3 Loops and Parameters

Overview

The commands in the Vic class operate physical clamps and springs to move the CDs around. In the context of this Vic software, you will learn about several new Java features:

- Section 3.1 presents the while-statement, which can be used to send a Vic to the end of its sequence. In general, a while-statement allows you to repeat an action many times.
- Section 3.2 explains the `equals` method in the Sun standard library String class.
- Sections 3.3-3.6 introduce more language features: private methods, the default executor, and parameters for passing extra information to a method. You only need study through Section 3.6 to understand the material in the rest of this book.
- Sections 3.7-3.8 describe and illustrate a highly reliable method for developing the logic of complex methods relatively quickly and with few errors.
- Sections 3.9-3.10 cover enrichment topics: Turing machines and Javadoc tags.

3.1 The While Statement

The `fillFourSlots` method that follows asks the executor to put a CD in the next four available slots, except that the four if-statements make the method stop early if the executor runs out of slots:

```
public void fillFourSlots()      // in a subclass of Vic
{  if (seesSlot())
   {  putCD();
      moveOn();                  // move to slot 2
      if (seesSlot())
      {  putCD();
         moveOn();               // move to slot 3
         if (seesSlot())
         {  putCD();
            moveOn();            // move to slot 4
            if (seesSlot())
               putCD();
         }
      }
   }
}  //======================
```

Three of the four if-statements have a block statement for the subordinate part, i.e., several statements enclosed in a matching pair of braces. When an if-condition is false, none of the statements within its block will be executed. Clearly, the method would be quite lengthy if you wanted to put a CD in each of the first six slots. And what if you wanted to put a CD in every single slot?

The following method contains a new Java statement that repeats its two inner statements any number of times, until the condition `seesSlot()` becomes false. It fills every slot possible. Note that it is far shorter than `fillFourSlots`, even though it does far more:

```
public void fillSlots()       // in a subclass of Vic
{   while (seesSlot())
    {   putCD();
        moveOn();
    }
}   //======================
```

The usual format of a **while-statement** is:

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

Subordinate statements

The statements in the block are **subordinate** to the while. If you have only one subordinate statement in the block you may omit the braces, as for an if-statement. Note that we boldface the word `while` as a signal it requires a subordinate statement.

The condition in the parentheses after `while` is the **continuation condition**. The meaning of a while-statement can be expressed as follows:

1. If the continuation condition is true, then
    1a. Execute all subordinate statements in sequence.
    1b. Repeat this process from Step 1.

Figure 3.1 gives a pictorial description of this action.

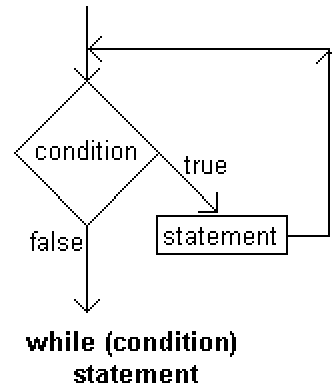


Figure 3.1 Flow-of-control for the while-statement

Examples of the while-statement

The `toggleCDs` method in Listing 3.1 gives another illustration of the while-statement. While moving to the end of the sequence, it switches the status of each slot by (a) removing a CD if the slot contains one, otherwise (b) putting a CD in the slot if the stack contains one.

Listing 3.1 A method that uses the while statement

```
public void toggleCDs()       // in a subclass of Vic
{   while (seesSlot())
    {   if (seesCD())
            takeCD();
        else
            putCD();
        moveOn();
    }
}   //======================
```

Programming Style The following statement as the subordinate part of the while-statement in `toggleCDs` would do exactly the same thing, but it is not good style to have the same statement at the end of both alternatives of an if-else statement. Such a statement should be "factored out" as in Listing 3.1:

```
if (seesCD())   // unfactored if-else statement
{  takeCD();
   moveOn();
}
else
{  putCD();
   moveOn();
}
```

The following main method empties the first slot of every sequence. Each execution of `sequence = new Vic()` makes `sequence` refer to the next sequence of slots. When `sequence.seesSlot()` is `false`, you have no more sequences to process:

```
public static void main (String[ ] args)
{  Vic sequence;
   sequence = new Vic();

   while (sequence.seesSlot())
   {  sequence.takeCD();
      sequence = new Vic();
   }
}  //======================
```

Programming Style It is good style to indent the subordinate part of an if-statement or while-statement by one extra tab position. That makes it clear to someone who reads the program what statements are subordinate to what other statements. This is especially useful with if-statements within if-statements or within while-statements. Note that all of the source code in this book indents after each boldfaced word.

Language elements

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

Note: All instance methods in the exercises for Chapter Three that ask you to do something for all of the executor's slots apply only to the current slot and those after it. For instance, Exercise 3.1 means the executor removes all CDs from this and later slots. You ignore any previous slots when you have no way to tell whether they exist.

Exercise 3.1 Write a method `public void removeAllCDs()` for a subclass of Vic: The executor removes all the CDs from its slots and puts them on the stack. Leave the executor at the end of its sequence of slots.
Exercise 3.2 Write a method `public void toLastSlot()` for a subclass of Vic: The executor advances to the last slot in its sequence. Precondition: The executor has at least one slot somewhere. Hint: Go to the end, then go back by one slot.
Exercise 3.3 (harder) Write a method `public void takeOneBefore()` for a subclass of Vic: The executor backs up until it sees a slot with a CD in it, then takes it. Precondition: There will be a filled slot somewhere before the current position of the executor.
Exercise 3.4* Write an application program that fills the first slot of every sequence of slots for which the first slot is empty. Stop when the stack becomes empty.
Exercise 3.5* Write a method `public void fillOneSlot()` for a subclass of Vic: The executor advances to the next available empty slot and puts a CD in it. If this is impossible, just have the executor advance to the next available empty slot or, if all slots are filled, to the end of the sequence. "Next available" includes the current slot.

3.2 Using The Equals Method With String Variables

The `fillSlots` method of the previous section leaves the Vic at the end of its sequence of slots. It cannot back up to the beginning, because it would not know when to stop. That makes it a not very useful Vic. This can be fixed by having the executor make a note of its current position before going through the sequence to fill the slots. Then it will be able to back up to that initial position.

The method call `sam.getPosition()` returns an object that records the current position of the Vic named `sam`. This object is a string of characters. **String** is the name of the Sun standard library class for such objects. The String class contains a method for testing whether two Strings are equal. These new methods are described in Figure 3.2.

aVic.getPosition()
produces a String object recording the current position of the object referred to by aVic.

aString.equals (someOtherString)
tests whether one String is equivalent to another. The String that getPosition() produces equals another String it has produced whenever they represent the same position in the same sequence, even if the actual String objects are different.

Figure 3.2 Two query methods for use with positions in a sequence

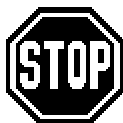
Now the `fillSlots` method can be revised so the executor is at the same position in its sequence at the end of execution that it was in at the beginning of execution. Put these two statements at the beginning to make a note of the current position in a variable:

```
String spot;
spot = getPosition();
```

Then put these statements at the end of the `fillSlots` method, after the while-statement that moves the executor down to the end of the sequence:

```
while ( ! spot.equals (getPosition()))
   backUp();
```

This logic checks whether `spot` (the position when `fillSlots` began execution) is the same as the current position (given by the call of `getPosition()`) and if not, backs up by one slot and then repeats the check for equality. When they are equal, the loop stops (a **loop** is the repeated execution of a group of statements; an **iteration** of a loop is one such execution of the group of statements).



Caution Always review the logic of each while-loop you write to be sure it will terminate eventually when executed. Nota Bene: Control-C in the terminal window kills the entire program. This is useful to know when you run a program with a loop that never terminates.

The hasSomeFilledSlot method

We will define two new methods so you can move a Vic object down its sequence of slots to the last slot that contains a CD, assuming there is such a slot. You will be able to have logic in a program something like the following:

```
if (sam.hasSomeFilledSlot())
   sam.goToLastCD();
```

The `hasSomeFilledSlot` method is to send a message that asks the executor whether any slot from its current position on down has a CD in it. An initial sketch of a plan to do this is: The executor goes down its sequence until it sees a filled slot (in which case it returns `true`) or it runs out of slots to look in (in which case it returns `false`). But it returns to its original position before returning the answer to the question. You can then refine this initial sketch as shown in the accompanying design block.

DESIGN of hasSomeFilledSlot

1. Make a note of the current position.
2. Move down the sequence until you see a filled slot or you run out of slots.
3. Let `valueToReturn` record whether, at that point, there is still a slot left.
4. Back up until you get to the original position as of the start of this method.
5. Return the value in `valueToReturn` as the answer to the question, "Does the sequence have some filled slot?"

This logic works because (a) if `valueToReturn` is `true`, the first loop stopped before the end, which can only be because the executor stopped at a slot with a CD in it; and (b) if `valueToReturn` is `false`, the executor must not have seen any CD to cause it to stop early. The design is implemented in the upper part of Listing 3.2, which defines a subclass of the Vic class. Figure 3.3 shows a sample execution of this method in detail.

