sue.putCD()` sends a message to `sue` requesting that `sue` put a CD into the current slot from the stack of CDs. So the main method in MoveOne (Listing 2.1) sends a message to a Vic requesting it to move forward to the third slot and take the CD from that slot. It then sends messages to the Vic requesting it to move back to the second slot and put the CD from the stack into that second slot.

Reminder: The stack and the sequences of slots exist independently of the program that controls them. The phrase `new Vic()` does not create a stack and a sequence; it only creates a description of them in RAM. You need the internal description (object) so you can send messages that cause changes in a physical stack and sequence.

Choosing statements to make into a method

You can expect to use the sequence of two messages

```
moveOn();
takeCD();
```

many times in many programs, with various Vic objects receiving that pair of messages. You saw `sue` doing these actions in Listing 2.1. Java allows you to invent new messages for Vics that are combinations of existing messages. This simplifies your programs. For instance, you can define a new message named `moveTake`: `sam.moveTake()` tells `sam` to execute those two commands in that order, and `sue.moveTake()` tells `sue` to execute those two commands in that order.

The sequence of two messages

```
backUp();
putCD();
```

will also be quite common, sent to various Vic objects (you saw it in Listing 2.1 with `sue` receiving the messages). You can define a new message named `backPut`: `sam.backPut()` tells `sam` to carry out those two messages in that order, and `sue.backPut()` tells `sue` to carry out those two messages in that order.

A simple Vic object does not know what the two words `moveTake` and `backPut` mean. They are not part of its vocabulary. You need a new class of objects that can understand these two messages plus all the messages a Vic understands. Let us call this new kind of Vic object a SmartVic. Then you could rewrite the main method in Listing 2.1 to do exactly the same thing but with a simpler list of statements, as follows:

```
   public static void main (String[ ] args)
   {  SmartVic sam;          // create variable named sam
      sam = new SmartVic();  // create object that sam refers to
      sam.moveOn();
      sam.moveTake();        // take CD in slot 3, put on stack
      sam.backPut();         // put that CD in slot 2
   }  //======================
```

How to define a class of objects

The class definition in Listing 2.2 says that, if you create an object using the phrase `new SmartVic()` instead of `new Vic()`, you can send the `moveTake` and `backPut` messages to that object, as well as all of the usual Vic messages. In a sense, a SmartVic object is better educated than a basic Vic object, because it inherits all of the capabilities of a Vic object and adds two more.

Listing 2.2 The SmartVic class of objects

```
public class SmartVic extends Vic
{
   public void moveTake()
   {  moveOn();
      takeCD();
   }  //======================


   public void backPut()
   {  backUp();
      putCD();
   }  //======================
}
```

A class definition that extends the capabilities of a Vic object must have the heading

```
   public class WhateverNameYouChoose extends Vic
```

followed by a matched pair of braces that contain some definitions. This particular class definition contains two method definitions beginning `public void`. Within such a method definition, you do not name the variable that refers to the object that receives the message. This lets you use any variable name you like (such as `sue`, `sam`, or whomever) when you give the command outside the method. So the `backPut` definition says that for any `x`, if you have previously defined `x = new SmartVic()`, then `x.backPut()` has the same meaning as the following:

```
   x.backUp();
   x.putCD();
```

Listing 2.3 (see next page) illustrates the use of these new commands in a program that moves three CDs backward one slot. The program creates a new SmartVic object and sends it messages to take the CD out of slot 2 and put that CD into slot 1. Then the object moves forward to slot 2 so it can repeat the actions, this time taking the CD out of slot 3 and putting it in slot 2. Finally, the object moves forward to slot 3 and repeats the actions, this time taking the CD out of slot 4 and putting it in slot 3. Figure 2.4 shows a sample run of this program.

Listing 2.3 An application program using one SmartVic

```
public class BringThreeBack
{
   // Move the CDs in slots 2, 3, and 4 back to slots 1, 2, 3,
   // respectively.  Presumes a reset with at least 4 slots.

   public static void main (String[ ] args)
   {  Vic.reset (args);     // 1
      SmartVic sue;         // 2
      sue = new SmartVic(); // 3
      sue.moveTake();       // 4  move to slot 2 and take CD
      sue.backPut();        // 5  back to slot 1 and put CD there

      sue.moveOn();         // 6
      sue.moveTake();       // 7  move to slot 3 and take CD
      sue.backPut();        // 8  back to slot 2 and put CD there

      sue.moveOn();         // 9
      sue.moveTake();       // 10 move to slot 4 and take CD
      sue.backPut();        // 11 back to slot 3 and put CD there
   }  //====================
}
```

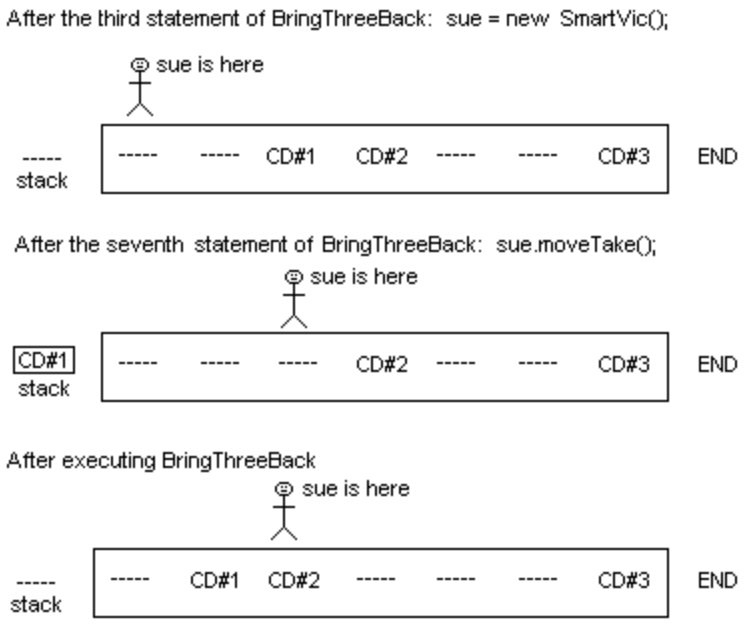


Figure 2.4 Stages of execution for BringThreeBack

Caution You will save yourself a lot of grief if you check the following three points before you compile a class definition: (1) No class heading or method heading has a semicolon at the end. (2) Each class heading and method heading has a left brace immediately below it. (3) Every left brace has a corresponding right brace several lines lower and aligned with it.

Language elements

A CompilableUnit can be: public class ClassName extends ClassName { DeclarationGroup }
A DeclarationGroup is any number of Declarations.
A Declaration can be: public void MethodName ( ) { StatementGroup }
A Statement can be: MethodName ( ) ; e.g., moveOn();

Terminology The phrase `sue.moveTake()` in the BringThreeBack class is a **method call**. The phrase `public void moveTake()` in the SmartVic class is the **method heading**. But the **method** itself is the process of moving the mechanical arm forward and taking a CD. If you misspell or miscapitalize the method call, the compiler will not be able to connect it to the method heading so that the runtime system can carry out the method.

Exercise 2.6 Rewrite the main method in the answer to Exercise 2.1 to use SmartVics instead of Vics, thereby shortening the logic.
Exercise 2.7 Rewrite the main method in the answer to Exercise 2.3 to use SmartVics instead of Vics, thereby shortening the logic.
Exercise 2.8 Write a SmartVic method `public void movePut()`: The executor moves forward one slot and puts a CD there from the stack (unless the stack is empty).
Exercise 2.9 Write an application program that uses a SmartVic object, augmented by the preceding exercise, to move a CD from the third slot to the fifth slot.
Exercise 2.10* Write a SmartVic method `public void backTake()`: The executor moves backward one slot and takes a CD from there to go on top of the stack.
Exercise 2.11* Write an application program that uses a SmartVic object, augmented by `movePut` and `backTake` as just described, to swap the CD in the second slot with the CD in the first slot using only four message commands.
Exercise 2.12* Write an application program that uses a SmartVic object, augmented by `movePut` and `backTake` as just described, to move the CD in the first slot into the second, the CD in the second slot into the third, and the CD in the third slot into the first.

2.3 The If Statement

A program that is given bad input can produce bad results. For instance, if some program is set up to accept only a numeric input at a certain point, and the input has letters in it, the program may fail. Or if a Vic program is set up to take a certain action at the fourth or fifth slot of a sequence and the sequence only has three slots, the program may fail. In both of these cases the program is not **robust**: It does not handle unexpected input well. For instance, Listing 2.3 is not robust because it fails if the sequence has only three slots.

You can make your Vic programs robust if you learn to expect the unexpected and adjust for it. A program fails if there is no slot at a point where a Vic tries to `putCD`, `takeCD`, or `moveOn`. To prevent failure, you need to be able to test whether a certain condition is true. Fortunately, you can ask a Vic object either of the two kinds of questions shown in Figure 2.5; the Vic object (referred to here as `aVic`) gives the answers in the form of a **condition** (an expression that is either true or false).

`aVic.seesSlot()`
is true if aVic is not past its last slot and is false otherwise. So it means aVic actually has a current slot.

