Chapter 5 More Decision Making

Chapter Objectives

After studying this chapter, you should be able to:

- Follow a process for constructing `while` loops with fewer errors
- Avoid common errors encountered with `while` loops
- Use temporary variables to remember information within methods
- Nest statements inside other statements
- Manipulate Boolean expressions
- Perform an action a predetermined number of times using a `for` statement
- Write `if` and `while` statements with appropriate style

The `if` and `while` statements studied in Chapter 4 form the basis for making decisions in programs. Any program you care to write can be written with using only the `if` and `while` statements to change the flow of control. This chapter continues the discussion with variations of the `if` statement and other ways to repeatedly execute statements that can simplify our code even though they are not strictly required. It explores a process for constructing `while` statements and points out errors to avoid. In short, this chapter summarizes the accumulated wisdom of programming with `if` and `while` statements.

Sometimes decisions are made based on what has happened in the past. Such decisions are facilitated by temporary variables that can remember information for later use in the same method.

5.1 Constructing `while` Loops

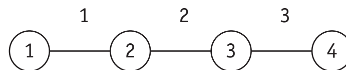
In Chapter 4, we learned how to affect the sequence in which program statements are executed with the `if` and `while` statements. In this section, we will examine some common errors and a more rigorous process for constructing `while` loops that can help you avoid those errors.

5.1.1 Avoiding Common Errors

The `while` statement provides a powerful programming tool. By using it wisely, we can solve complex problems. However, the sharper the ax, the deeper it can cut, as they say. With the power of the `while` statement comes the potential for making some significant mistakes. This section will examine several typical errors beginning programmers make when using the `while` statement. If we are aware of these errors, we have a better chance of avoiding them and an easier time identifying them for correction when they do occur.

The Fence-Post Problem

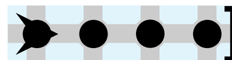
If we want to build three fence sections, how many fence posts will we need? The obvious answer is three, but that is wrong. Look at Figure 5-1. The figure should help us to understand why the correct answer is four.



(figure 5-1)

Fence-post problem

We can encounter the fence-post problem when using the `while` loop. For example, consider the problem of clearing all the `Things` between a robot and a wall. The robot's starting intersection also contains a `Thing`, as shown in Figure 5-2.



(figure 5-2)

Initial situation for an example of the fence-post problem

It might seem that a natural way to solve this problem is with the following method:

```
public void clearThingsToWall()
{ while (this.frontIsClear())
  { this.pickThing();
    this.move();
  }
}
```

If we trace the method's execution carefully, we discover that the loop finishes and the robot does not crash into the wall. However, the easternmost Thing is not picked up, as shown in Figure 5-3.

(figure 5-3)

After executing clearThingsToWall

In this example, the Things are the fence posts and the moves are the fence sections. The while loop executes the same number of pickThing and move statements. Consequently, one Thing will be left when the loop finishes. We can handle this situation by adding an extra pickThing command after the while loop finishes executing, as shown in the following code fragment:

PATTERN

Loop-and-a-Half

```
public void clearThingsToWall()
{ while (this.frontIsClear())
  { this.pickThing();
    this.move();
  }
  this.pickThing();
}
```

LOOKING AHEAD

The while-true *loop is an elegant solution to the loop-and-a-half problem. See Section 5.5.3.*

It is surprising how often the fence-post problem occurs in computer science. It is also known as the **loop-and-a-half** problem because, in one sense, the loop executes an extra half iteration when the last thing is picked.

Infinite Loops

You may have experienced a computer program that **hangs**. It appears to be running fine and then mysteriously fails to respond to your commands. The entire program appears "frozen." Such a program is probably caught in an **infinite loop**. An infinite loop is one that has no way of ending because the programmer has forgotten to include a statement (or sequence of statements) whose execution allows the loop's test to become false. Here is an example:

```
while (this.isFacingNorth())
{ this.pickThing();
  this.move();
}
```

KEY IDEA

Every loop must have a statement that can affect the test.

Nothing within this loop will change the robot's direction. As a result, the loop will iterate zero times if the robot is initially facing any direction other than north. Unfortunately, if it is facing north, we condemn the robot to walk forever (unless, of course, it breaks because there is no Thing to pick up or it runs into a wall; it will also

stop if the computer itself crashes or the computer's power supply is disrupted).[1] We must be very careful when we plan the body of the `while` loop to avoid the possibility of an infinite loop.