Listing 3.2 The VicPlus class of objects

```
public class VicPlus extends Vic
{
   /** Tell whether any slot here or later has a CD. */

   public boolean hasSomeFilledSlot()
   {  String spot;                                 // design step 1
      spot = getPosition();
      while (seesSlot() && ! seesCD())             // design step 2
         moveOn();
      boolean valueToReturn;                       // design step 3
      valueToReturn = seesSlot();
      while ( ! spot.equals (getPosition()))       // design step 4
         backUp();
      return valueToReturn;                        // design step 5
   }  //======================


   /** Move to the last CD at or after the current position. But
    *  if there is no such CD, stay at the current position. */

   public void goToLastCD()
   {  String spot;                                 // design step 1
      spot = getPosition();
      while (seesSlot())                           // design step 2
      {  if (seesCD())
            spot = getPosition();
         moveOn();
      }
      while ( ! spot.equals (getPosition()))       // design step 3
         backUp();
   }  //======================
}
```

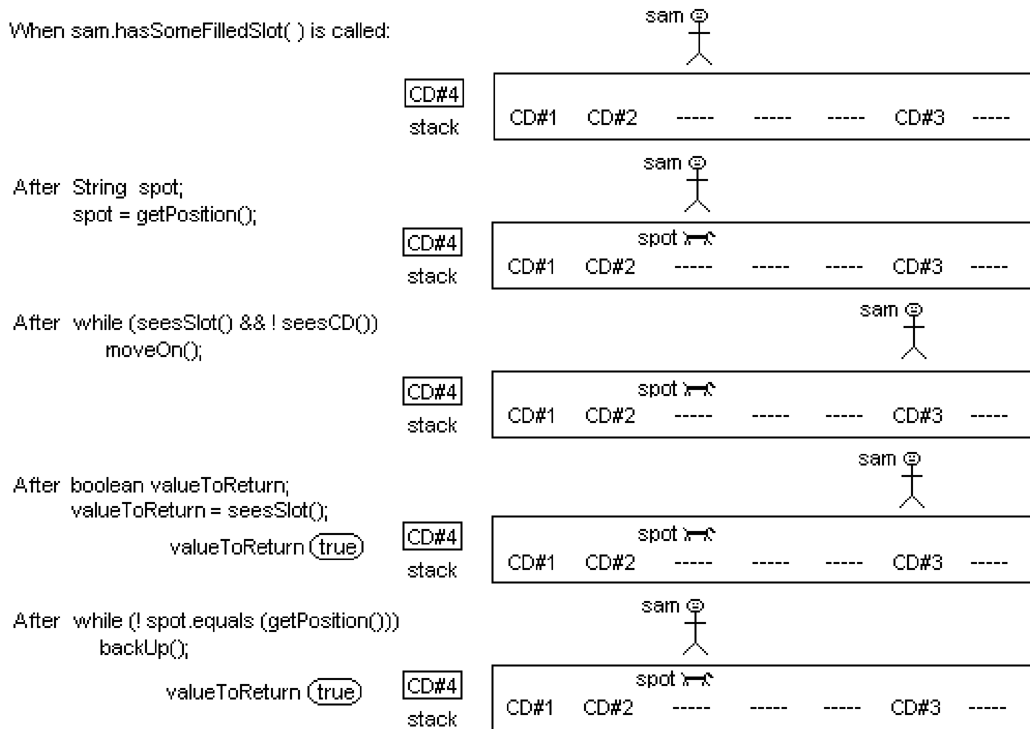


Figure 3.3 Execution of hasSomeFilledSlot

The goToLastCD method

The `goToLastCD` method advances the executor to the last slot that contains a CD. The logic to do so is a bit tricky. How do you go to the last CD? When you see a CD as you go down the sequence, you will not know whether it is the last one. A reasonable plan is to note its position and go further to see if there is another. If you do not see another, go back to the position marked. But if you do see another, forget about the earlier position and mark the later position instead. You need to see a description in ordinary English of how to do this, shown in the accompanying design block. The coding is in the lower part of Listing 3.2.

DESIGN of goToLastCD

1. Make a note of the current position in a variable; call it `spot`.
2. For each slot in the sequence from this position forward, do...
   If the slot you are at contains a CD, then...
   Change `spot` (the note) to indicate this new position instead.
3. Back up to the last position that was stored in `spot`.

Language elements

A Condition can be:	MethodName (Expression)
or:	VariableName . MethodName (Expression)

Exercise 3.6 How would you revise `goToLastCD` in Listing 3.2 so the executor advances to the last empty slot in its sequence?
Exercise 3.7 Explain why the assignment to `valueToReturn` in Listing 3.2 must not be replaced by `valueToReturn = seesCD()`.
Exercise 3.8 (harder) How would you revise `hasSomeFilledSlot` in Listing 3.2 so the executor tells whether at least two slots at or after the current slot are filled?

Exercise 3.9* Write a method `public void fillFirstEmptySlot()` for a subclass of Vic: The executor puts a CD in its first empty slot. Leave the position of the executor unchanged. Precondition: It has an empty slot and a CD is on the stack.
Exercise 3.10* Rewrite the `hasSomeFilledSlot` method in Listing 3.2 to not use any boolean variable.
Exercise 3.11** Write a method `public void fillLastEmptySlot()` for a subclass of Vic: The executor puts a CD in its last empty slot, but only if it has an empty slot and has a CD on the stack. Leave the position of the executor unchanged.

3.3 More On UML Diagrams

Suppose you want to have an application program that moves all the CDs in the first sequence of slots up to the front of the sequence. A reasonable plan is shown in the accompanying design block, assuming the stack is empty (an exercise shows how to do this when the stack is not known to be empty).

DESIGN of MoveToFront
1. Create a Vic for the first sequence of slots; name it `chun`.
2. Record its current position (the first slot) in a variable named `spot`.
3. Send `chun` down the sequence of slots, placing all CDs that it sees onto the stack.
4. Move `chun` back to the beginning of the sequence, to the position stored in `spot`.
5. Send `chun` down the sequence of slots, putting a CD in each slot, until `chun` gets to the end or `chun` runs out of CDs in the stack.

Each step of the design translates quite easily into just a few Java statements. A Java implementation of this design is in Listing 3.3 (see next page). Note that calls of `Vic.say` are inserted where appropriate, even though the design does not mention them. This illustrates the fact that people sometimes add to a design during implementation.

What classes do

The Vic class illustrates three <u>key functions of classes</u>, which you will see time and again:

1. <u>It defines the behaviors</u> an individual Vic object can have (e.g., `takeCD` and `moveOn`).
2. <u>It serves as a factory</u> for objects, since it allows the construction of new Vic objects (by calling on `new Vic()`).
3. <u>It defines class methods</u> that relate to Vic objects as a group, not to one individual Vic object (e.g., `say` , `stackHasCD`, and `reset`).

Going further with UML diagrams

Figure 3.4 (see next page) is the class diagram for this MoveToFront program. It illustrates the two remaining notations for creating UML class diagrams this book uses:

1. You may give the types of parameters in parentheses after the method name if you choose, e.g., the `say` and `equals` methods in Figure 3.4.
2. You may give the type of value a method returns if you choose (standard UML notation puts it after the parentheses, in contrast to Java method headings), e.g., the `getPosition` and `seesSlot` methods in Figure 3.4.

Listing 3.3 An application program using a Vic object

```
public class MoveToFront
{
   /** Move all the CDs in the first sequence of slots up to the
    *  front of the sequence. Precondition: stack is empty. */

   public static void main (String[ ] args)
   {  Vic chun;                                   // design step 1
      chun = new Vic();
      String spot;                                // design step 2
      spot = chun.getPosition();

      while (chun.seesSlot())                     // design step 3
      {  chun.takeCD();
         chun.moveOn();
      }
      Vic.say ("All CDs are now on the stack.");

      while ( ! spot.equals (chun.getPosition()))// design step 4
         chun.backUp();

      while (chun.seesSlot() && Vic.stackHasCD())// design step 5
      {  chun.putCD();
         chun.moveOn();
      }
      Vic.say ("The first few slots are now filled.");
   }  //=======================
}
```

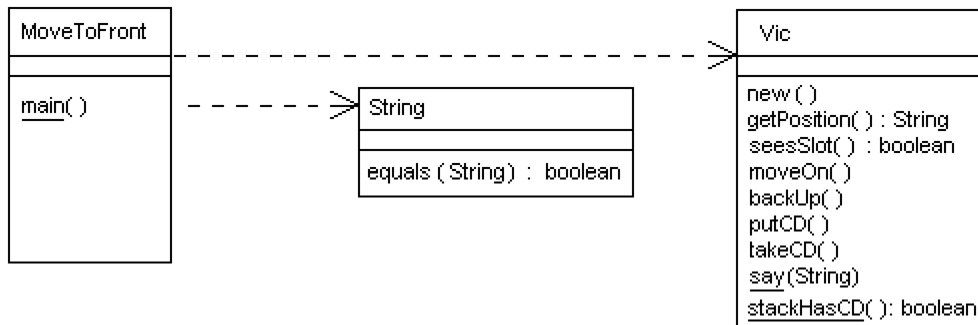


Figure 3.4 UML class diagram for MoveToFront

Exercise 3.12 The MoveToFront program has the CDs that were initially nearer the front of the sequence end up nearer the end of the sequence. Revise the program so that CDs that were initially nearer the front remain so. Hint: Take CDs while backing up.
Exercise 3.13 (harder) Write a query method `public boolean lastIsFilled()` for a subclass of Vic: The executor tells whether its last slot is filled. Precondition: `seesSlot()` is true.
Exercise 3.14* Write a Finder subclass of Vic with two methods: `goToFirstEmpty` and `goToFirstFilled`, where the executor advances until it comes to the first empty slot or the first filled slot, respectively, or to where `seesSlot()` is false if necessary.
Exercise 3.15** Add a method `public void byOnes()` to the Finder class of the preceding exercise: The executor moves one CD at a time to the earliest empty slot that comes before that CD in the sequence. This gets all the CDs to the front of the sequence without having more than one extra CD on the stack at a time. This `byOnes` method should call on the other two methods in Finder as needed.