`aVic.seesCD()`
is true if aVic's current slot has a CD in it and is false otherwise. Evaluation of this condition causes the program to fail if aVic.seesSlot() is false.

Figure 2.5 Two Vic methods that answer questions

Examples of if statements

In order to use these conditions, you need to have a kind of statement that will take an action if a certain condition is true but will skip the action if the condition is false. The if-statement is that kind of statement. It comes in two forms, illustrated in Listing 2.4 (see next page).

Listing 2.4 An application program using one Vic object, with if-statements

```
public class TwoToFour
{
   // If a CD is in the second slot, take it out and then
   // put it in the fourth slot if possible (robustly).

   public static void main (String[ ] args)
   {  Vic.reset (args);       // 1
      Vic sue;                // 2
      sue = new Vic();        // 3
      sue.moveOn();           // 4

      if (sue.seesCD())       // 5  if sue sees a CD in slot 2,
      {  sue.takeCD();        // 6     then sue takes that CD
         sue.moveOn();        // 7
         sue.moveOn();        // 8
         if (sue.seesSlot())  // 9  if sue has a slot number 4,
            sue.putCD();      // 10    then sue puts the CD there
      }                       // 11
   }  //======================
}
```

Listing 2.4 creates a Vic object and has it move to the second slot in its sequence (lines 1-4). If it does not see a CD in that slot, nothing else happens in the program. This is because the if-statement (line 5) tests the condition `sue.seesCD()` and, if the condition is false, skips execution of all the statements between the matched braces that follow the condition (lines 6-11).

Those statements to be executed if the second slot has a CD are to take the CD from the slot, move on to the fourth slot, and put the CD there. However, a sequence might have only three slots. If so, trying to put a CD in the fourth slot would make the program fail (specifically, the Vic software prints a warning message and stops the program). So this program tests the condition `sue.seesSlot()` at the fourth slot (line 9) and, only if that condition is true, executes the one statement that follows the condition (line 10).

A basic **if-statement** has two parts for you to fill in: the condition it tests and the subordinate statement it executes only if the condition is true. This book boldfaces the word `if` in listings to signal it needs a subordinate statement. The two general forms of the basic if-statement are as follows:

```
if (Condition)        if (Condition)
   Statement          {  Statement
                        Statement...
                      }
```

If you want a group of two or more statements to be executed only when the given condition is true, you must put the matched pair of braces around the group. If you want only one statement conditionally executed, you do not need the braces around it. However, you may use them if you wish. This book uses braces when the subordinate part is two or more lines. As Listing 2.4 illustrates, you are allowed to have any statements inside the braces of an if-statement, even another if-statement.

Continuing the email message metaphor of Section 1.6, when you write `sam.seesCD()` in a program, it sends a message to the person whose email address is stored in `sam`. The message says, "Is it true or false that you see a CD in your current slot?" The person sends back to you a response of `true` or `false`. You then look at the response to decide what to ask the person to do next.

Using several Vic objects in a program

Suppose you want to move the CD from slot 3 into slot 5 for each of the first three sequences of slots. This is a very straightforward thing to do except for two possible problems: One is that you might not have as many as three sequences, and the other is that one or more of those sequences might not have as many as five slots.

The `seesSlot()` method call can be used to solve both of those problems. The first time your program creates a new Vic object, you get the first sequence of slots. The next time your program creates a new Vic object, you get the second sequence of slots, if there is one. If you then immediately test `seesSlot()`, it will be false if there was no second sequence. Otherwise, it will have at least three slots, since all sequences do. This all leads to the structured plan shown in the accompanying design block.

DESIGN for moving the CD from slot 3 into slot 5 for each of three sequences

1. Create the first Vic object.
2. If its third slot contains a CD, then...
   Move it into the fifth slot, checking first that the Vic has a fourth and fifth slot.
3. Create a second Vic object.
4. If it has at least one slot, it has at least three (by definition of Vics), so...
   - 4a. Do what you did for the first sequence, as described in step 2.
   - 4b. Create a third Vic object.
   - 4c. If it has at least one slot, it has at least three (by definition of Vics), so...
     Do what you did for the first sequence, as described in step 2.

Listing 2.5 (see next page) implements this plan as a Java program. Since Step 2 requires a method for accomplishing a subtask that is used in two more places, it is best to make a Java method out of it, the `shiftThreeToFive` method in the Shifter class (Note: You do not need to memorize any of these Vic subclasses for later in this book).

Java requires you to put each class with the heading `public class Whatever` in a separate file to be compiled, with the name `Whatever.java`. Listing 2.5 is the contents of two separate compilable files, `ThreeSequences.java` and `Shifter.java`.

Note particularly the advantage of defining a method without mentioning which object is to carry out the task. That allows you to have three different Vic objects execute the sequence of statements in the `shiftThreeToFive` method.

The ThreeSequences program creates three different sequences, so it uses the three different descriptive variable names `one`, `two`, and `three`. However, since `one` is never mentioned after `two` is created, nor `two` after `three` is created, you could use just one Vic variable throughout. For instance, you could write `sam` in place of each of the three variable names in that class. But then you would have to omit the second and third declarations of Vic variables -- you can only declare a variable one time in a method.

For all exercises from now on, unless otherwise stated, write your answer so that the application programs cannot fail. You <u>are</u> doing all the unstarred exercises, are you not? You cannot learn this material well if you do not do them. You may compile a method you write as an exercise by enclosing it in a matched pair of braces with an appropriate class heading.

Language elements

A Statement can be:	if (Condition) Statement	
or:	if (Condition) { StatementGroup }	
A Condition can be:	MethodName ( )	e.g., seesCD()
or:	VariableName . MethodName ( )	e.g., sue.seesCD()

Listing 2.5 An application program using three objects of the Shifter class

```
public class ThreeSequences
{
   // For each of the first three sequences, move a CD from the
   // third slot to the fifth slot, insofar as possible.

   public static void main (String[ ] args)
   {  Vic.reset (args);
      Shifter one;
      one = new Shifter();                  // design step 1
      one.shiftThreeToFive();               // design step 2

      Shifter two;
      two = new Shifter();                  // design step 3
      if (two.seesSlot())                   // design step 4
      {  two.shiftThreeToFive();            // design step 4a
         Shifter three;
         three = new Shifter();             // design step 4b
         if (three.seesSlot())              // design step 4c
            three.shiftThreeToFive();
      }
   }  //====================
}
//#################################################################

public class Shifter extends Vic
{
   // take the CD from slot 3 (if any) and put it in slot 5.

   public void shiftThreeToFive()
   {  moveOn();
      moveOn();                             // now in slot 3
      if (seesCD())
      {  takeCD();
         moveOn();                          // now in slot 4
         if (seesSlot())
         {  moveOn();                       // now in slot 5
            if (seesSlot())
               putCD();
         }
      }
   }  //====================
}
```

Exercise 2.13 Revise the program in the earlier Listing 2.3 so it cannot fail even with bad input (presumably from the `reset` method).
Exercise 2.14 (harder) Write an application program that creates a Vic and then has it take a CD from each of its fourth and fifth slots. Be sure the program cannot fail.
Exercise 2.15 (harder) Write a method `public void swapTwo()` for a subclass of Vic: The executor swaps the CD in the current slot with the CD in the following slot, but it moves no CD at all if either CD is missing. Leave the executor at its original position.
Exercise 2.16* Write an application program that creates a Vic and then has it take a CD out of its fifth slot and put it into its seventh slot. Be sure the program cannot fail.
Exercise 2.17* Write an application program that moves a CD from slot 3 to slot 2 of the first sequence and also from slot 2 to slot 3 of the second sequence. Use SmartVic objects (as defined in the earlier Listing 2.2). Be sure the program cannot fail.
Exercise 2.18* Write an application program that takes the CD out of the second slot of each of the first four sequences. Use SmartVic objects. Be sure the program cannot fail.

2.4 Using Class Methods And Javadoc Comments In A Program

Some methods can be called with the class name in place of an executor. An example is `Vic.reset(args)`. Such a method is called a **class method.** You are sending a message to the class as a whole, not to an individual instance of the class. By contrast, a method that requires an instance (object) of the class for its executor is an **instance method**.

Two new Vic class methods

The Vic class provides a class method for you to print messages on the display. If you put `Vic.say("Hello world")` in your program, `Hello world` will appear on an LCD screen on the front of the machine. `Vic.say("No CDs")` will cause `No CDs` to appear on the front of the machine. You can put whatever you want inside the quotes except a backslash `\` or quotes themselves (this caveat is explained in Chapter Six).

The Vic class also provides a class method for you to find out whether the stack has any CDs on it (as opposed to being empty). If you execute `sam.putCD()` and the stack is empty, nothing happens. Often you need to know whether something happens. So you test the condition `Vic.stackHasCD()`. Figure 2.6 describes these two new methods more precisely.

`Vic.say("whatever")`
displays the given message on an LCD screen the machine has, so the operator of the Programmable CD Organizer can read it. The command first erases whatever might have been on the LCD screen before.