5.1.2 A Four-Step Process for Constructing `while` Loops

The common errors discussed in the previous section, plus difficulties you probably experienced constructing loops in the previous chapter, should motivate you to study a formal process to construct `while` loops. The goal is to structure our thinking so that our loops are more likely to be written correctly.

There are four steps. We first outline them briefly and then illustrate them with two examples.

Step 1: Identify the actions that must be repeated to solve the problem.

Step 2: Identify the Boolean expression that must be true when the `while` statement has completed executing. Negate it.

Step 3: Assemble the `while` loop with the actions from Step 1 as the body and the Boolean expression from Step 2 as the test.

Step 4: Add additional actions before or after the loop to complete the solution.

We will now apply this four-step process to two examples. The first is the `clearThingsToWall` problem discussed in Section 5.1.1. The second is a more complicated problem.

Applying the Four-Step Process to Clearing Things

Consider again the problem of clearing all the `Things` between (and including) the robot's intersection and a `Wall`, as shown in Figure 5-2. We don't know how far away the `Wall` is.

Step 1 is to identify the actions that must be repeated. One way to do this is to solve a small example of the problem without a loop. The four `Things` in Figure 5-2 already qualifies as a small problem (much smaller than, say, 400!). To solve it, we need to perform the following actions:

[1] Of course, if we have overridden `pickThing` or `move`, then anything is possible. One of the new versions could change the direction, and then the robot would exit the `while` loop.

pick a thing
move
pick a thing
move
pick a thing
move
pick a thing

Clearly, two actions are repeated, *pick a thing* and *move*. In particular, note that the two actions appear in groups—as shown by the brackets on the right—and that one of the actions doesn't appear in any of the groups. Because of that extra action, there are actually two ways to group the repeated actions. The other grouping results in an extra *pick a thing* at the beginning of the sequence.

Step 2 is to identify the Boolean expression that must be true when the loop finishes. The robot should stop collecting things when it is blocked by a wall; that is, the loop should stop when the test `this.frontIsBlocked` is true. But the test for a `while` statement isn't whether the loop should stop; the test is whether the loop should continue. Therefore, the test to use is the negation of `this.frontIsBlocked()`: `!this.frontIsBlocked()` or `this.frontIsClear()`.

Step 3 assembles the `while` loop using a group of repeated actions from Step 1 and the Boolean expression from Step 2. This yields the following code:

```
while (this.frontIsClear())
{ this.pickThing();
  this.move();
}
```

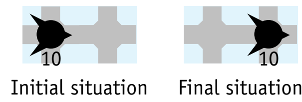
Loop-and-a-Half

Finally, Step 4 cleans things up. Recall, for example, that there was one action in Step 1 that wasn't included in any of the groups. This is the extra fence post from the loop-and-a-half problem. The extra action was at the end of the preceding sequence and so it is placed after the loop. The final solution is as follows:

```
while (this.frontIsClear())
{ this.pickThing();
  this.move();
}
this.pickThing();
```

Applying the Four-Step Process to Shifting `Things`

A more difficult looping problem is a robot shifting a pile of `Things` from one intersection to the next, as shown in Figure 5-4.



(figure 5-4)

Shifting a pile of `Thing` *objects to the next intersection*

There are four requirements:

- The pile will always have at least one `Thing`.
- The robot can only move one `Thing` at a time.
- The robot must finish on the intersection to which it has moved the pile.
- The robot must not move unnecessarily. Specifically, it must not go back to the original intersection when that intersection is empty.

This problem clearly requires actions to be performed zero or more times rather than once or not at all. Therefore a `while` loop is required, and we can apply the four-step process.

KEY IDEA

Identify the repeating steps with a loopless solution.

Step 1 is to identify the steps that must be repeated. As before, we assume a typical initial situation and solve the problem without a loop. In this case, assume the pile has four `Things` on it. (Remember, our final solution must handle a pile of any size. We are assuming four things only while we find the steps that repeat.) To move all four `Things` to the next intersection, the robot must perform the following steps:

pick up one thing
shift it to the next intersection
go back to the original intersection
pick up one thing
shift it to the next intersection
go back to the original intersection
pick up one thing
shift it to the next intersection
go back to the original intersection
pick up one thing
shift it to the next intersection

This sequence of actions has three actions that must be repeated: *pick up one thing*, *shift it to the next intersection*, and *go back to the original intersection*. As before, we group them with brackets, as shown in the preceding pseudocode. With two actions left over in the sequence, however, determining what comes before and after the loop will be trickier than in the previous example.