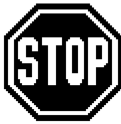
3.4 Using Private Methods And The Default Executor

The methods that go through an entire sequence, such as `fillSlots` and `toggleCDs` in Section 3.1, seem to call for a new subclass called Looper (since the methods will usually involve loops). We develop several such methods in this section and the next.

Initializing a variable when it is declared

Java allows you to combine the declaration of a variable with its initial assignment of a value. The following three statements illustrate how this language feature can be used. In each case, the one statement given replaces two statements in the specified listing:

```
boolean valueToReturn = seesSlot();   // in Listing 3.2
Vic chun = new Vic();                 // in Listing 3.3
String spot = chun.getPosition();     // in Listing 3.3
```



Caution A common error is to try to use the value of a variable before you put a value in it. For instance, you cannot say `sam.takeCD()` unless you previously assigned a value to sam, as in `sam = new Vic()`. You can avoid this error if you always assign a value to a variable in the same statement where you declare it or in the very next statement.

Listing 3.4 (see next page) illustrates the use of this new language feature in the Looper subclass of the Vic class. This class contains the `fillSlots` method discussed in Section 3.1 plus a `clearSlotsToStack` method with a very similar logic, since it does the exact opposite of `fillSlots`: It has the executor move all the CDs in its slots to the stack. Study both to make sure you thoroughly understand them.

Private methods

The third method in Listing 3.4, `backUpTo`, is for convenience; it saves writing out the loop that backs up, as seen in the two previous listings. You can call `backUpTo(x)` for any position value `x`; the value in `x` is assigned to `someSpot` inside the `backUpTo` method. So the loop in the `backUpTo` method executes until `getPosition()` returns a value that indicates the same spot `x` indicates.

The `backUpTo` method will be used only by other Looper methods (more Looper methods are in the next section). The restriction to Looper methods only is produced by having the `private` modifier in place of `public`. A **private** method cannot be mentioned outside of the class it is defined in. A method defined as `public` can be mentioned in any other class, as long as you supply the executor or other indication of the class it is in. These two modifiers indicate the "visibility" of the method.

You cannot call any of the methods in Listing 3.4 without an executor, i.e., an instance of the class before the dot. So all of these methods are instance methods of the Looper class.

Using the Looper methods

The new Looper methods in Listing 3.4 let you write the entire body of the earlier Listing 3.3 more simply as follows:

Listing 3.4 The Looper class of objects, some methods postponed

```
   /** Process a sequence of slots down to the end, usually
    *  without changing the current position of the executor. */

public class Looper extends Vic
{
   /** Fill in the current slot and all further slots
    *  from the stack until the end is reached. */

   public void fillSlots()
   {  String spot = getPosition();
      while (seesSlot())
      {  putCD();
         moveOn();
      }
      backUpTo (spot);
   }  //======================


   /** Move all CDs in the slots into the stack. */

   public void clearSlotsToStack()
   {  String spot = getPosition();
      while (seesSlot())
      {  takeCD();
         moveOn();
      }
      backUpTo (spot);
   }  //======================


   /** Back up to the specified position.  Precondition:
    *  someSpot records a slot at or before the current slot. */

   private void backUpTo (String someSpot)
   {  while ( ! someSpot.equals (getPosition()))
         backUp();
   }  //======================
}
```

```
   public static void main (String[ ] args)
   {  Looper chun = new Looper();
      chun.clearSlotsToStack();
      Vic.say ("All CDs are now on the stack.");
      chun.fillSlots();
      Vic.say ("The first few slots are now filled.");
   }  //=======================
```

Garbage collection

Inside each of the two public method definitions in Listing 3.4, the String object that `getPosition()` returns is assigned to a String variable declared inside the method. Such a variable is temporary, transient. The variable is created when the method is called and it is discarded when the method is exited, by coming to the end of the commands in the method.

When the method says spot = getPosition(), the newly-created object has only spot to refer to it. When the method is exited, no variable at all refers to the String object. Whenever that happens, the runtime system automatically disposes of the object so it does not clutter up RAM. This is **garbage collection**. Programs written in languages without garbage collection can leave RAM littered with unusable space after they terminate execution; this is called memory leakage.

A metaphor may help to explain garbage collection: An object is a boat. An object variable is a metal ring on a river dock to which you can tie a boat. A boat can be tied to more than one ring at the same time. A statement such as sam = sue has the boat that is tied up to sue also tie up to sam. Whatever boat may have been already tied up to sam is cast off from sam, since only one boat can tie up to a ring at a time (the ring is not big enough for two ropes). If any boat becomes untied from all rings, it floats down the river, goes over a waterfall, and smashes into kindling at the bottom. The garbage is then collected by Java's automatic garbage collectors and recycled to make new boats.

The default executor: this

Within the definition of an instance method, you cannot refer to the executor by name, since it varies depending on the method call. One time you might have the statement sam.fillSlots(), and another time you might have sue.fillSlots(). So inside the definition of fillSlots, you cannot mention either sam or sue, because it could be either of them or some other Looper variable altogether.

Java provides a pronoun for the executor: this always refers to the executor when used in a statement inside an instance method. For instance, the body of the fillSlots method of Listing 3.4 could be rewritten as follows with the same effect:

```
public void fillSlots()  // illustrating use of this
{  String spot = this.getPosition();
   while (this.seesSlot())
   {  this.putCD();
      this.moveOn();
   }
   this.backUpTo (spot);
}  //======================
```

If your main logic executes the statement sam.fillSlots(), then for that execution of the logic of fillSlots, this refers to sam. If your main logic later executes the statement sue.fillSlots(), then for that second execution of the logic of fillSlots, this refers to sue. The rule the compiler applies is: If you do not explicitly state the executor where an executor is required, it supplies the **default executor** this.

More Looper methods

Listing 3.5 (see next page) contains the definition of two more Looper methods, with three methods left as exercises. The this pronoun appears in Listing 3.5 wherever an instance method is called, to help you remember what it means. But in the future, this book only uses the optional this when other objects are mentioned in the method; in such a case, this helps you keep straight which object you are talking about.

The fillOddSlots method in the upper part of Listing 3.5 fills in every other slot starting with the first one. It does not seem to need a detailed design because you can just make a small change in the logic of fillSlots as follows: After that logic moves on by one slot, insert an if-statement to check that there really is a slot there and, if so, move on by one extra slot. Figure 3.5 gives an example of what happens when fillOddSlots is called.

Listing 3.5 More methods of the Looper class

```
// public class Looper extends Vic, continued

   /** Fill in every other slot from the stack, beginning
    *  with the current slot, until the end. */

   public void fillOddSlots()
   {  String spot = this.getPosition();
      while (this.seesSlot() && stackHasCD())
      {  this.putCD();
         this.moveOn();
         if (this.seesSlot())
            this.moveOn();
      }
      this.backUpTo (spot);
   }  //======================


   /** Tell whether every slot here and later has a CD. */

   public boolean seesAllFilled()
   {  String spot = this.getPosition();           // design step 1
      while (this.seesSlot() && this.seesCD())    // design step 2
         this.moveOn();
      boolean valueToReturn = ! this.seesSlot(); // design step 3
      this.backUpTo (spot);                       // design step 4
      return valueToReturn;                       // design step 5
   }  //======================


   // the following three are left as exercises
   public void fillEvenSlots()      {  }
   public boolean seesOddsFilled()  {  }
   public boolean seesEvensFilled() {  }
```

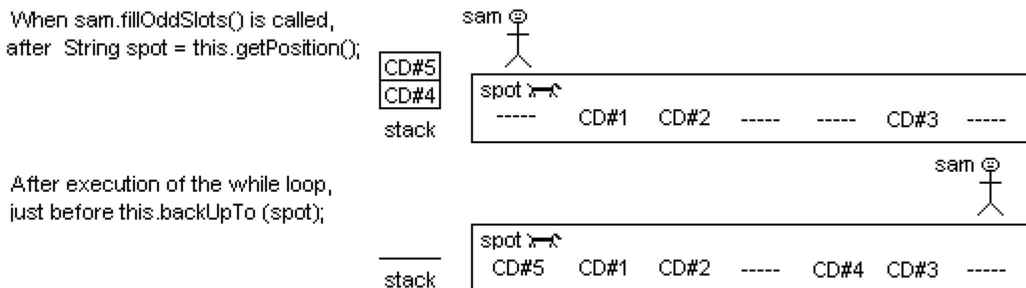


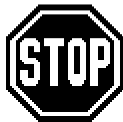
Figure 3.5 Stages of execution for fillOddSlots

Development of seesAllFilled

If `seesAllFilled` is called when the position of the Vic is already at the end of its sequence of slots, we still say it is true that all slots are filled, in the **vacuous** sense that there is no unfilled slot as a counterexample. This is an application of the computer science meaning of an assertion of the form All-A-are-B; it may not coincide with the vernacular meaning.

The `seesAllFilled` method has the executor tell whether all remaining slots, starting from the current position, contain CDs. A reasonable plan is in the accompanying design block. The coding for `seesAllFilled` is in the lower part of Listing 3.5.

DESIGN of seesAllFilled
1. Mark the current position so you can return to it when you have the answer to the question.
2. Go down the sequence until you get to the end or else you see an empty slot.
3. Determine the value to be returned, which is `false` if you are now at an empty slot and is `true` otherwise.
4. Go back to the position you had at the beginning of execution of the process.
5. Return the value found at step 3 of this logic.