`Vic.stackHasCD()`
is true if the stack contains at least one CD and is false otherwise.

Figure 2.6 Two new Vic class methods

A **parameter** or **argument** is a value you put in the parentheses after a method name to make a method call. It gives information to the method that it needs so it can do its job. The parameter of `Vic.say` is always a string of characters. The phrase `Vic.say("Hello")` is a method call with a parameter, namely `"Hello"`. `Vic.reset(args)` is another method call with a parameter `args`, which contains whatever strings of characters might have been typed on the command line.

Programming Style Java allows you to call a class method by using an instance of the class in place of the class name. For instance, if you declare `Vic sam`, then `sam.say("Hi")` is a legal method call and does the same thing as `Vic.say("Hi")`. However, this can be deceptive, so it should be avoided.

The `switchTo` method in the Turtle class (Section 1.4) is actually a class method, since the color applies to the drawing surface rather than to the individual Turtle. So it is better to use `Turtle.switchTo(aColor)` than `sam.switchTo(aColor)`. This was not mentioned in Chapter One only because it contained enough material as it was. Note: You may use `say` or `stackHasCD` within a method in a subclass of Vic without having "Vic." before it -- the compiler will know what you mean.

Example using the three class methods

Suppose you would like to have a program to fill in the first three slots of the first sequence. The program begins with an unknown number of CDs in the stack. It makes sense to stop trying to fill slots as soon as the stack runs out of CDs. So begin by seeing if the stack has any CDs. If so, you create a new Vic object and fill the first slot, then check to see if the stack has more CDs. If so, you move on to the second slot and put a CD there. You check one more time to see if the stack still has at least one CD. If so, you move on to the third slot and put a CD there.

This logic is implemented in Listing 2.6. The condition in line 3 causes the executor to skip lines 4-15 if the stack is empty. The condition in line 7 causes the executor to skip lines 8-14 if the stack is empty at that point. Of course, if a slot already has a CD in it, the `sue.putCD()` message has no effect, so the CD on the stack will be available for the next slot.

Listing 2.6 An application program using the three Vic class methods

```
public class FillThreeApp
{
   /** Put a CD in each of the first three slots.
    *  Stop as soon as you run out of CDs on the stack. */

   public static void main (String[ ] args)
   {  Vic.say ("This program fills the first three slots.");
      Vic.reset (args);                  // 2
      if (Vic.stackHasCD())              // 3
      {  Vic sue;                        // 4
         sue = new Vic();                // 5
         sue.putCD();                    // 6    in slot 1
         if (Vic.stackHasCD())           // 7
         {  sue.moveOn();                // 8    to slot 2
            sue.putCD();                 // 9
            if (Vic.stackHasCD())        // 10
            {  sue.moveOn();             // 11   to slot 3
               sue.putCD();              // 12
            }                            // 13
         }                               // 14
      }                                  // 15
      Vic.say ("All done!");             // 16
   }  //======================
}
```

Elementary javadoc comments

Listing 2.6 illustrates a second kind of Java comment: `/*` causes everything to be ignored until `*/` is seen, even if several lines later. This book henceforth uses the **javadoc standard for commenting methods**, which is the following:

- Put the description of the method before the method heading, beginning with `/**`.
- Put an asterisk at the beginning of each additional line (this asterisk is optional).
- Put `*/` at the end of the comment.

The javadoc formatting tool uses these special comments. You execute it by issuing in the terminal window the command `javadoc SomeClass.java` for a class declared in the `SomeClass.java` file. Then the javadoc formatting tool produces a webpage in a file named `SomeClass.html` which displays documentation for the class very nicely. Each comment that comes right before a public class, method, or variable declaration and begins with `/**` appears in this documentation (you will learn about public variables in Chapter Five).

Language elements

A Condition can be:	ClassName . MethodName ( )	e.g., Vic.stackHasCD( )
A Statement can be:	ClassName . MethodName (someString)	e.g., Vic.say ("Hi");

Anything between /* and */, even over several lines, has no effect on the program.

Exercise 2.19* Write an application program that fills the third and fourth slots in the second sequence. Stop as soon as you run out of CDs in the stack. Print an appropriate message at the beginning of the program. Be sure the program cannot fail.
Exercise 2.20* Write a method `public void fillThree()` for a subclass of Vic: The executor puts a CD in each of its first three slots, but stopping when the stack is empty (hint: similar to part of Listing 2.6). Then revise Listing 2.6 to use that method to have both the first and the second sequence fill their first three slots.

2.5 The If-Else Statement And The Block Statement

Listing 2.7 is a Java program for a machine with at least two sequences. It creates one Vic for each sequence. Then it sends messages to each Vic asking it to take a CD if one is in its first slot, but to put a CD there if no CD is in its first slot. The program uses the **if-else statement** to decide which of two messages to send, `takeCD` or `putCD`. The word `else` in this context essentially means "otherwise" or "if that condition wasn't true". You should compile this program and run it several times to watch it work.

Listing 2.7 An application program using two Vics

```
public class TwoVics
{
   /** Take the CD in slot 1 if there is one, otherwise put a
    *  CD there; repeat for the second sequence, if any. */

   public static void main (String[ ] args)
   {  Vic first;                        // 1
      first = new Vic();                // 2
      if (first.seesCD())               // 3
         first.takeCD();                // 4
      else                              // 5
         first.putCD();                 // 6

      Vic second;                       // 7
      second = new Vic();               // 8
      if (second.seesSlot())            // 9
      {  if (second.seesCD())           // 10
            second.takeCD();            // 11
         else                           // 12
            second.putCD();             // 13
      }                                 // 14
   }  //=====================
}
```

The general form of an if-else statement is as follows. If you want to replace either of these Statements by a group of two or more statements, you put a matched pair of braces around that group. The compiler will then treat the braces plus the group of statements within it as one statement, called a **block statement**:

```
if (Condition)
   Statement
else
   Statement
```

When an if-else statement is executed, the runtime system begins by evaluating the if-condition. If the condition is true, the statement after it is executed and the statement after the `else` is ignored. If the condition is false, the statement after it is ignored and the statement after the `else` is executed. That is, exactly one of them is executed. Figure 2.7 illustrates the meaning of the two varieties of conditional statements you have.

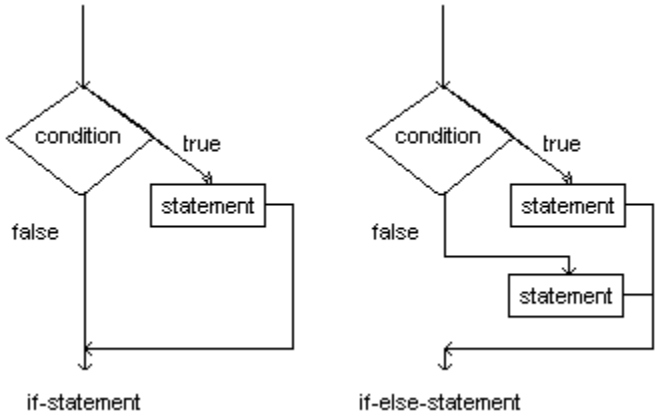


Figure 2.7 Flow-of-control for if and if-else statements

Examples of block statements

The following are some examples that use block statements in an if-else statement. The general principle is that, if the action to be taken when the if-condition is false consists of two or more statements in sequence, you need to put a matched pair of braces around those statements:

```
if (Vic.stackHasCD())
{  putCD();
   moveOn();
}
else
{  moveOn();
   takeCD();
}
```

```
if (seesSlot())
   takeCD();
else
{   backUp();
    takeCD();
    moveOn();
}
```

```
if (seesCD())
{  takeCD();
   moveOn();
   putCD();
}
else
   moveOn();
```

Caution If you forget a pair of braces around a group of statements after `else` or after the if-condition, the program usually produces the wrong result. Even if you indent properly, the program still produces the wrong result. It may not even compile without error. Indentation makes the logic easier for humans to understand; the compiler ignores it.

Technically, the if-statement in lines 10-13 of Listing 2.7 is only one statement, not two separate statements. It is clearer to use braces around it, though braces are not required.

The physical mechanism

You are perhaps puzzled about how the physical mechanism can do a `moveOn` operation to a non-existent position where `seesSlot()` is false, e.g., when it is already at the seventh slot in its sequence of seven. The physical mechanism has a gripper part that moves in a groove beside the sequence, stopping next to the appropriate slot. Slots are 3/8" apart, so the gripper moves 3/8" each time. The groove extends 3/8" beyond the last slot. So it can move in the groove though it is not to a slot.

In the virtual mechanism, there is no gripper and no groove. The `moveOn` operation adds 1 to the numeric value tracked by its position. So even if there are only seven slots, its position can still be the number 8.