Step 2 identifies the test that must be true when the loop has finished executing. We can consider several possible tests:

- *the robot is on the next intersection*—This can't be the correct test to end the loop because the robot is on the next intersection many times while moving the `Things`.

- *the original intersection has no things on it*—This is the test that must be true when the loop is finished executing. If it's not true, the task obviously isn't finished. If it is true, we might have a little cleanup to do, but the repetitious work is over.

In Java, this test can be expressed with the Boolean expression `!this.canPickThing()`. That is, the loop stops when the robot can't pick up any more `Things` from the original intersection. The test that determines when the `while` loop should continue is the negation of this, or `this.canPickThing()`.

Step 3 assembles the loop. The test was identified in Step 2, giving us the following structure:

```
while (this.canPickThing())
{ ...
}
```

The question is how to arrange the repeated action inside the loop. There are actually three possibilities, as follows. In each case, Step 4 is anticipated and the leftover actions are placed either before or after the loop, depending on how they appear in the loopless solution from Step 1.

```
while (this.canP…)
{ pick up one thing
  shift it
  go back
}
pick up one thing
shift it
```

```
pick up one thing
while (this.canP…)
{ shift it
  go back
  pick up one thing
}
shift it
```

```
pick up one thing
shift it
while (this.canP…)
{ go back
  pick up one thing
  shift it
}
```

Barring a flash of insight, the way to choose one of these options is to trace them. An excellent situation to use for your first trace is the smallest possible problem: one `Thing` on the original intersection. Try tracing the left-most loop for yourself. You should convince yourself that it fails for two reasons. First, it tries to pick a `Thing` from an empty intersection. Second, the problem specification says it can't return to the intersection after it is empty, which it does.

The right-most loop also has problems. On any sized problem it picks up a `Thing` and shifts it to the next intersection. But then it determines if it can pick up the `Thing` it, just shifted. It's performing the test on the wrong intersection.

The middle loop executes correctly. It picks up a `Thing` from the intersection and then asks if there is another `Thing` for the *next* trip. When the pile has just one `Thing` in it, there is nothing left for another trip, and so it skips the loop body and shifts the `Thing` it just picked up to the next intersection.

This is *not* an obvious solution. It takes a deep insight to realize that the test for picking up a `Thing` should be performed *after* picking one up and not before. There is no

algorithm for solving such a problem, but the four-step process provides significant guidance in finding a solution.

Reasoning about the `while` Statement

We just demonstrated that, at least in pseudocode, our solution works for one particular instance of the problem. But what about all the other sizes of piles? We cannot test all possible initial situations (there are infinitely many of them), but we can test several and do a reasonable job of convincing ourselves that the solution is correct.

One method of informally reasoning about the statement has two steps. First, we must show that the statement works correctly when the initial situation results in the `while` statement's test being `false`. That is, in fact, the situation we just traced where the pile contained only one `Thing`. It picked that `Thing` up and then performed the test, which evaluated to `false`.

KEY IDEA

Trace a situation where the loop body does not execute.

Second, we must show that each time the loop body is executed, the new situation is a smaller but similar version of the old situation. By smaller, we mean there is less work to do before finishing the loop. By similar, we mean that the situation has not radically changed while executing the loop body. (In this example, a non-similar change could be that the robot is facing a different direction.)

KEY IDEA

Demonstrate that executing the loop body results in a smaller but similar version of the problem.

By tracing a few iterations of the loop, we see that after each iteration, the size of the pile decreases by one. This gives us confidence that the situation will eventually reach the case where the `while` loop becomes `false`, which we already checked for correctness.

5.2 Temporary Variables

The `Robot` class has a method to count the number of `Things` in its backpack, but there is no corresponding method to count the number of `Things` on an intersection. One can imagine, for example, a `Robot` that is following a trail left behind by another `Robot`. Finding zero `Things` on the intersection means to go forward, finding one `Thing` means to go left, and finding two `Things` means to go right. With the knowledge we have now, such a problem would be very difficult to solve.