Caution A common error people make in Java is to put a semicolon right after the parentheses around a condition, as in `if(whatever);` or `while(whatever);`. That semicolon marks the end of the `if` or `while` statement, so the compiler does not consider the statement on the next line to be subordinate. A bare semicolon directly after the parentheses around the `while` or `if` condition counts as a subordinate statement that does nothing. That is rarely what you want, so avoid doing that.

Language elements
A Statement can be: Type VariableName = Expression ;
You may use "private" in place of "public" in a method heading.
You may use "this" within an instance method to explicitly indicate the executor of the method.

Exercise 3.16 Explain why the following method heading causes a compilation error:
`public static void Main (string[] args)`.

Exercise 3.17 Explain why the following coding causes a compilation error:

```
   Looper Bob = new Looper();
   Vic spot = Bob.getPosition();
   while ( ! spot.equals (Bob.getPosition())
      Bob.moveOn();
```

Exercise 3.18 Rewrite the `hasSomeFilledSlot` method in the earlier Listing 3.2 to use `this` wherever it is allowed.

Exercise 3.19 Write the `public void fillEvenSlots()` method described in the Looper class. The first slot filled should be the slot after the current position (if it exists). Call on `fillOddSlots` to do most of the work.

Exercise 3.20 (harder) Write the `seesOddsFilled` method described in the Looper class.

Exercise 3.21 (harder) Write the `seesEvensFilled` method described in the Looper class. Call on `seesOddsFilled` to do most of the work.

Exercise 3.22* Write an application program that tests out your solutions to the two preceding exercises by calling each one for the first sequence and printing a message saying what each returned.

Exercise 3.23* Write a Looper method `public void bringBack()`: The executor removes the CD in its current slot, if any, then brings each CD that is later in the sequence back one slot. Leave the position of the executor unchanged.

Exercise 3.24* Compare the coding of `seesAllFilled` in Listing 3.5 with the coding of `hasSomeFilledSlot` in the earlier Listing 3.2. Note that the only material difference is the presence or absence of the not-operator in two places. If both places had the not-operator, what would be a good name and comment heading for the resulting method? What would they be if neither place had the not-operator?

Exercise 3.25** Write a Looper method `public void overOrOut()`: The executor moves each CD in its sequence of slots (a) to the following slot if the following slot exists and is empty, or (b) to the stack if not. Leave the position of the executor unchanged.

3.5 A First Look At Declaring Method Parameters

When one of the public methods in the earlier Listing 3.4 executes the statement `backUpTo(spot)`, it assigns the value in `spot` to the `someSpot` variable in the `backUpTo` method. That is, it executes `someSpot = spot`, so that `someSpot` now refers to the same position `spot` refers to. Then any test of `someSpot`'s object inside the method is by definition a test of the object `spot` refers to.

A value inside the parentheses of a method call is an **actual parameter** or **argument** of the call. For instance, the actual parameter of `backUpTo(spot)` is `spot`, and the actual parameter of `Vic.say("whatever")` is `"whatever"`.

A variable inside the parentheses of a method heading is a **formal parameter** of the method. For instance, `someSpot` is the formal parameter of the method called by `backUpTo(spot)`, as defined in Listing 3.4

The hasAsManySlotsAs method

The method call `sue.hasAsManySlotsAs(ruth)` is to tell whether the sequence represented by `sue` has exactly the same number of slots as the sequence represented by `ruth` has, assuming that `sue` and `ruth` are declared as Vic variables. The logic can be designed as shown in the accompanying design block.

DESIGN of hasAsManySlotsAs, with a parameter
1. Make a note of the current position of the executor; store it in `thisSpot`.
2. Repeat the following until either the executor or the parameter has no more slots...
    2a. Move both the executor and the parameter forward one slot.
3. Make a note that the value to be returned by this method is `true` only if both of the two Vics now have no more slots.
4. Back up both of them one at a time until the executor gets back to `thisSpot`.
5. Return the value noted in Step 3.

Listing 3.6 (see next page) contains an implementation of this design. Suggestion: When you need to write a method whose logic is not immediately obvious, first make a design similar to the ones you have seen so far in this chapter.

Correspondence of formal and actual parameters

Suppose the statement `sue.hasAsManySlotsAs(ruth)` is in a main method. Then it causes an execution of the boolean method that assigns the value in `ruth` to `par` and the value in `sue` to `this`. So whatever `this` does is actually being done by `sue` and whatever `par` does is actually being done by `ruth`. `par` is the formal parameter that corresponds to the actual parameter `ruth`. Note: `par` is short for parameter; this book uses this name when the context suggests no better name for a parameter.

Listing 3.6 An instance method to compare the lengths of two sequences

```
public class TwoVicUser extends Vic
{
   /** Tell whether the executor has exactly the same number of
    *  slots as the Vic parameter.  */

   public boolean hasAsManySlotsAs (Vic par)
   {  String thisSpot = this.getPosition();       // design step 1
      while (this.seesSlot() && par.seesSlot())   // design step 2
      {  this.moveOn();
         par.moveOn();
      }

      boolean valueToReturn = ! this.seesSlot()   // design step 3
                           && ! par.seesSlot();

      while ( ! thisSpot.equals (this.getPosition()))// d. step 4
      {  this.backUp();
         par.backUp();
      }
      return valueToReturn;                       // design step 5
   }  //======================
}
```

If the main method also contains `bill.hasAsManySlotsAs(ted)`, it causes an execution of the boolean method that gives `par` the value in `ted` and `this` the value in `bill`. Note: `bill` must be a TwoVicUser object, because it is the executor of a method defined in the TwoVicUser class. But `ted` can be any Vic object, such as a Vic or a TwoVicUser object. That is, you may assign to a Vic variable (such as `par`) any object of any subclass of Vic. Note also that it is legal to have a statement run over two or more lines; a semicolon marks the end of a statement, not an end-of-line.

Review of the email metaphor

In the email metaphor of Section 1.6, a method call is an email message you send to an object. The method name is the subject line of the email. The parameter is the body text of the email. A method call with an empty pair of parentheses indicates there is no body text in the message. But when the email message is `bill.hasAsManySlotsAs(ted)`, the recipient `bill` sees from the subject line that you want to know whether it has as many slots as some other object. Then `bill` looks at the body text of the email to find out who the other object is.

The giveEverythingTo method

The action method in Listing 3.7 (see next page) removes all CDs from the executor's slots and puts them into the Looper parameter's slots, along with any CDs that are already on the stack. `sam.giveEverythingTo(sue)` is a sample call. The logic is quite straightforward because both the executor and the parameter are Loopers: Tell the executor to `clearSlotsToStack`, then tell the Looper parameter to `fillSlots`. Since the Giver class is a subclass of Looper which is a subclass of Vic, a Giver object inherits all the capabilities of Loopers as well as Vics. Figure 3.6 shows the **hierarchy** of object classes involving the Giver class.

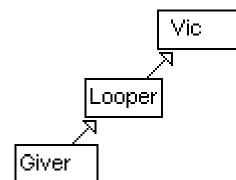


Figure 3.6 Giver's hierarchy

Listing 3.7 The Giver class of objects

```
public class Giver extends Looper
{
   /** The executor gives all of its CDs to the Looper parameter,
    *  which distributes them to its own slots to the extent
    *  possible, along with any CDs originally on the stack. */

   public void giveEverythingTo (Looper target)
   {  this.clearSlotsToStack();
      target.fillSlots();
   }  //======================
}
```

Programming Style The `giveEverythingTo` method uses the fact that the two Vics are in fact Looper objects, by calling the `clearSlotsToStack` method and the `fillSlots` method defined in the earlier Listing 3.4. The executor must of course be an instance of Giver, because the `giveEverythingTo` method is defined in the Giver class. However, the parameter only needs to be able to execute `fillSlots`, so it is enough that it be an instance of Looper. It is good style to not specify it to be an instance of Giver. That retains the greatest flexibility in the use of the `giveEverythingTo` method.

Local variables versus parameters

You cannot call the `giveEverythingTo` method unless you have a Giver object to do the giving and a Looper object to be given to. The value that is listed before the dot in that method call is assigned to `this` inside the `giveEverythingTo` method. The value that is listed inside the parentheses of that method call is assigned to `target` inside the `giveEverythingTo` method.

Say a main method declares four different Giver objects and contains the following two statements using them:

```
steve.giveEverythingTo (don);
mike.giveEverythingTo (dru);
```

The main method has `steve`, `don`, `mike`, and `dru` as its **local variables** (variables declared within the body of the method). In general, the only way you can refer to the value of a local variable of one method within another method is to have the local variable be an executor or an actual parameter of the method call. Either way, that other method cannot change which value is stored in the local variable; it can only change the state of the object to which the value refers.

On the first call of the `giveEverythingTo` method, `steve` is the executor, so `this` is an **alias** for `steve` during execution of the first call. On the second call of the `giveEverythingTo` method, `mike` is the executor, so `this` is an alias for `mike` during execution of the second call.

Likewise, `target` is the formal parameter, so `target` is an alias for the actual parameter `don` on the first call but it is an alias for the actual parameter `dru` on the second call. In effect, the first method call performs the assignment `target = don` and the second method call performs the assignment `target = dru`.