Lab Practice You are probably writing most of the unstarred exercises on a sheet of paper and checking them against the answers. But you should from time to time type up at least a few of the exercises and compile them without first looking at the answers (the answer to an exercise saying "Write a method...for a subclass of Vic" should be put inside your own class that extends Vic and compiled). This tests to make sure you know where the parentheses, braces, and semicolons go, and that you are using correct capitalization, too. These things are not hard, but they do take some practice: **You can't learn stuff if you don't do stuff.**

Language elements

A Statement can be:	if (Condition) Statement else Statement
or:	if (Condition) Statement else { StatementGroup }
or:	if (Condition) { StatementGroup } else Statement
or:	if (Condition) { StatementGroup } else { StatementGroup }

Exercise 2.21 (harder) Write a statement that puts a CD in `sam`'s current spot if there is no CD there and the stack has a CD, but otherwise moves on to the next slot.
Exercise 2.22 (harder) Write a method `public void shiftForward()` for a subclass of Vic: The executor moves the CD from the next slot into the current slot, but only if the current slot is empty and the next slot exists and is not empty. In either case, the executor is at the next slot after the method finishes executing.
Exercise 2.23** Write an application program that only does things with the second sequence: First take a CD from its first slot and one from its fourth slot; then if at least one of those two slots was not empty, put a CD in its second slot.

2.6 Boolean Methods And The Not-Operator

You have seen two kinds of Vic instance method calls: four actions (`takeCD`, `putCD`, `moveOn`, and `backUp`) and two conditions (`seesSlot` and `seesCD`). And you have seen how to define new actions in an extension of the Vic class. In this section you will see how to define new conditions in an extension of the Vic class.

Preconditions

Programming Style It is good style to have a comment at the beginning of each method definition to describe what happens when the commands in the method are carried out. This header comment should include the precondition of the method, if any. The **precondition** of method M is what each method that calls M must verify is true before making the call, in order that method M produce the expected result (i.e., the result described by its header comment).

This book will henceforth include the precondition in all listings where they are needed. The methods earlier in this chapter that had an unstated precondition are as follows:

- `moveTake` in Listing 2.2. Precondition: There is a slot after the current slot.

- `backPut` in Listing 2.2. Precondition: There is a slot before the current slot.
- `shiftThreeToFive` in Listing 2.5. Precondition: The sequence has at least two slots after the current slot.

The main method in Listing 2.3 fails if its sequence does not have four slots, but that is not a precondition -- `main` is called by the operating system, not by another method.

The not-operator

Sometimes you need to test to see if a slot is empty. The Vic class does not provide a separate condition to do that. But in Java, if you put an exclamation mark in front of a condition, it means the opposite of the condition. **`!`** `sam.seesCD()` means "**it is false that** sam sees a CD", i.e., the current slot is empty. (When you read a phrase involving `!` aloud, you could say "not" for the "!").

The first if-else statement in Listing 2.7, repeated below on the left, could be written instead as shown on the right to use this **not-operator** and have the same effect:

```
if (first.seesCD())
   first.takeCD();
else
   first.putCD();
```

```
if ( ! first.seesCD())
   first.putCD();
else
   first.takeCD();
```

Development of the hasTwoOnStack method

The first method defined here is `hasTwoOnStack`, a method that has the executor answer the question, "Are at least two CDs on the stack?" This particular question is to be asked only when the executor is at an empty slot (the precondition). An example of a statement that asks this question and makes use of the answer is the following:

```
if (sue.hasTwoOnStack())
   sue.moveOn();
```

The task is not simple to do, so you should design a plan in ordinary English to do it (or whatever natural language you think in most comfortably) before you try to implement it in Java. Write as if explaining it to a literal-minded not-very-bright person. A reasonable plan is shown in the accompanying design block.

DESIGN of the hasTwoOnStack method
If you do not have any CD at all on the stack, then...
    The answer is `false`; all further analysis is terminated.
Otherwise...
    Put a CD from the stack into the empty slot.
    Make a note of whether you have a CD on the stack at this point.
    Take the CD back onto the stack.
    The answer is whatever you made a note of; no further analysis is needed.

Return statements

When you define a new method asking a true-false question, such as `hasTwoOnStack`, you have to use a special statement to say whether the answer to the question is `true` or `false`. The statement `return false;` means execution of the method gives the answer `false` at that point and ignores the rest of the statements in that method. Similarly, the statement `return true;` means execution of the method gives the answer `true` at that point and ignores the rest of the statements. The key point: Executing a `return` means the rest of the statements in the method are skipped.

A Java implementation of this `hasTwoOnStack` method is in the upper part of Listing 2.8. It looks different from earlier methods in two ways: It has the word `boolean` and it has `return` statements (lines 2, 6, and 10). The word `boolean` in the heading says this method returns a true-false value ("boolean" commemorates a 19th-century logician named George Boole). So you may put a call of this **boolean method** inside the parentheses of an if-statement, as in `if (sam.hasTwoOnStack()) sam.putCD()`.

Listing 2.8 The Checker class of objects, with two boolean methods

```
public class Checker extends Vic
{
   /** Tell whether the stack has at least two CDs.
    *  Precondition: the executor is not at the end
    *  of its slots and the current slot is empty. */

   public boolean hasTwoOnStack()
   {  if ( ! stackHasCD())           // 1
         return false;               // 2
      putCD();                       // 3
      if (stackHasCD())              // 4
      {  takeCD();                   // 5
         return true;                // 6
      }                              // 7
      else                           // 8
      {  takeCD();                   // 9
         return false;               // 10
      }                              // 11
   }  //=====================


   /** Tell whether the executor's current slot and the one
    *  after it exist and have no CD.  No precondition. */

   public boolean seesTwoEmpty()
   {  if ( ! seesSlot())             // 12
         return false;               // 13
      if (seesCD())                  // 14
         return false;               // 15
      moveOn();                      // 16
      if (seesSlot())                // 17
      {  if ( ! seesCD())            // 18
         {  backUp();                // 19
            return true;             // 20
         }                           // 21
      }                              // 22
      backUp();                      // 23
      return false;                  // 24
   }  //=====================
}
```

Programming Style Surely you wondered why the executor should bother to take the CD back to the stack (using the if-else statement) before it returns `true` or `false`. The reason is, `someVic.hasTwoOnStack()` is a question you ask some Vic about its current state, and it is not good style to have the process of answering the question alter that state in any way. Quite often, alteration would make the answer useless.

Development of the seesTwoEmpty method

The second boolean method defined here is `seesTwoEmpty`, asking the executor whether both the current slot and the next slot exist and are empty. The task is not simple to do, so you should design a plan in ordinary English to do it before you try to implement it in Java. A reasonable plan is shown in the accompanying design block.

DESIGN of seesTwoEmpty
If you do not have a slot or if you see a CD in your current slot, then...
    The answer is `false`; all further analysis is terminated.
Move on to the next slot (if any).
If you have a slot there and if you do not see a CD in your current slot, then...
    The answer is `true`.
Otherwise...
    The answer is `false`.
Return the answer determined in the previous step, after backing up to the original position.

The definition of the `seesTwoEmpty` method is in the lower part of Listing 2.8. Note that, after the answer to the question is determined, the executor should back up to its original position (lines 19 and 20) so answering the question about its state does not change that state. Suppose you use these methods in some instance method as follows:

```
if (seesTwoEmpty())
{  if (hasTwoOnStack())
   {  putCD();
      moveOn();
      putCD();
   }
}
```

This logic makes sure you can safely put a CD in each of the next two slots. Imagine your surprise if the program were to fail because verifying the safety of the actions was what made those actions unsafe. That is why the executor should restore its state.

Note: Execution of a `return` statement terminates all action in the current method. Therefore, the `else` in the `hasTwoOnStack` logic can be omitted (and its matching braces as well) without affecting what the method does.

Design before you implement in Java

The written designs you have seen for `hasTwoOnStack` and `seesTwoEmpty` give a structure to the ordinary English sentences when a selection of several alternatives is to be made. Specifically, they show the alternatives indented below the conditions that decide which alternative is to be chosen. This kind of **logic design** is highly effective in helping you create individual methods that work correctly the first time.

Whenever you have to develop a method to accomplish a task, and the logic is not immediately obvious, you should first design a plan as illustrated here, in accordance with the general programming principle, "If you do not know where you are going, you are not likely to get there."

The translation of a design into a particular programming language is called **coding**. The translation of "Look before you leap" into programming is, "Design before you code."

Language elements		
A Declaration can be:	public boolean MethodName ( ) { StatementGroup }	
A Condition can be:	! Condition	e.g., !seesCD()
or:	true	
or:	false	
A Statement can be:	return Condition ;	

Exercise 2.24 Write a method `public boolean hasNoSlot()` for a subclass of Vic to mean the opposite of `seesSlot`.
Exercise 2.25 Write a method `public boolean canTakeCD()` for a subclass of Vic: The executor tells whether it could take a CD from its current position if told to do so.
Exercise 2.26 Write a method `public boolean canPutCD()` for a subclass of Vic: The executor tells whether it could put a CD in its current position if told to do so.
Exercise 2.27* Write a method `public void putNextAvailable()` for a subclass of Vic: The executor puts a CD in the next available slot. Precondition: The stack is not empty and either the current slot or the next slot is empty.
Exercise 2.28* Write a method `public boolean seesTwoFilled()` for a subclass of Vic: The executor tells whether its current slot and the next slot exist and both have CDs. Leave the executor in its original state.
Exercise 2.29* Write a method `public boolean hasJustOneOnStack()` for a subclass of Vic: The executor tells whether exactly one CD is on the stack. Precondition: Its current slot exists and is empty. Leave the executor in its original state.