A **temporary variable**, also called a **local variable**, is the core of one solution. A temporary variable stores a value for later use within the same method. We have already used temporary variables, starting with Chapter 1. `sanDiego` and `karel` are both temporary variables in the following code fragment:

KEY IDEA

A temporary variable stores a value for later use in the method.

```
City sanDiego = new City();
Robot karel = new Robot(sanDiego, 1, 2, Direction.EAST);
...
karel.move();
```

Both of them are storing a value, a reference to an object, for later use within the same method (`main`).

To count the number of `Things` on an intersection, we'll use a temporary variable, but one storing an integer rather than a reference to a `Robot` or `City`.

A temporary variable used to count something would typically be declared like this:

```
int counter = 0;
```

PATTERN
Temporary Variable

Like the `City` and `Robot` declarations shown earlier, this declaration has a type and a name followed by its initial value. The type is `int` and the name is `counter`. The type of `int` specifies that this variable will only store a particular kind of value, integers. The **initial value** is zero, the first value assigned to the variable.

We have already worked with one kind of integer variable when we used parameters in Section 4.6.1. In that case, we decremented the parameter `howFar` with the statement `howFar = howFar - 1`. Similarly, `counter` can be decremented with the statement `counter = counter - 1`. It's not surprising that `counter = counter + 1` increments the value in `counter` by one.

5.2.1 Counting the `Things` on an Intersection

With this background, let's count the `Things` on an intersection with the idea that a robot may be used to follow a trail, as described earlier. In pseudocode, we use the following:

```
count the number of things here
if (number of things here == 0)
{ move
}
if (number of things here == 1)
{ turn left
}
if (number of things here == 2)
{ turn right
}
```

We begin by declaring a temporary variable to use in determining the number of `Things` on the intersection. We can also update the pseudocode to use it in appropriate places, as follows:

PATTERN
Temporary Variable

```
int numThingsHere = 0;

update numThingsHere with the number of things on this intersection

if (numThingsHere == 0)
{ this.move();
}
```

```
if (numThingsHere == 1)
{ this.turnLeft();
}
if (numThingsHere == 2)
{ this.turnRight();
}
```

We can now focus on the remaining pseudocode to update `numThingsHere`. Our strategy will be to pick up all of the `Things` on the intersection, increasing `numThingsHere` by one each time a thing is picked up. There may be many things, so a `while` loop is appropriate. In terms of the four-step process for writing a loop, the actions to repeat (Step 1) are picking a `Thing` and incrementing the variable. The test for stopping (Step 2) is when there is nothing left on the intersection. Therefore, the loop should continue while `canPickThing` returns `true`. Assembling the loop (Step 3) yields the following code:

```
while (this.canPickThing())
{ this.pickThing();
  numThingsHere = numThingsHere + 1;
}
```

For this problem, there is nothing to do before or after the loop (Step 4).

The completed code fragment for counting the number of `Things` on an intersection and turning in the appropriate direction is shown in Listing 5-1.

Listing 5-1: *A code fragment to count the number of* `Things` *on an intersection and move appropriately*

```
1  int numThingsHere = 0;
2  while (this.canPickThing())
3  { this.pickThing();
4    numThingsHere = numThingsHere + 1;
5  }
6
7  if (numThingsHere == 0)
8  { this.move();
9  }
10 if (numThingsHere == 1)
11 { this.turnLeft();
12 }
13 if (numThingsHere == 2)
14 { this.turnRight();
15 }
```

Counting

5.2.2 Tracing with a Temporary Variable

We can increase our understanding of the code in Listing 5-1 by tracing it. Doing so will also increase our confidence in its correctness. As usual, we shall employ a table to record the statements executed and the state of the program. To adequately trace the state for this fragment, we need to record the `Robot`'s street, avenue, and direction; the value of `numThingsHere`; and the number of `Things` on the intersection. It is also useful to record the results of the tests performed by the `if` and `while` statements. A single column will do for both tests, and we will only record the result in the line where it is executed. We don't need to record the number of `Things` in the robot's backpack because that information is not used in the code fragment.

(table 5-1)

Tracing the execution of the code fragment in Listing 5-1

Table 5-1 traces the situation in which the `Robot` is facing north on (3, 5). That intersection has two `Thing` objects. The code should cause the `Robot` to turn right to face east—which it does.