A formal parameter (declared inside the parentheses of a method heading) differs from a local variable (declared in the body of the method) in that (a) a formal parameter receives its initial value at the time the method is called, but (b) a local variable has no initial value until the statements explicitly give it one. Both kinds of variables are local to the method definition in the sense that they cannot be used outside the method definition.

Multiple parameters

A method can have two or more parameters, separated by commas. For instance, the following method has two Vic parameters. The executor tells whether at least two of the three sequences (its own, `one`'s, and `two`'s) have at least one slot available:

```
public boolean atLeastTwoNotAtEnd (Vic one, Vic two)
{  if (this.seesSlot() && one.seesSlot())
      return true;
   if (this.seesSlot() && two.seesSlot())
      return true;
   return one.seesSlot() && two.seesSlot();
}  //======================
```

Two examples of how this method might be called are:

```
if (jazz.atLeastTwoNotAtEnd (pop, classical))...
boolean result = first.atLeastTwoNotAtEnd (third, second);
```

Language elements
You may put the following within the parentheses of a method heading: Type VariableName
If you have two or more such phrases within the parentheses, separate those phrases with commas.

Exercise 3.26 If you changed the first statement of `hasAsManySlotsAs` to be `String thisSpot = par.getPosition()`, what other changes would you have to make so it gives the right answer?
Exercise 3.27 How would you change the `hasAsManySlotsAs` method so the executor tells whether it has more slots than the Vic parameter?
Exercise 3.28 Write a query method `public boolean isAtOneGivenPosition (String one, String two)` for a subclass of Vic: The executor tells whether either of those two parameters is the same as its current position.
Exercise 3.29 (harder) Write a method `public void moveToCorrespondingSlot (Vic par)` for a subclass of Vic: Every CD in a slot of the parameter that corresponds to an empty slot in the executor is moved over to the executor's corresponding slot. Leave the position of the two Vics unchanged.
Exercise 3.30* Write a query method `public boolean hasMoreThanDouble (Vic par)` for Looper: The executor tells whether it has more than twice as many slots as the Vic parameter. Do not use numeric variables. Precondition: The executor is known to have an even number of slots. Extra credit: Remove the precondition.
Exercise 3.31* Draw the UML diagram for Listing 3.6.
Exercise 3.32* Write a query method `public boolean matches (Vic par)` for Looper: The executor tells whether it has a CD wherever the parameter has a CD and it does not have a CD wherever the parameter does not. That is, the two sequences of slots are the same in terms of the presence of CDs, starting from the current slot in each.
Exercise 3.33** Write a method `public void shiftOne (Vic one, Vic two)` for a subclass of Vic: At each position where the executor has an empty slot and either of the two Vic parameters has a filled slot in the corresponding position, shift the CD to the executor's slot. When a choice is possible, take a CD from the first parameter's slot.
Exercise 3.34** Write a query method `public boolean sameNumber (Vic par)` for Looper: The executor tells whether it has the same number of CDs in its slots as the Vic parameter. Do not use numeric variables. Hint: Advance each to the next non-empty slot. Repeat this until one runs out of slots. Does the other?

3.6 Returning Object Values

You have written several methods that return boolean values. It is also legal to have a method return an object value, such as a Vic or Looper or String value. For instance, `sam.getPosition()` returns a String object. In a method heading, you put the **return type** (the type of value returned by the method) immediately before the method name.

The `lastEmptySlot` method in Listing 3.8 illustrates the return of a String object. Its purpose is to return the position of the last empty slot in a sequence of slots; but it returns the current position, empty or not, if there is no empty slot after the current position. A main method could use this `lastEmptySlot` method in a statement such as

```
String spot = sam.lastEmptySlot();
```

or in a condition such as

```
sue.lastEmptySlot().equals (sue.getPosition())
```

which tells whether `sue` is positioned at its last empty slot. This method must be in the Looper class because it calls the private method `backUpTo`, which is not even accessible from a subclass of Looper.

Listing 3.8 A Looper method returning a String

```
/** Return the position for the last empty spot in
 *  the sequence, or the current spot if no empty spot. */

public String lastEmptySlot()
{  String spot = this.getPosition();
   String lastEmpty = spot;  // in case no later slot is empty
   while (this.seesSlot())
   {  if ( ! this.seesCD())
         lastEmpty = this.getPosition();
      this.moveOn();
   }
   this.backUpTo (spot);
   return lastEmpty;
}  //======================
```

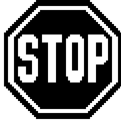
The logic in this `lastEmptySlot` method goes through each slot in the sequence, setting `lastEmpty` to the position of each empty slot it sees. `lastEmpty` could be given several different values, but each assignment replaces whatever was already stored in the variable. So only the last value assigned is in `lastEmpty` when the loop terminates. At that time, `lastEmpty` must contain the position of the last empty slot.

No design block is given for this method because it is so similar to the `goToLastCD` method in the earlier Listing 3.2. You will find it informative to compare and contrast the two step by step.

Returning a Vic object

You can return a Vic object as well as a String object, as illustrated by the following method. It repeatedly creates new Vic objects until it finds one whose first slot contains a CD (or until it runs out of Vics).

```
public Vic firstWithCD()
{  Vic sequence = new Vic();
   while (sequence.seesSlot() && ! sequence.seesCD())
      sequence = new Vic();
   return sequence;
}  //======================
```



Caution You can avoid the most common compiler errors that beginners make if you just check two things before compiling a program: First, every left brace has a matching right brace directly below it, and vice versa. Second, no line followed by an indented line ends in a semicolon, and every line not followed by an indented line does end in a semicolon.

Reminder Variable names should start with lowercase letters and class names should start with capitals. You may ask why it should be so. You might as well ask why you should not say "el mano" in Spanish instead of "la mano." You would be understood alright, but it is not proper Spanish.

Exercise 3.35 What change would you have to make in the `lastEmptySlot` method of Listing 3.8 to return the position of the last non-empty slot?
Exercise 3.36 (harder) Write a Looper method `public Vic shorterOne (Vic par)`: Return the Vic with the fewer slots, either the executor or the parameter. Leave both unchanged. Precondition: They do not have an equal number of slots.
Exercise 3.37* Revise the `lastEmptySlot` method to return the position of the next-to- last empty slot. Return the initial position if the executor has less than two empty slots.
Exercise 3.38* Rewrite the `lastEmptySlot` method to have the executor go directly to the end of the sequence and then find the last empty slot as it comes back towards the starting position.

Part B Enrichment And Reinforcement

3.7 More On The Analysis And Design Paradigm

Problem Statement Write a program to work with three sequences of slots for storing CDs. The first Vic has perhaps some country music CDs in its slots, and the second Vic has perhaps some jazz CDs in its slots. You are to put all of these CDs in the third Vic's slots, alternating the two kinds of music (for variety). The third Vic may already have some CDs in its slots; you are to leave these where they are and fill in the rest of the slots, to the extent possible.

Many people, given a problem assignment such as this, are not sure where to start. It is extremely useful to break up the process into five basic stages:

Analysis (clarifying what to do) Make sure you clearly understand what the program is to accomplish. Consider exceptional cases and how you are to handle them. Write down data you will use to test the final software and figure out what will happen when that data is used. Go to the client (or your instructor if appropriate) for a decision on ambiguous points. You need a clear, complete, unambiguous specification before you can go further.

Logic Design (deciding how to do it) Make a step-by-step plan of how you will get the job done. The design method described later in this section is a reliable and efficient way to do this. You have already seen many design blocks illustrating the method.

Object Design (choosing the objects that help you do it) See what kinds of objects you have already available that can provide the services you need. Perhaps you have to add more capabilities to existing objects (e.g., additional Looper methods). Perhaps you need to invent completely new kinds of objects to do the job.

Refinement (making sure you are doing it) Go over your logic at length to make sure it satisfies the stated requirements, that it will do what it should for the given test data, and that it will behave correctly in exceptional cases.

Implementation (doing it) Translate your logic design into Java using the methods supplied by the objects you have designed.

When you implement the design, you will usually find that several steps are too complex to do easily. In that case you call a new method for which you repeat the entire process on a lower level: (a) analyze the specification for the sub-problem to make sure it is clear; (b) design a step-by-step solution of the sub-problem; (c) select or invent the object methods you need; (d) refine the plan; (e) implement the plan in Java.

To create a good plan, write out or say aloud the steps the computer will take, in ordinary English sentences. Then organize this list of steps to show which steps are done conditionally or repeatedly and to highlight the condition for doing them or repeating them. Otherwise keep it in English (or whatever natural language you speak most fluently). You can sharpen your plan by planning the data with which you will test your program when it is done and computing what the results will be for the various test runs.

Analysis for the Interleaf program

When you think further about the problem statement for the alternating CDs, you realize it is not quite clear whether the first CD moved is to come from the first sequence or the second sequence. Also, if the first or second sequence of slots has more CDs than are required to fill in the slots in the third sequence, are those extra CDs supposed to stay where they are, or should they go onto the stack?

You talk to the client to get the answers to these questions. For the rest of this discussion, assume that the client says all CDs from the first sequence are to go into the odd-numbered slots (1, 3, 5, etc.) of the third sequence, with any leftovers to be put on the stack. The client tells you there will be enough CDs in the first sequence to do this. However, if all of the odd-numbered slots of the third sequence are already filled, you are to leave the CDs in the first sequence. You are to handle the second sequence analogously, with CDs going into the even-numbered slots. The client is quite clear that the CDs put into the third sequence are to be in the same order as they were in the sequence they came from.