2.7 Boolean Variables And The Assignment Operator

A method that effects a change in the state of one or more objects and does not return a value is an **action method**. Commands such as `sam.moveOn()` and `sue.putCD()` call action methods in the Vic class. These commands tell their executor to do something.

A method that returns a value and does not effect a change in the state of any object (executor or otherwise) is a **query method**. Commands such as `Vic.stackHasCD()` and `sam.seesSlot()` call query methods in the Vic class. These commands ask their executor a question and get an answer in return, without changing the state of the executor.

Technical Note This is not official Java vocabulary, just words to help you see patterns and categories: A method call can be a message to the "executor" to answer a "query" or perform an "action". An action method is a method with "void" just before the method name in the heading. But "void" is not a description of the method, it is just the signal to the compiler that no value is to be returned.

The `hasTwoOnStack` method in Listing 2.8 is a query method because care was taken to put the CD back on the stack before returning. This restored the original state. A true query method answers a question without making any change in any object. Similarly, `seesTwoEmpty` is a query method because care was taken to restore the original position in the sequence. However, the logic in those two methods is clumsy and a bit difficult to follow. It is time you learned about boolean variables.

You have seen how to declare the name of a variable that holds a Vic object. You can also declare the name of a variable that holds a boolean value (i.e., a value that is either `true` or `false`). For instance, `boolean theAnswer` declares `theAnswer` as the name of a place where either `true` or `false` can be stored. You can then have the command `return theAnswer` in a boolean method that has declared `theAnswer` and given it a value of `true` or `false`.

Declaring a variable name within a method does not by itself give the variable a value. You have to **assign** a value to it using the assignment symbol =, which you have seen used in statements such as `sue = new Vic()`. You should almost always assign a value to a variable by the very next statement after you declare it. You can later change the value assigned to the variable if you wish.

Rewrite of the hasTwoOnStack method to use a boolean variable

For the `hasTwoOnStack` method in Listing 2.8, after `putCD()` is executed, the value of `stackHasCD()` is either `true` or `false`. And that is the value to be returned. It is clearer to store the value of `stackHasCD()` in a boolean variable (named perhaps `result`), then execute `takeCD()` before returning the value of `result`.

The upper part of Listing 2.9 shows how the `hasTwoOnStack` method can be rewritten using a boolean variable. The assignment to `result` in the method (line 5) could be written instead as follows, with exactly the same effect as `result = stackHasCD()`, but it is wasteful to do so:

```
if (stackHasCD())
   result = true;
else
   result = false;
```

Listing 2.9 Rewrites of the methods in the Checker class

```
public class Checker extends Vic   // improved version
{
   /** Tell whether the stack has at least two CDs.
    *  Precondition: the executor is not at the end
    *  of its slots and the current slot is empty. */

   public boolean hasTwoOnStack()
   {  if ( ! stackHasCD())                  // 1
         return false;                      // 2
      putCD();                              // 3    modify state
      boolean result;                       // 4
      result = stackHasCD();                // 5
      takeCD();                             // 6    restore state
      return result;                        // 7
   }  //=====================

   /** Tell whether the executor's current slot and the
    *  one after it exist and have no CD.  No precondition. */

   public boolean seesTwoEmpty()
   {  if ( ! seesSlot())                    // 8
         return false;                      // 9
      if (seesCD())                         // 10
         return false;                      // 11
      moveOn();                             // 12   modify state
      boolean valueToReturn;                // 13
      if ( ! seesSlot())                    // 14
         valueToReturn = false;             // 15
      else                                  // 16
         valueToReturn = ! seesCD();        // 17
      backUp();                             // 18   restore state
      return valueToReturn;                 // 19
   }  //=====================
}
```

Rewrite of the seesTwoEmpty method to use a boolean variable

In the earlier `seesTwoEmpty` method definition, after `moveOn()` is executed, the value to be returned is `true` if `seesSlot()` is `true` and `seesCD()` is `false`, otherwise the value to be returned is `false`. It is clearer to store that true/false value in a boolean variable (named perhaps `valueToReturn`), then execute `backUp()` before returning the value of `valueToReturn`. This is done in the lower part of Listing 2.9.

You should carefully compare the coding in Listing 2.9 with the logic in the earlier Listing 2.8 to see the difference. In each case, the first few lines are the same, down to where the state of the executor changes. The line numbers are so future discussion can refer to those lines.

The crucial difference between Listing 2.8 and Listing 2.9 for the `seesTwoEmpty` method occurs after `moveOn`: In Listing 2.8, if `seesSlot()` is `true` and `seesCD()` is `false`, the executor backs up and returns `true`, otherwise the executor backs up and returns `false`. By contrast, in Listing 2.9, if `seesSlot()` is `true` and `seesCD()` is `false`, the executor simply stores `true` in the variable, otherwise the executor stores `false` in the variable (lines 14-17). Then, in either case, the executor backs up and returns whatever value it previously stored in the variable (line 19).

Programming Style All methods that return a value are to restore the original state of all objects before returning, unless explicitly stated otherwise (and then only if there is a very good reason). This is an important point of good style.

Pictorial representation of values in variables

Figure 2.8 shows a picture of what might be stored in RAM after you (a) create an object for `pam` to refer to (indicated by an arrow from `pam` to the internal description of the object), and (b) assign `true` to a boolean variable named `result`. The Vic object keeps track of the stack of CDs, its sequence of slots, and its position in the sequence. A newly-created Vic always starts at position number 1.

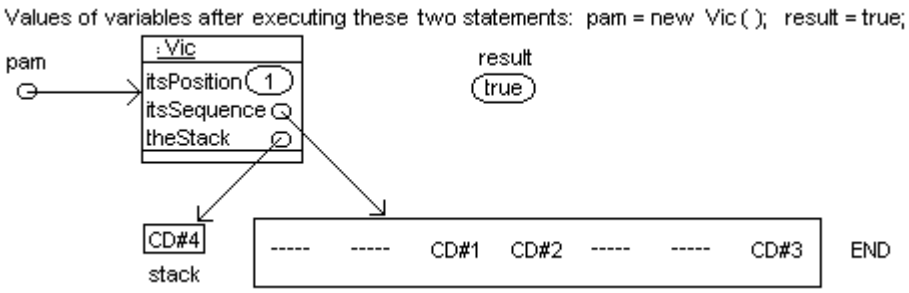


Figure 2.8 Object references as arrows to the objects

Each rectangle in the figure represents one variable. An arrow is shown leaving each variable which refers to an object. A standard notation for an object is a box with three parts, where the top part has the class of the object underlined, the middle part lists variables which are part of the object, and the bottom part is often left blank.

Are you skipping many of the unstarred exercises? That is not a good idea, you know. You cannot just read this material as if it were a novel. You will start to become confused by so many new concepts. At a minimum, spend just two minutes on each unstarred exercise and then, done or not, check out its answer at the end of the chapter. This will take less than ninety minutes per chapter, you will learn a lot more, and that learning will help you complete your graded course assignments faster and better.

Language elements		
A Statement can be:	boolean VariableName ;	e.g., boolean valueToReturn;
A Condition can be:	VariableName	e.g., valueToReturn

Exercise 2.30 Rewrite the `hasTwoOnStack` method of Listing 2.9 to have just one return statement. Precondition: Its current slot exists and is empty. Hint: You will need `else` followed by braces.
Exercise 2.31 Write a query method `public boolean atMostOneOnStack()` for a subclass of Vic: The executor tells whether the stack has either zero or one CD on it. Precondition: Its current slot exists and is empty. Only have one return statement.
Exercise 2.32* Write a query method `public boolean hasOneBefore()` for a subclass of Vic: The executor tells whether a CD is in the slot before the current slot. Have only one return statement. Precondition: The executor is not at the first slot in its sequence.
Exercise 2.33** Write an application program to move the first two available CDs in the first sequence of slots to the stack. Precondition: That sequence has at least two CDs in its first four slots. Hint: First create a boolean variable `alreadyTookCD`. Make it `true` if you take the first CD and `false` if not. Use it to decide what to do at later slots.

2.8 Boolean Operators And Expressions; Crash-Guards

If you put **&&** between two true-false expressions it means "and". That is, the combined expression is true only when both of its two parts are true. The two parts are called the **operands** of `&&`. Now you can rewrite these lines 14-17 of Listing 2.9

```
if ( ! seesSlot())
   valueToReturn = false;
else
   valueToReturn = ! seesCD();
```

more compactly and more clearly, but with exactly the same effect, as follows:

```
valueToReturn = seesSlot() && ! seesCD();
```

If you put **||** between two true-false expressions, it means "or". That is, the combined expression is true when either the first operand of `||` is true or the second operand of `||` is true, or both. Now you can rewrite these lines 8-11 of Listing 2.9

```
if ( ! seesSlot())
   return false;
if (seesCD())
   return false;
```

more compactly and more clearly, but with exactly the same effect, as follows:

```
if ( ! seesSlot() || seesCD())
   return false;
```

The three symbols `&&` and `||` and `!` are called **boolean operators** because they operate on boolean (true-false) expressions to produce a boolean expression. The `!` operator has only one operand, but `&&` and `||` each require two operands.