Program Statement	test	(str, ave)	Direction	numThingsHere	Number on Intersection
		(3, 5)	north	???	2
1 `int numThingsHere = 0;`					
		(3, 5)	north	0	2
2 `while (this.canPickThing())`					
	true	(3, 5)	north	0	2
3 `{ this.pickThing();`					
		(3, 5)	north	0	1
4 `numThingsHere = numThingsHere + 1;`					
		(3, 5)	north	1	1
2 `while (this.canPickThing())`					
	true	(3, 5)	north	1	1
3 `{ this.pickThing();`					
		(3, 5)	north	1	0
4 `numThingsHere = numThingsHere + 1;`					
		(3, 5)	north	2	0
2 `while (this.canPickThing())`					
	false	(3, 5)	north	2	0

Program Statement	test	(str, ave)	Direction	numThingsHere	Number on Intersection
`7 if (numThingsHere == 0)`					
	false	(3, 5)	north	2	0
`10 if (numThingsHere == 1)`					
	false	(3, 5)	north	2	0
`13 if (numThingsHere == 2)`					
	true	(3, 5)	north	2	0
`14 { this.turnRight();`					
		(3, 5)	east	2	0

(table 5-1) ***continued***

Tracing the execution of the code fragment in Listing 5-1

5.2.3 Storing the Result of a Query

In Listing 5-1 we turned a `Robot` based on how many `Things` it found on the intersection. We could perform a similar task based on how many `Things` are in its backpack. One way to do this is shown in the following code fragment:

```
if (this.countThingsInBackpack() == 0)
{ this.move();
}
if (this.countThingsInBackpack() == 1)
{ this.turnLeft();
}
if (this.countThingsInBackpack() == 2)
{ this.turnRight();
}
```

Suppose you had your own backpack and performed this same task. You probably wouldn't count the number of things in the backpack three times—you would count them once and then remember the answer long enough to decide whether to turn or move. Using a temporary variable, a `Robot` can do the same thing. Instead of assigning a value of `0` to the temporary variable when we declare it, we will assign whatever value `countThingsInBackpack` returns, as shown in the following fragment:

```
1 int numThings = this.countThingsInBackpack();
2 if (numThings == 0)
3 { this.move();
4 }
5 if (numThings == 1)
6 { this.turnLeft();
7 }
```

```
 8  if (numThings == 2)
 9  { this.turnRight();
10  }
```

Suppose that the robot has one thing in its backpack. Then `countThingsInBackpack` will return the value 1. The variable `numThings` will refer to that value until it is changed or the method ends. In line 2, the value that `numThings` refers to (1) is compared to 0. They are different, and so line 3 is not executed. In line 5, the value that `numThings` refers to (1) is again compared, this time to the value 1. They are equal, and so the robot turns left. In line 8, `numThings` is again compared, but the values are not equal and so the turn in line 9 is not completed.

5.2.4 Writing a Query

Assigning the result of `countThingsInBackpack` to a temporary variable seems valuable. Can we write similar queries that return a value, such as the number of `Things` on an intersection? Yes. In fact, we have already written queries that return a value—predicates such as `frontIsBlocked`.

KEY IDEA

The value returned must match the query's return type.

Like predicates, a query such as `countThingsHere` (on the robot's intersection) will have a return type and a `return` statement. The return type in this case will be `int` because we expect this query to return an integer value. The `return` statement returns a value whose type must match the query's return type. In this particular query, the `return` statement will return the value stored in the temporary variable at the end of the method. See line 11 of Listing 5-2.

Query
Temporary Variable

Listing 5-2: *A method to count and return the number of* `Things` *on an intersection*

```
 1  /** Count and return the number of things on this robot's current intersection. Replace the
 2   *  things after counting them.
 3   *  @return the number of things on this robot's current intersection. */
 4  public int countThingsHere()
 5  { int numThingsHere = 0;
 6    while (this.canPickThing())
 7    { this.pickThing();
 8      numThingsHere = numThingsHere + 1;
 9    }
10    this.putThing(numThingsHere);
11    return numThingsHere;
12  }
```

It's a good idea for a query to return information without changing the situation in which it was called. If it needs to make changes to the program's state—such as picking `Things` up—the query should undo those changes before returning the answer. This query does so in line 10 where it calls a helper method to put down a specific number of `Things`. This helper method could be implemented using the Count-Down Loop pattern. A query that changes the program's state is said to have **side effects**.