Logic design for the Interleaf program

A reasonable logic design of the problem is shown in the accompanying design block. This design illustrates **Structured Natural Language Design**, **SNL design** for short. Steps 3b and 4b specify that the target's slots are to be filled in reverse order from the stack, so that whatever was furthest down the source sequence, and thus ended up on top of the stack after transferring the CDs to the stack, goes furthest down the target sequence.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic

1. Create two Vic objects to serve as the source of the CDs.
2. Create a third Vic object to receive the CDs. Refer to it as `target`.
3. If `target` does not have all of its odd-numbered slots filled, then...
    - 3a. Transfer every CD the first Vic has in its slots to the stack.
    - 3b. Fill `target`'s odd-numbered slots from the stack in reverse order.
4. If `target` does not have all of its even-numbered slots filled, then...
    - 4a. Transfer every CD the second Vic has in its slots to the stack.
    - 4b. Fill `target`'s even-numbered slots from the stack in reverse order.

You saw several examples of structured design earlier in this chapter and in Chapter Two. The three ways that SNL design differs from ordinary discourse are as follows:

1. When action X is executed conditionally, express it in the form `if whatever then...X` with the action X on a separate line and indented beyond the description of the condition.
2. When action X is executed repetitively, express it in something like the form `For each value do...X` with action X on a separate line and indented beyond the description of the looping. The exact phrasing is unimportant; `Repeat until whatever...X` often makes more sense in a particular situation.
3. When you must refer to a particular value several times, give it a name (`target` in the preceding example). This is clearer than using a phrase such as "the third Vic that was created" many times.

This design is an **algorithm**, which means a step-by-step description of a process for accomplishing a task, specific enough that at each step there is no question what to do next. You do not have to put the secondary line numbers in your design if you do not want to. The crucial part is to show which actions depend on which conditions.

Everything about this design is ordinary English except for variable names and indenting to show which actions are done under which conditions. That is the "structured" part of the design. Do not write in Java until you know what you are going to say.

Do not try to break the design down into very many small steps; ten steps is usually more than enough. But include all significant steps. Your steps can specify quite complex actions, such as Steps 3a and 3b in the preceding design.

Object design for the Interleaf program

Now you decide what kinds of objects will help you get the job done quickly and easily. Checking whether odd-numbered or even-numbered slots are filled, and clearing out all the CDs from a sequence, are skills possessed by Looper objects (Listing 3.4 and Listing 3.5). So you choose them to help you, rather than the poorly-educated Vic objects.

Each step of the logic design turns out to be easy to implement in Java (using e.g. a Looper's `clearSlotsToStack` for Step 3a) except for Steps 3b and 4b. An object that can do Step 3b can do Step 4b with a small adjustment. So you really need an even smarter kind of Looper, one that can carry out the process for Step 3b. You could add a method to the Looper class for this task. But you probably will never need it again, and the Looper class is becoming rather cluttered. So you could make a subclass of Looper that has this capability, intended for use in this program only.

You need to develop an SNL design for this second-level process. It could be as shown in the accompanying design block. You then refine your overall design by studying it to make sure you understand every aspect of it and by studying the original specifications to make sure you met them all.

DESIGN of the sub-algorithm: filling odd-numbered slots in reverse order

1. Make a note of the current position in the sequence.
2. Move two slots at a time down the sequence until you reach the end.
3. If you moved an odd number of times to get to the end then...
    - 3a. Back up to the last slot and put a CD there.
4. Repeat the following for every other slot until you are where you started...
    - 4a. Back up two slots.
    - 4b. Put a CD in the current slot.

Implementation of the Interleaf program

The implementation stage translates each sentence of the design into a few statements of the programming language. You often make some minor additions while coding. For instance, you could start the program with Vic's `reset` command, which lets you run several test cases easily. And you could display a message when the program finishes. Listing 3.9 is a possible final solution for the coding. Figure 3.7 is the UML diagram.

Listing 3.9 Application program using the BackLooper class of objects

```
public class Interleaf
{
   /** Move the first sequence's CDs to the odd-numbered slots
    *  of the third sequence.  Move the second sequence's CDs to
    *  its even-numbered slots.  No effect if not 3 sequences. */

   public static void main (String[ ] args)
   {  Vic.reset (args);
      Looper one = new Looper();                    // design step 1
      Looper two = new Looper();
      BackLooper target = new BackLooper();         // design step 2
      if ( ! target.seesOddsFilled())               // design step 3
      {  one.clearSlotsToStack();                   // design step 3a
         target.fillInReverse();                    // design step 3b
      }
      if ( ! target.seesEvensFilled())              // design step 4
      {  two.clearSlotsToStack();                   // design step 4a
         target.moveOn();                           // design step 4b
         target.fillInReverse();
      }
      Vic.say ("All done putting CDs in #3");
   }  //======================
}
//################################################################


public class BackLooper extends Looper
{
   /** Fill slots 0,2,4,6,... ahead of the current one, reverse
    *  order. Precondition: The executor has at least 1 slot. */

   public void fillInReverse()
   {  String spot = getPosition();              // sub-design step 1
      boolean movedInPairs = true;              // sub-design step 2
      while (seesSlot())
      {  movedInPairs = ! movedInPairs;
         moveOn();
      }
      if ( ! movedInPairs)                      // sub-design step 3
      {  backUp();
         putCD();
      }
      while ( ! spot.equals (getPosition())) // sub-design step 4
      {  backUp();
         backUp();
         putCD();
      }
   }  //======================
}
```

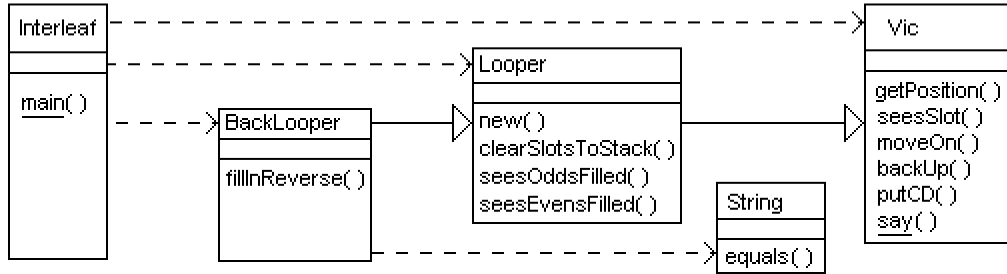


Figure 3.7 UML class diagram for the Interleaf class

The statement `movedInPairs = ! movedInPairs` illustrates a technique you have not seen before. The statement switches the value of the boolean variable between being `true` and being `false` each time through the loop. So it will be `true` at the test of `seesSlot()` if and only if the loop has executed an even number of times. If it is `false` when the loop terminates, the executor must back up one slot to be an even number of slots away from the slot where it started.

Technical Note Java will let you put the BackLooper class in the same file with the application program class if you remove the word `public` from the class heading for BackLooper. The compiler will "complain" if you try to use a non-public class in another class in some other file. But since you do not expect to use the BackLooper class for any other situation, keeping it in the same file with Interleaf is a reasonable thing to do.

Other aspects of software development

In larger projects, you should normally set an intermediate goal of developing software that does much of what the final project should do. After you test it thoroughly, you add to it to come closer to the final version. Repeat this until done. This is called **iterative development**. You will see examples of it later in this book.

Most people cannot go directly from the statement of a complex problem to the expression of the algorithm in Java with few errors. It is far easier, and takes much less time overall, to go through the intermediate stages just described.

If you recite your solution aloud in ordinary English sentences, you will more easily hear any bugs it might have. That will make less work for you in getting the final Java solution right.

Programming Style It is good style to rarely comment individual statements in your programs; commenting each method as a whole is usually enough. This book comments some individual statements in Chapters One through Three only to help you understand what newly-introduced commands and concepts mean and to see how steps of the design are implemented in the coding.

Exercise 3.39 Rewrite the first loop in the `fillInReverse` method so that it advances two slots each time through the loop, except if it can only advance one slot it sets `movedInPairs` to `false`.
Exercise 3.40* Write out a design in SNL for the program of the following exercise.
Exercise 3.41** Write an application program that moves a CD from each slot in the first sequence to the corresponding slot in the second sequence where possible, and also from each slot in the second to the corresponding slot in the first where possible.

3.8 Analysis And Design Example: Finding Adjacent Values

Suppose you need a special kind of Looper object that can answer the question, "Do you have two CDs right next to each other?" You only want it to consider CDs at or after its position at the time you ask the question. So you need a subclass of Looper with a query method that tells whether the executor contains two CDs right next to each other, looking only at slots at or after the current position. Two examples of how such a method might be used are as follows:

```
if (sue.hasTwoTogether())...
while (sam.hasTwoTogether())...
```

You could use the accompanying design block for this `hasTwoTogether` method.

DESIGN of hasTwoTogether
1. Make a note of the current position so you can return to it when you have the answer to the question.
2. Go down the sequence and find out whether you have two CDs together.
3. Go back to the position you had at the beginning of execution of the process.
4. Return the value found at step 2 of this logic.

This plan does not have enough detail. Steps 1, 3 and 4 can be implemented with just one or two Java statements, but step 2 is quite complex. You need a sub-plan to break step 2 down into enough detail that you can easily implement it in Java.

One tactic for solving this sub-problem is to make a note at each slot of whether a CD is in the slot. When you come to the next slot, you know to return true if it has a CD and your note says the previous slot has a CD. Otherwise you update the note for the current slot and go further.

This logic is hard to follow written in normal paragraph form. You need to lay it all out in a structured design so you can study it. The accompanying design block works well.