Short-circuiting boolean expressions

In the expression `seesSlot() && ! seesCD()`, if it happens `seesSlot()` is `false`, the runtime system does not look at the operand after the `&&` and the result `false` is obtained. This is good, because otherwise the program would fail. And if `seesSlot()` is false in the expression `! seesSlot() || seesCD()`, the operand after the `||` is not looked at and the result `true` is obtained, so again the program does not fail.

In each case, the first operand of the expression is a "crash-guard" for the second operand, in that it prevents evaluation of the second operand in precisely those situations where the evaluation would crash the program. The `&&` and `||` operators **short-circuit** the condition they form: The second operand of an `&&` or `||` expression is not evaluated if the first operand by itself determines its truth or falseness. From this discussion you can see that the `seesTwoEmpty` method of Listing 2.9 can be written more compactly as shown in Listing 2.10.

Listing 2.10 Improved replacement for the seesTwoEmpty method in Checker

```
public boolean seesTwoEmpty()
{  if ( ! seesSlot() || seesCD())
      return false;
   moveOn();
   boolean valueToReturn;
   valueToReturn = seesSlot() && ! seesCD();
   backUp();
   return valueToReturn;
}  //=====================
```

Caution Put parentheses around an expression formed with `||` or `&&` if the expression does not stand alone as an if-condition or as something assigned to a boolean variable. That is, it should have parentheses around it when it is an operand of `||` or `!` or `&&`. `(x || y) && z` is different from `x || (y && z)`. Also, if you want `!` to apply to an expression that is not a simple method call or variable, you should put parentheses around that expression. There are times when the parentheses are not needed, but it is safer to always use them in such cases.

Language elements

A Condition can be:	Condition && Condition
or:	Condition || Condition

Exercise 2.34 Rewrite the answer to Exercise 2.25 (`canTakeCD`) so that the body is a single return statement.
Exercise 2.35 Rewrite the answer to Exercise 2.26 (`canPutCD`) so that the body is a single return statement.
Exercise 2.36 Explain why the following causes at least two different compilation errors:

```
if(sam.seesSlot)
   sam.putCD();
   sam.moveOn();
else
   sam.takeCD();
```

Exercise 2.37* Rewrite `seesTwoEmpty` in Listing 2.10 so it contains only one return statement.
Exercise 2.38** Write a method `public void shiftDown()` for a subclass of Vic: The executor moves the CD in its current slot into the following slot, except no change is to be made at all if there is no empty following slot or if there is no CD in the current slot. Avoid all program failures. Use boolean operators where appropriate.

2.9 Getting Started With UML Class Diagrams And Object Diagrams

The standard method for drawing a model of a program uses the Unified Modeling Language, **UML** for short. A **class diagram** is a picture that shows the classes the program uses and some relations between them. For instance, Figure 2.9 is a class diagram for the TwoToFour application program in the earlier Listing 2.4.

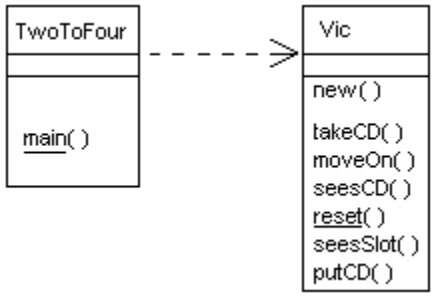


Figure 2.9 UML class diagram for TwoToFour

This diagram has two class boxes with a dependency indicated by a dotted-line arrow. A **class box** is a rectangle with three parts separated by horizontal lines. A **dependency** means the one class uses the methods in the class pointed to.

The top part of the class box contains the name of the class (not underlined, to distinguish it from an object box as shown in Figure 2.8). The middle part (between the two lines) is for attributes. It is blank in this diagram. The bottom part lists the method calls used in the program. The names of class methods are to be underlined.

UML generalization

Figure 2.10 shows the class diagram for the program BringThreeBack in the earlier Listing 2.3. The diagram shows the two classes mentioned in this program, namely, BringThreeBack itself plus the SmartVic class. Since the `moveOn` and `reset` methods that BringThreeBack uses are inherited from the Vic class, the class diagram shows the Vic class too. The solid line with the triangular head signals that SmartVic inherits from Vic. UML calls it a **generalization**.

The three arrows in the class diagram can be read as follows:

1. BringThreeBack uses Vic.
2. BringThreeBack uses SmartVic.
3. SmartVic is a kind of Vic.

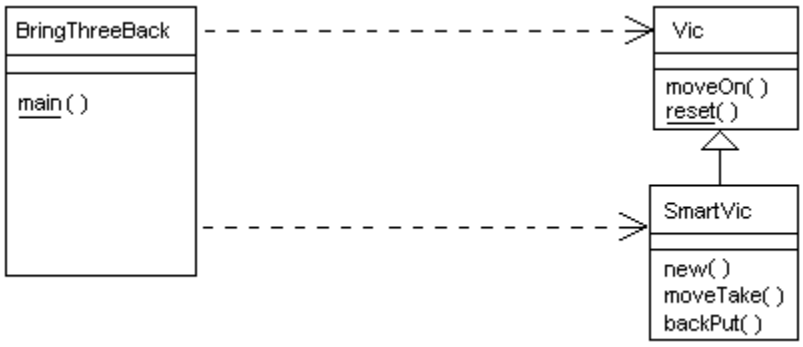


Figure 2.10 UML class diagram for BringThreeBack

UML allows you a great deal of choice in how much detail you choose to show in a class diagram. The class diagrams shown in this section are at the minimum level of detail. Chapter Three describes some additional choices you have, but you should first practice this minimum-level kind of class diagram for a while, defined as follows:

1. Write down a list of all of the methods mentioned in the class or classes you want to diagram (in the examples, we only want to diagram one application program).
2. Draw a rectangle for each class in which one or more of those methods are defined.
3. Divide the rectangle horizontally into three parts, with the name of the class in the top part.
4. List each method from step 1 in the bottom part of the rectangle for the class that contains its definition. Underline `main` and other class methods (such as `reset`).
5. Draw a solid-line-triangle-arrowhead from a class to its superclass if both appear in the diagram.
6. Draw a dotted-line-arrow from a class to each other class (besides its superclass) whose methods or instances it mentions.

UML is also used to diagram individual objects. The earlier Figure 2.8 is a **UML object diagram**. For that diagram, an alternate way of representing the object `pam` refers to is to put `pam:Vic` in the top part of the box and omit the arrow from `pam`. That is, a rectangle that represents a particular Vic object has `:Vic` or `variable:Vic` underlined in the top part of the rectangle; a rectangle that represents a class just has the class name and it is not underlined.

Client-server relations

When one class provides services (generally methods) that another class uses, the first class is said to be a **server** for the second class, and the second is said to be a **client** of the first. In this client-server relation, for instance, the client may have parameters that are instances of the server class, or may return instances of the server class, or may use the methods of the server class locally within its coding. Both dependencies and generalizations are kinds of client-server relations.

Each method that provides a service should have a descriptive comment saying what its effect is in all situations in which the method can be called (except when the name of the method alone makes it quite clear). This is the **specification** for the method. A **fault** in a program (colloquially known as a **bug**) is a failure of a method to do in some situation what its specification says it does in that situation.

A precondition on a method states a condition that must be true when the method is called. If some method M calls a method P when P's precondition is false, then it is a bug in M, not in P. Essentially, a precondition shifts the responsibility for verifying something from the called method to the calling method. So it is a deficiency in P to verify its preconditions and a fault in M to not verify P's preconditions. An implied part of each method's specification is that it does not call any method in a situation when the called method's precondition is false.

For example, if the coding for a method begins with `if ( ! seesSlot()) say...` then its specification should not say "Precondition: its current slot exists"; it should instead say "If its current slot does not exist, say... otherwise...".

Exercise 2.39 Draw the UML diagram for the program in Listing 2.1.
Exercise 2.40* Draw the UML diagram for the program of Exercise 2.9.

Part B Enrichment And Reinforcement

2.10 Analysis And Design Example: Complex Conditionals

Perhaps you have a need to write a method to add to a subclass of Vic, to remove the first two CDs from a sequence known to have at least two CDs in the next three slots. Since this is not a simple thing to do, you should begin with an analysis of the situation:

Question 1: What do you do first? *Answer:* It depends.
Question 2: Depends on what? *Answer:* On whether this first slot contains a CD.
Question 3: What do you do first if it does? *Answer:* Take the CD from there and then move on to the second slot.
Question 4: What do you do first if it does not? *Answer:* Move on to the second slot.