DESIGN of the sub-algorithm foundPair
1. Create a boolean variable `previousSlotIsEmpty` that can be tested at any slot to tell whether the previous slot is empty. Since you will first test it when at the second slot, initialize it to `true` if the first slot is empty, to `false` if not.
2. Move forward to the second slot.
3. For each slot in the sequence, from this second slot forward, do...
   - 3a. If you do not see a CD then...
     - Make a note that `previousSlotIsEmpty` is `true`, to be tested later.
   - 3b. But if you do see a CD and the previous slot was empty then...
     - Make a note that `previousSlotIsEmpty` is `false`, to be tested later.
   - 3c. Otherwise you see a CD and the previous slot was not empty, so...
     - Return the answer `true` without going any further in this logic.
   - 3d. Move forward to the next slot.
4. Return `false`, since you reached the end of the sequence without seeing two together.

Always review your design for logical consistency and completeness before you implement it. A review of this plan finds a defect: The program will fail if you try to move forward to the next slot (step 2 of the sub-algorithm) when you are already at the end of the sequence. So the plan should be corrected to guard against that possibility. The implementation in Java in Listing 3.10 makes this correction with a crash-guard: an extra check of `seesSlot()` avoids calling the private method when there is no slot there.

Listing 3.10 The PairFinder class of objects

```
public class PairFinder extends Vic
{
   /** Tell whether there are two CDs in a row at any point at
    *  or after this position.  Leave the executor unchanged. */

   public boolean hasTwoTogether()
   {  String spot = getPosition();               // design step 1
      boolean hasTwoTogether = seesSlot()        // design step 2
                            && foundPair();
      while ( ! spot.equals (getPosition()))     // design step 3
         backUp();
      return hasTwoTogether;                     // design step 4
   }  //======================

   private boolean foundPair()
   {  boolean previousSlotIsEmpty = ! seesCD(); // design step 1
      moveOn();                                 // design step 2
      while (seesSlot())                        // design step 3
      {  if ( ! seesCD())                       // design step 3a
            previousSlotIsEmpty = true;
         else // has one in this slot           // design step 3b
         {  if (previousSlotIsEmpty)
               previousSlotIsEmpty = false;
            else                                // design step 3c
               return true;
         }
         moveOn();                              // design step 3d
      }
      return false;                             // design step 4
   }  //======================
}
```

This logic uses the boolean variable `previousSlotIsEmpty` in a way you have not seen before. The purpose of such a variable is to store information obtained during one iteration of the loop to be used during the next iteration of the loop. It is best to name such a variable to convey its meaning at the time it is tested, not at the time it is assigned a value.

Is it a bad thing that the `foundPair` method changes the object and is thus not a true query method? No, it does not count as a style violation because (a) a call of `hasTwoTogether` does not, and (b) no one outside the class can call `foundPair`, since it is a private method.

Sequential/selection/repetition

The key activity in creating software is designing and implementing methods. Specifically, you design and implement a main method which calls on other methods which, unless you have them in your library, you must also design and implement. Some of those methods in turn can call on other methods which you must then design and implement or else find in your library of existing methods, and so forth.

Whatever the objects your software uses, whether Vics or Turtles or something else, the design of a method comes down to repeatedly choosing one of three kinds of activities, as follows. The last two kinds of activities listed are usually done with an if-statement or a while-statement, respectively:

- which sequence of actions you execute, or
- which query you test to determine which of two sequences of actions you execute, or
- which query you test to determine how many times you execute one sequence of actions.

Even when you start using numbers in your programs, you will find that the calculations or the tests for inequality you perform are all done only as part of actions or queries to be used as described in the preceding list. The list of activities can be summarized as: sequential, selection, and repetition.

In short, an essential part of programming is putting together actions and queries using `if` and `while` to create a method that performs a single well-defined task. Even though most of the programming you have seen has been in the highly limited context of Vics and Turtles, the skills and concepts you have learned are highly useful in most programming situations.

Loop control values

You must check out any looping logic you write to make sure it eventually terminates. The best way to do this is to make sure it has a **loop control value**. That is a numeric expression that (a) must be positive for the loop to continue executing, but (b) decrements by at least 1 each time the loop executes.

For most of the loops you have seen, the loop control value is the number of slots left in the sequence from the current position, since (a) the continuation condition usually tests `seesSlot()` to make sure it is true, and (b) each iteration of the loop executes `moveOn()`. That is, the number of slots left must be positive and each iteration subtracts 1 from the number of slots left.

For some loops you have seen, the loop control value is the number of CDs on the stack, since (a) the continuation condition usually tests `stackHasCD()` to make sure it is true, and (b) each iteration of the loop executes `putCD()`. That is, the number of CDs on the stack must be positive and each iteration subtract 1 from the number of CDs on the stack.

For some other loops you have seen, the loop control value is the number of slots between a previous position and the current position, since (a) the continuation condition tests `! spot.equals (getPosition())`, and (b) each iteration of the loop executes `backUp()`.

Exercise 3.42 Revise the `foundPair` method to have only one return statement. Hint: Declare a boolean variable `found` before starting the loop and put `return found;` in place of the last statement.
Exercise 3.43** Revise the `foundPair` method to not use a boolean variable. Instead, when the executor sees a CD, have it go forward to see if there is one after it and, if not, move back again. Then discuss whether this is a better solution than the one in Listing 3.10. Can you think of a better solution than either?

*3.9 Turing Machines (*Enrichment)*

The Vic machine described in these two chapters is a modification of a Turing Machine. A Turing Machine is an extremely simplified version of a computer, one that is highly impractical for actual use. The advantage of this is that it is far easier to develop logical proofs about what is and is not computable by a computer if the computer has maximal simplicity.

The **Church-Turing Thesis** is that, for any computational process that can be programmed on any computer, some Turing Machine program carries out exactly the same process. This thesis is generally accepted by computer scientists. So when you see a proof in a Theory of Computation course that a certain problem cannot be solved by a Turing Machine program, that is accepted as a proof that no computer program will ever exist that can solve that problem.

A Turing Machine works with a sequence of positions (like Vic slots). The sequence is called a **tape**. Each position contains a single digit or else a blank. The machine begins operation at the far left of the sequence of positions. It is not allowed to back up past the position it starts on; the tape begins at that position. However, the tape goes on as far as necessary to the right (so there is no need for anything resembling `seesSlot()`).

A **Turing Machine** can check what digit is at its current position, if any; it can write a blank or any digit at the current position; and it can go forwards and backwards on the tape. To help you understand exactly what a Turing Machine is, we describe a class of objects similar to Vic. We could call it Tum for short (from TUringMachine). A Tum object understands only four basic commands:

- `sees(0)` tells whether there is a digit 0 at the current position, and similarly for other digits 1 through 9. The `sees(-1)` message tells whether there is a blank at the current position.
- `put(0)` puts the digit 0 at the current position, and similarly for other digits. Any negative value, as in `put(-1)` or `put(-30)`, puts a blank at the current position. The new value replaces whatever value was already at that position.
- `moveOn()` goes one position further right, away from the beginning of the tape.
- `backUp()` goes one position further left, towards the beginning of the tape. It crashes the program if the current position is the one at the tape's beginning.

You also have a strong restriction on how you can put these basic commands together to create new methods for subclasses of Tum:

- Each method is to have no parameters, no local variables, and no return value. So the only way to pass information around or store it is to put it on the tape.
- Each method body for a subclass of Tum is to consist of (a) at most one while-statement, followed by (b) at most one multi-way selection statement. No two conditions are to be true for the same digit. The subordinate statements in either case are simple method calls, selected from the four basic commands and other methods in a subclass of Tum.

An example of a permissible subclass of Tum is shown in Listing 3.11 (see next page). The reason for the restrictions is that whatever you write can then be easily translated to a hypothetical machine code that has only one kind of instruction, structured as follows:

> If the current method is X and the current position contains Y then...
> > Put Z in that position (or leave it unchanged if you wish).
> > Move 1 position forward or backward (or remain there if you wish).
> > Switch to some method (or not, as you wish).

Listing 3.11 A subclass of Tum

```
public class SampleTum extends Tum
{
   public void clear()
   {  while (sees (0) || sees (1))
      {  put (-1);
         moveOn();
      }
      if (sees (-1))
         backUp();
   }  //======================

   public void switch()
   {  if (sees (0))
      {  put (1);
         backUp();
      }
      else if (sees (1))
      {  put (0);
         backUp();
         clear();
      }
   }  //======================
}
```

Each method you could be in represents a different **state** of the Turing Machine. Since a program can only have a finite number of methods, this is a **finite-state machine**.

For this implementation in Java, you cannot write on the physical tape before the machine begins its operation or read the tape after it finishes. So you need a way to initialize the tape for the Tum object and a way to display the current status of the tape. This can be done using statements such as the following:

```
Tum sam = new Tum ("104 52");
sam.carryOutSomeProcess();
sam.showStatus (4);
```

The creation of a new Tum makes the tape consist of the given String of characters followed by many blanks. And the `showStatus` command displays the tape on the screen (plus the numeric parameter, which helps you figure out which call of `showStatus` produced which output). You will learn how to fully implement the Tum class as described here by the middle of Chapter Five. It is a major programming project to do this, so the Tum class is not provided here.

For a Turing Machine that works with binary numbers, you would only allow input consisting of 1s and 0s and blanks. You could then develop, for instance, a subclass of Tum that could add two such binary numbers together and leave the result on the tape for the `showStatus` message to display.

*3.10 Javadoc Tags (*Enrichment)*

As you learned in Chapter Two, a comment that begins with `/**` and ends with `*/` and comes immediately before a public class, public method, or public variable is special (you will see public variables in Chapter Five). When you give the command

```
javadoc SomeClass.java
```

in the terminal window, the javadoc formatting tool creates a webpage named `SomeClass.html` which displays those comments (documentation for the class) in a useful form. The first complete sentence in each such comment is put in a summary section, so you want to make sure it conveys the key idea of your comment. Multi-line comments can have each line after the first begin with an asterisk if you like.

The javadoc tool creates several more html files for you. One is `index.html`, which lists in one section all of the methods in your class in alphabetical order with clear descriptions. It also lists any variables you have in another section. Another is `index-all.html`, which gives an alphabetical index of all the parts of your class. Browse one of these files and click on the Tree and Help options to see other documentation.

The twelve javadoc tags

You can put `@return` in a comment to tell the reader that the phrase that follows describes the value that is returned by a method. The javadoc tool will display it in a special way, because `@return` is one of the standard javadoc **tags**. The following are three tags that can be used in javadoc comments for classes, methods or variables:
`@see` lists other classes or methods that are highly related to this one.
`@since` tells which version of the software first had this feature.
`@deprecated` means it is outdated and should not be used anymore.

A class can have the following two tags:
`@author` tells the author of the coding.
`@version` tells the current version of the software.

Methods can have the following four tags:
`@param` describes a parameter of the method.
`@return` describes the value that the method returns, if it returns one.
`@throws` names the kind of Exception thrown and under what conditions. You will learn about Exceptions shortly; they almost never arise when programming with Vics.
`@exception` is the older form of `@throws`.

The three remaining permissible tags `@serial`, `@serialField`, and `@serialData` are ones for which you will not have any use for a long time.

3.11 Review Of Chapter Three

Listing 3.4 and Listing 3.7 illustrate almost all Java language features introduced in this chapter.

About the Java language:

- A while-statement states a **continuation condition** followed by the **subordinate statements**. The continuation condition must be true in order for the subordinate statements to be executed. Those statements are placed within matching braces unless there is only one subordinate statement. An **iteration** is one execution of the subordinate statements in this **loop**.
- `someString.equals(anotherString)` is a method in the **String** class in the Sun standard library. This method tests whether the two String values have the same content, i.e., the same characters in the same order.
- A method declared as `private` can only be called from within the class where it is defined. A method declared as `public` can be called from any class.

- You can use `this` inside an instance method as a reference to the executor of the method call. If you call an instance method without an executor, the compiler uses the **default executor**, which is `this` of the method containing the method call (i.e., it is `this` instance of the class). This applies when the method call is itself inside an instance method, not a class method such as `main`.
- When a method heading has a variable declaration in its parentheses, each call of the method must have a value of the same type in its parentheses. When the method executes, this **formal parameter** is initialized to the value given in the method call (the **actual parameter**, also known as the **argument**).
- You can declare additional variables within the body of a method, e.g., booleans, Vics, and Strings. These **local variables** have no connection with variables outside of the method, and they have no initial value. You can only use a local variable after the point where it is declared and inside whatever braces contain the declaration.
- See Figure 3.8 for the remaining new language features. In that grammar summary, the `Type` could be a ClassName or boolean; an `ArgumentList` is a number of expressions separated by commas; and a `ParameterList` is a number of `Type VariableName` combinations separated by commas.

`while (Condition)` `   Statement`	**statement** that repeats test-Condition-do-Statement, quitting when the Condition is false

`while (Condition)`
`{   StatementGroup`
`}`
	statement with the same effect as described above, except the entire sequence of 0 or more statements is executed between tests.

`Type VariableName = Expression;`	**statement** that combines declaring and defining a variable
`ClassName.MethodName (ArgumentList)`	**expression** that calls a no-executor-method in the class
`VariableName.MethodName(ArgumentList)` `MethodName (ArgumentList)`	**expressions** that call an instance method with parameters

`public Type MethodName(ParameterList)`
`        { StatementGroup }`
`public void MethodName(ParameterList)`
`        { StatementGroup }`
	declarations of instance methods that accept input initially assigned to the formal parameters

Figure 3.8 Declarations, expressions, and statements added in Chapter Three

Other vocabulary to remember:

- When an object your program has created has no variable that refers to it, then the object is recycled by Java's **garbage collection** mechanism.
- The **hierarchy** of some classes is the set of relationships between subclasses and superclasses of that group of classes.
- **Structured Natural Language Design** expresses the logic of an **algorithm** completely in English or some other natural language, except that (a) statements that are executed conditionally (depending on whether some condition is true) are indented relative to the condition, and (b) some variable names are used.

About Vic methods (developed for this book):

- `someVic.getPosition()` returns a String that describes the current position in the sequence represented by `someVic`. If `x` and `y` are two such Strings returned when at the same position in the same sequence, then `x` and `y` may be different String objects, but it will be true that `x.equals(y)`.
- All other Vic methods were described in Chapter Two: four action instance methods (`moveOn`, `backUp`, `putCD`, `takeCD`), two query instance methods (`seesSlot`, `seesCD`), three class methods (`reset`, `say`, `stackHasCD`), and `new Vic()`.

About UML notation (all class diagram notations used in this book):

- A **class box** is a rectangle 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.
- 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.
- Class methods and class variables are to be underlined.
- You may add the parameter types in the parentheses after a method name.
- You may add the return type after those parentheses, with a colon in between.

Answers to Selected Exercises

3.1
```
public void removeAllCDs()
{    while (seesSlot())
     {    takeCD();  // reminder:  This does no harm if there is no CD in the slot
          moveOn();
     }
}
```
3.2
```
public void toLastSlot()
{    while (seesSlot())
          moveOn();
     backUp();  // because the loop went one step PAST the last slot
}
```
3.3
```
public void takeOneBefore()
{    backUp();
     while ( ! seesCD())
          backUp();
     takeCD();
}
```
3.6 Put an exclamation mark in front of seesCD().

3.7 If there is no slot there, the program fails. And if seesSlot() is true, so is seesCD().

3.8 You could insert the following lines after the first while-statement:
```
if (seesSlot())
{    moveOn();
     while (seesSlot() && ! seesCD())
          moveOn();
}
```
3.12 Remove the chun.takeCD() method call from the first while statement, remove the first Vic.say statement, and replace the second while-statement by the following:
```
while ( ! spot.equals (chun.getPosition()))
{    chun.backUp();
     chun.takeCD();
}
```
3.13
```
public boolean lastIsFilled()
{    String spot = getPosition();
     while (seesSlot())
          moveOn();
     backUp();
     boolean valueToReturn = seesCD();
     while ( ! spot.equals (getPosition())
          backUp();
     return valueToReturn;
}
```
3.16 "string" must be capitalized, but "Main" should not be capitalized.

3.17 You cannot assign a String value to a Vic variable, so change "Vic" to "String". It is bad style to capitalize the name of a variable such as Bob, but it is not a compilation error.

3.18
```
public boolean hasSomeFilledSlot()
{    String spot = this.getPosition();
     while (this.seesSlot() && ! this.seesCD())
          this.moveOn();
     boolean valueToReturn = this.seesSlot();
     while ( ! spot.equals (this.getPosition()))
          this.backUp();
     return valueToReturn;
}
```

3.19
```
public void fillEvenSlots()
{	if (seesSlot())
	{	moveOn();
		fillOddSlots();
		backUp();
	}
}
```
3.20
```
public boolean seesOddsFilled()
{	String spot = getPosition();
	while (seesSlot())
	{	if ( ! seesCD())
		{	backUpTo (spot);
			return false;
		}
		moveOn();
		if (seesSlot())
			moveOn();
	}
	backUpTo (spot);
	return true;
}
```
3.21
```
public boolean seesEvensFilled()
{	if ( ! seesSlot())
		return true;   // vacuously true, since there are no slots to be empty
	moveOn();
	boolean valueToReturn = seesOddsFilled();
	backUp();
	return valueToReturn;
}
```
3.26 Change the second while-condition to ! thisSpot.equals (par.getPosition())

3.27 Change the middle statement to the following:
```
boolean valueToReturn = this.seesSlot();
```
3.28
```
public boolean isAtOneGivenPosition (String one, String two)
{	return one.equals (this.getPosition()) || two.equals (this.getPosition());
}
```
3.29
```
public void moveToCorrespondingSlot (Vic par)
{	String thisSpot = this.getPosition();
	while (this.seesSlot() && par.seesSlot())
	{	if (par.seesCD() && ! this.seesCD())
		{	par.takeCD();
			this.putCD();
		}
		this.moveOn();
		par.moveOn();
	}
	while ( ! thisSpot.equals (this.getPosition()))
	{	this.backUp();
		par.backUp();
	}
}
```
3.35 Remove the exclamation mark from the if-condition.

3.36 Modify the hasAsManySlotsAs method in Listing 3.6 as follows:
Replace "boolean" by "Vic" in the method heading.
Replace the statement between the two while-loops by the following:
```
Vic valueToReturn;
if (this.seesSlot())
	valueToReturn = par;
else
	valueToReturn = this;
```
3.39
```
while (seesSlot())
{	moveOn();
	if (this.seesSlot())
		this.moveOn();
	else
		movedInPairs = false;
}
```
3.42 Replace "while (seesSlot())" by the following two lines:
```
boolean found = false;
while (seesSlot() && ! found)
```
Replace "return true" within the loop by "found = true".
Replace "return false" at the end by "return found".