This gives you the following overall if-else design for your logic:

```
If the first slot contains a CD then...
   Take that CD out of the slot and move to the second slot.
   // do more in Case 1
Otherwise...
   Move to the second slot.
   // do more in Case 2
```

In both cases, you end up in the second slot. Next decide whether what you do in Case 1 is the same as what you do in Case 2. If that were so, you would handle that common case after the if-else statement (change the design by removing the two comments and putting `// do more either case` at the end). But that is not so.

Now you figure out what comes next in each case. Case 2 is the easier situation, since at this point you know a CD is in each of the next two slots (remember, the precondition is that you have at least two CDs in the three slots, and there was no CD in the first slot). So you know in Case 2 what actions to take without testing any more conditions:

```
// do more in Case 2 expands to:
Take the CD from the second slot.
Move to the third slot and take the CD from that slot.
```

In Case 1, you also have to move on to the second slot, because you have one CD left to find and it is in one of the next two slots. After you move on to the second slot, you realize what you do again depends on what you find. Analyze it with the same four key questions:

Question 1: What do you do first? *Answer:* It depends.
Question 2: Depends on what? *Answer:* On whether this second slot contains a CD.
Question 3: What do you do first if it does? *Answer:* Take the CD from there.
Question 4: What do you do first if it does not? *Answer:* Move on to the third slot and take the CD from that slot.

From this you can develop a design of the logic. Note that the design is in English, not in Java. You need to review the design and make sure it is logically correct and covers all of the possibilities. This is much easier to do in a language you know well, e.g., English.

```
// do more in Case 1 expands to:
If the second slot contains a CD then...
   Take the CD from that slot.
Otherwise...
   Move to the third slot and take the CD from that slot.
```

You should combine the parts of this design into one unit so you can study it further before you try to translate it to Java. The accompanying design block does this. One small change is made: Since Case 2 turned out to be shorter and simpler than Case 1, it is presented first in the design. Putting the simpler alternative first is normally clearer.

DESIGN of takeTwoOfThree

1. If you do not see a CD in the current slot then...
    1a. Move on to the second slot and take that CD.
    1b. Move on to the third slot (there must be a CD in that slot).
2. Otherwise...
    2a. Take the CD in the current slot.
    2b. Move on to the second slot.
    2c. If you do not see a CD in the second slot, then...
        Move on to the third slot (there must be a CD in that slot).
3. Take the CD in the current slot (which is either the second or the third slot).

After you work out the design in this form, you need to read it over several times, even recite it out loud and listen to it, to make quite sure there are no errors in the logic. Read again the specification of what the method is to do, to be sure your design fulfills the requirements. This can then be translated into Java as the method shown in Listing 2.11. Figure 2.11 shows which statements are actually executed for the three possibilities.

Listing 2.11 An instance method for a subclass of Vic

```
/** Take the first two available CDs.
 *  Precondition: the first 3 slots contain at least 2 CDs. */

public void takeTwoOfThree()
{  if ( ! seesCD())                          // design step 1
   {  moveOn();                              // design step 1a
      takeCD();
      moveOn();                              // design step 1b
   }
   else                                      // design step 2
   {  takeCD();                              // design step 2a
      moveOn();                              // design step 2b
      if ( ! seesCD())                       // design step 2c
         moveOn();
   }
   takeCD();                                 // design step 3
}  //======================
```

// THE CODING	CD IN 1ST AND 2ND	CD IN 1ST AND 3RD	CD IN 2ND AND 3RD
if (! seesCD())	! seesCD() is false	! seesCD() is false	! seesCD() is true
{ moveOn();			moveOn();
takeCD();			takeCD(); // slot 2
moveOn();			moveOn();
}			
else			
{ takeCD();	takeCD(); // slot 1	takeCD(); // slot 1	
moveOn();	moveOn();	moveOn();	
if (! seesCD())	! seesCD() is false	! seesCD() is true	
moveOn();		moveOn();	
}			
takeCD();	takeCD(); // slot 2	takeCD(); // slot 3	takeCD(); // slot 3

Figure 2.11 Three possible sequences of actions for Listing 2.11

Now read this section down to here one more time and note well the process described here. For all but the simplest Java methods, if you apply this process to code a method to perform a given task, you will find that you spend somewhat longer before you get to the coding part but that you spend far less time getting the coding part right.

The Dangling Else Trap

Look at the following two methods that could be in a subclass of Vic. The first three lines of the bodies are the same in both cases; they differ beginning with `else`:

```
public void sayWhenSlotEmpty()
{  if (seesSlot())
      if (seesCD())
         takeCD();
      else
         Vic.say ("No CD!");
}  //======================
```

```
public void sayWhenNoSlot()
{  if (seesSlot())
      if (seesCD())
         takeCD();
   else
      Vic.say ("No slot!");
}  //======================
```

From the indentation and the method names, it is clear that the first method is to print a message only when there is a slot that contains a CD. And the second method is to print a message only when there is no slot at all (i.e., the Vic is at the end of its sequence). The first method behaves as it is intended. But the second method behaves just like the first method except for the words in the message.

General principle: The compiler pays no attention to indentation. Indentation is only to make it easy for you and other people to understand your logic. And the compiler does not understand the words inside the calls of `Vic.say`. So the compiler has no choice but to interpret these two methods the same; they have the same structure. The compiler's rule in such a case is to match `else` with the most-recently occurring `if`.

Caution It is best never to write `if (Condition) if` in your logic. Put braces around the subordinate if-statement. Doing this guarantees you will never fall prey to this **Dangling Else Trap**. Applying this rule to the two preceding examples gives the following (mark ye well the position of the braces):

```
public void sayWhenSlotEmpty()
{  if (seesSlot())
   {  if (seesCD())
         takeCD();
      else
         Vic.say ("No CD!");
   }
}  //======================
```

```
public void sayWhenNoSlot()
{  if (seesSlot())
   {  if (seesCD())
         takeCD();
   }
   else
      Vic.say ("No slot!");
}  //======================
```

Multiway selection format

The `verbosePutCD` method shown in Listing 2.12 has an if-else structure that occurs with some frequency in developing methods. This logic prints a different message in each of four different cases. Only in the fourth case does it actually put a CD in the slot.

Listing 2.12 An instance method illustrating multiway selection

```
/** Try putCD().  Print a description of the situation. */

public void verbosePutCD()   // in a subclass of Vic
{  if ( ! seesSlot())
      Vic.say ("There is no slot");
   else if (seesCD())
      Vic.say ("The slot is full");
   else if ( ! stackHasCD())
      Vic.say ("I have no CD");
   else
   {  putCD();
      Vic.say ("Mission accomplished!");
   }
}  //======================
```

The indentation used in this method is the **Multiway selection format**. It makes it clear to the reader that exactly one of the four cases will be selected. Standard indentation would require two more lines and indent two additional levels. That is not as readable.

Some programming languages offer a special multiway selection statement using e.g. the special word ELSIF. The designers of Java apparently felt programmers could just use the normally-two-way if-else with the modified indentation -- form follows function.

Programming Style There are two commonly-used styles in the placement of braces after an if- or while- condition, or method heading, or the equivalent. One is the Allman style, used in this book. The other is the Kernighan & Ritchie style, where the opening brace is placed at the end of the line that has the `if` or `while` or method name. It is a matter of personal preference which one you use. It is also a matter of preference whether you have a blank line (other than perhaps the beginning brace) between the controlling phrase and the subordinate part. However, it is wise to use the Allman style until late in your first course of programming, because beginners find it easier to have their braces match up when they are vertically lined up with each other and everything between is indented.

Exercise 2.41 Rewrite the answer to Exercise 2.21 in the multiway selection format. Pay close attention to indenting.
Exercise 2.42 (harder) Write a method `public void downShift()` for a subclass of Vic: The executor takes the CD from the current slot and puts it in the very next slot. However, do not do anything if there is no current slot or no CD in it, and leave the CD on the stack if there is no following slot or it has a CD in it already.
Exercise 2.43* Look ahead to Listing 3.10 and rewrite the body of the lengthy while-statement in the multiway selection format. Pay close attention to indenting.
Exercise 2.44** Write an application program that moves the first available CD in the first sequence of slots to the stack. Do not go any further in the sequence than necessary. Precondition: The sequence is known to have at least one CD in its first four slots.
Exercise 2.45** Revise the `takeTwoOfThree` method in Listing 2.11 to remove all preconditions.

2.11 Review Of Chapter Two

Listing 2.5, Listing 2.9, and Listing 2.10 illustrate almost all Java language features introduced in this chapter.

About the Java language:

- The compiler ignores anything between `/*` and `*/`, even over several lines. By contrast, the compiler ignores anything after `//` up to the end of the physical line.
- You cannot "run" an object class that has no main method. You can only "run" a class with `java ClassName` when the class has a main method to run. You may test out an object class X (i.e., a class with instance methods) by executing some other class's main method that creates objects of type X and sends them messages.
- The words `true` and `false` are constants that indicate the only two possible boolean values. They are called "reserved words" in Java: You cannot use them to name a method, variable, or class, just as with the keywords `boolean` and `return`. A **condition** is an expression whose value is either `true` or `false`.
- The symbol for "not" is an exclamation point, for "and" is `&&`, and for "or" is `||`. In expressions involving two or more **boolean operators**, you should use parentheses liberally to make sure the meaning is clear. If the first **operand** `x` is true in `x || y`, or is false in `x && y`, then the second operand `y` is not evaluated, and the result of the expression is the value of `x`. This is **short-circuiting**.
- See Figure 2.12 and Figure 2.13 for the remaining new language features. `StatementGroup` means zero or more statements in sequence.

`if (Condition)` `  Statement`	**statement** that executes the subordinate Statement if the Condition is true

`if (Condition)`
`  Statement`
`else`
`  Statement`
	statement that executes only the first Statement if the Condition is true, but executes only the second Statement if the Condition is false

`{  StatementGroup` `}`	**statement** that can replace the one Statement in ifs and if-elses and others
`ClassName.MethodName (Expression);`	**statement** that calls a class method with one parameter
`boolean VariableName;`	**statement** that creates a true-false variable
`VariableName = Condition;`	**statement** that assigns a value of true or false to a boolean variable
`return Condition;`	**statement** that returns a true-false value; it terminates the method it is in

Figure 2.12 Statements introduced in Chapter Two

`public boolean MethodName()`
`{  StatementGroup`
`}`
	declaration of a method that returns a true-false value, i.e., a Condition

Figure 2.13 Method declaration introduced in Chapter Two

Caution Do not put any statement directly after a `return` statement if it is appropriately indented the same (i.e., is subordinate to the same condition). Such a statement can never be executed.

Other vocabulary to remember:

- The expression you put in the parentheses of a method call is a **parameter**, also known as an **argument** of the method call.
- The parts of an if-statement or if-else statement are named the **condition** (in parentheses after `if`), the first **subordinate part** (directly after those parentheses), and the second subordinate part (after `else` if present). If one of the subordinate parts has two or more statements, it should be enclosed in braces (to make it a **block statement**).
- The **javadoc** standard for commenting methods puts a description of each public method just before the method heading, beginning with `/**` and ending with `*/`.
- The **precondition** for a method is what has to be true when the method begins execution, in order that the method produce the expected result.
- The translation of a particular design into a compilable class is called **coding**. "Design before you code" is as basic a principle as "Think before you act."
- An **action method** changes the state of one or more objects but does not return a value. A **query method** returns a value but does not change the state of any object. Try to avoid methods that do both.
- You can convey the information that a choice is being made from among three or more alternatives by putting an `if` immediately after `else` on the same physical line, rather than indented on the following line. This **multiway selection format** gets around the limitations of using a language feature which offers only a choice from two alternatives.

About Vic methods (developed for this book):

- `new Vic()` creates a Vic object.
- You can send four action messages to a Vic: `sam.putCD()`, `sam.takeCD()`, `sam.moveOn()`, and `sam.backUp()`.
- You can ask two questions of a Vic: `sam.seesCD()` and `sam.seesSlot()`.
- You have three Vic class methods: `Vic.stackHasCD()`, `Vic.reset(args)`, and `Vic.say("whatever")`. The `reset` method uses whatever strings of characters follow `javac ProgramName` on the command line to initialize the sequences of CDs. But a random arrangement is used if there are no such **command-line arguments** or the `reset` method is not executed until after some Vic object has been created.
- If `sam.seesSlot()` is `false`, then `sam.putCD()`, `sam.takeCD()`, `sam.moveOn()`, and `sam.seesCD()` all cause the program to fail (i.e., gracefully terminate execution). Also, the `backUp` message causes the program to fail if the Vic object is positioned at its first slot. You should avoid letting this happen. If you let it happen, your program is not **robust**, since it does not handle unexpected input well. Ideally, all application programs should be written so that they cannot fail.

About UML notation:

- A **UML class diagram** uses **class boxes** -- rectangles divided into three parts. The top part has the class name and the bottom part lists any of its method calls you wish to mention, with `main` and other class methods underlined.
- A **dependency** of the form `X uses Y` is indicated by an arrow with a dotted line.
- A **generalization** of the form `X is a kind of Y` is indicated by an arrow with a solid line and a big triangular head.
- A **UML object diagram** also uses three-part rectangles. The top part has a colon followed by the class name, optionally with the name of the variable that refers to it before the colon. The middle part generally lists attributes of the object.

Answers to Selected Exercises

2.1
```
public class ThirdToFirst
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        Vic cat;
        cat = new Vic();
        cat.moveOn();       // to slot 2
        cat.moveOn();       // to slot 3
        cat.takeCD();
        cat.backUp();       // to slot 2
        cat.backUp();       // to slot 1
        cat.putCD();
    }
}
```

2.2
```
public class PutThree
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        Vic cat;
        cat = new Vic();
        cat.putCD();
        cat.moveOn();       // to slot 2
        cat.putCD();
        cat.moveOn();       // to slot 3
        cat.putCD();
    }
}
```

2.3
```
public class TakeSecondAndFourth
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        Vic cat;
        cat = new Vic();
        cat.moveOn();       // to slot 2
        cat.takeCD();
        cat.moveOn();       // to slot 3
        cat.moveOn();       // to slot 4
        cat.takeCD();
    }
}
```

2.6
```
public static void main (String [ ] args)  // rewrite of Exercise 2.1
{   Vic.reset (args);
    SmartVic cat;
    cat = new SmartVic();
    cat.moveOn();
    cat.moveTake();     // to slot 3
    cat.backUp();
    cat.backPut();      // to slot 1
}
```

2.7 Put "cat.moveTake();" in place of the two lines beginning at the comment // to slot 2 and also in place of the two lines beginning at the comment // to slot 3

2.8
```
public void movePut()
{   moveOn();
    putCD();
}
```

2.9
```
public class ThirdToFifth
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        SmartVic cat;
        cat = new SmartVic();
        cat.moveOn();       // to slot 2
        cat.moveTake();     // to slot 3
        cat.moveOn();       // to slot 4
        cat.movePut();      // to slot 5
    }
}
```

2.13 Replace the last two statements by the following:
```
sue.moveOn();
if (sue.seesSlot())
{   sue.takeCD();
    sue.backPut();
}
```

```
2.14    public class TakeFourthAndFifth
        {   public static void main (String [ ] args)
            {   Vic.reset (args);
                Vic cat;
                cat = new Vic();
                cat.moveOn();
                cat.moveOn();
                cat.moveOn();           // to slot 4
                if (cat.seesSlot())
                {   cat.takeCD();
                    cat.moveOn();       // to slot 5
                    if (cat.seesSlot())
                        cat.takeCD();
                }
            }
        }
2.15    public void swapTwo()
        {   if (seesSlot())
            {   if (seesCD())
                {   moveOn();
                    if (seesSlot())
                    {   if (seesCD())
                        {   takeCD();
                            backUp();
                            takeCD();
                            moveOn();
                            putCD();
                            backUp();
                            putCD();
                        }
                    }
                }
            }
        }
2.21    if (sam.seesCD())
            sam.moveOn();
        else
        {   if (Vic.stackHasCD())
                sam.putCD();
            else
                sam.moveOn();
        }
2.22    public void shiftForward()
        {   if (seesCD())
                moveOn();
            else
            {   moveOn();
                if (seesSlot())         // this test must be made first
                {   if (seesCD())
                    {   takeCD();
                        backUp();
                        putCD();
                        moveOn();
                    }
                }
            }
        }
2.24    public boolean hasNoSlot()
        {   if (seesSlot())
                return false;
            else
                return true;
        }
        Or you could just have the one statement  return ! seesSlot();.
2.25    public boolean canTakeCD()
        {   if (seesSlot())
            {   if (seesCD())
                    return true;
            }
            return false;
        }
```

2.26
```
public boolean canPutCD()
{       if ( ! seesSlot())
                return false;
        if (seesCD())
                return false;
        if (stackHasCD())
                return true;
        else
                return false;
}
```
2.30
```
public boolean hasTwoOnStack()
{       boolean result;
        if ( ! stackHasCD())
                result = false;
        else
        {       putCD();
                result = stackHasCD();
                takeCD();
        }
        return result;
}
```
2.31 This is just the opposite of the preceding exercise, so replace its two assignments by:
```
result = true;
result = ! stackHasCD();
```
2.34
```
public boolean canTakeCD()
{       return seesSlot() && seesCD();
}
```
2.35
```
public boolean canPutCD()
{       return (seesSlot() && ! seesCD()) && stackHasCD();
}
```
2.36 The empty parentheses are missing at the end of the sam.seesSlot method call.
Also, you must put the two statements subordinate to if inside matching braces.

2.39 It is the same as Figure 2.9 except that you remove three items from the Vic box (reset, seesSlot, seesCD) and add backUp(). Also replace TwoToFour by MoveOne.

2.41
```
if (sam.seesCD())
        sam.moveOn();
else if (Vic.stackHasCD())
        sam.putCD();
else
        sam.moveOn();
```
2.42
```
public void downShift()
{       if (seesSlot() && seesCD())
        {       takeCD();
                moveOn();
                if (seesSlot())
                        putCD();
        }
}
```