5.2.5 Using a `boolean` Temporary Variable

Temporary variables, as well as parameters and other types of variables, can have one of many different types. `int` is just one of the possibilities. Besides references to objects like `City` and `Robot`, another possibility is the `boolean` type.

To illustrate, consider a predicate to determine whether the right side of a `Robot` is blocked. To answer this, the robot must turn to the right, determine if its path in that direction is blocked, and then somehow remember that answer while it turns back to its original direction and returns the answer. Remembering a value for use later in the method is a perfect application for a temporary variable. In this case, it just happens to be a `boolean` and will store either `true` (the right side is blocked) or `false` (no, it isn't). See Listing 5-3.

Listing 5-3: *The* `rightIsBlocked` *predicate*

```
1 /** Determine whether the right side of this robot is blocked. The robot's state doesn't change.
2  *  @return true if this robot's right side is blocked; false otherwise. */
3 public boolean rightIsBlocked()
4 { this.turnRight();
5   boolean blocked = this.frontIsBlocked();
6   this.turnLeft();
7   return blocked;
8 }
```

Temporary Variable
Predicate

This predicate uses a helper method to determine if its front is blocked. Line 5 could also be written `boolean blocked = !this.frontIsClear()`. The value is stored in the temporary variable `blocked` until it is returned in line 7.

5.2.6 Scope

Temporary variables are always declared within a pair of braces. It may be the pair of braces defining the body of a method or the pair of braces used to define the body of a loop or a clause in an `if` statement. Each of these pairs of braces defines a **block**.

The region of the program in which a variable may be used is called its **scope**. The scope of a variable extends from its declaration to the end of the smallest enclosing block. Four examples are shown in Figure 5-5, where the scope of `tempVar` is shaded. Statements outside of the shaded areas may not use `tempVar`.

(figure 5-5)

Examples of the scope of a temporary variable

```
public void method()
{ int tempVar = 0;
  «statements»
}
```

```
public void method()
{ «statements»
  int tempVar = 0;
  «statements»
}
```

```
public void method()
{ if («booleanExpression»)
  { «statements»
    int tempVar = 0;
    «statements»
  }
  «statements»
}
```

```
public void method()
{ «statements»
  int tempVar = 0;
  while («booleanExpression»)
  { «statements»
  }
  «statements»
}
```

5.3 Nesting Statements

Recall that the general form of the `while` statement is as follows:

```
while («test»)
{ «list of statements»
}
```

The general form of the `if` and `if-else` statements are similar. So far all of our examples have used only method calls and assignment statements in *«list of statements»*. That need not be the case. `if` and `while` statements are also statements and can be included in *«list of statements»*.

5.3.1 Examples Using `if` and `while`

For example, consider the situation shown in Figure 5-6. A robot is an unknown distance from a wall. Between it and the wall are a number of `Thing` objects placed randomly on the intersections. The robot is to pick up one thing from each intersection (if there is one) and stop at the last intersection before the wall.

(figure 5-6)

Task requiring both `if` *and* `while` *statements*

The robot needs to move zero or more times, indicating that a `while` loop is needed. In addition, at each intersection, the robot must execute `pickThing` either once or not

at all, depending on whether or not a thing is present. An `if` statement solves this kind of problem.

These two ideas can be combined in a single method, as shown in Listing 5-4. The `if` statement is said to be **nested** within the `while` statement, just as toys such as blocks or dolls are sometimes nested, one inside another.

Listing 5-4: *An `if` statement nested inside a `while` statement*

```
1  /** Pick up one thing (if there is a thing) from each intersection between this robot and the
2   *   nearest wall it is facing. */
3  public void pickThingsToWall()
4  { while (this.frontIsClear())
5    { this.move();
6      if (this.canPickThing())
7      {  this.pickThing();
8      }
9    }
10 }
```

In `pickThingsToWall`, the `while` loop executes zero or more times to move the robot to the wall. The test ensures the robot will stop when it reaches the wall. Inside the loop, two things happen. First, the robot moves to the next intersection. Once it is there, it asks if it can pick up a `Thing`. If the answer is yes, the robot picks that thing up.

It is also possible to nest `if` or `while` statements within an `if` statement. For example, suppose that when a `Robot` comes to an intersection with a `Thing`, it should turn. However, which way it turns is determined by whether it has `Things` in its backpack. If it does, it turns right; if it doesn't, it turns left. The following nested `if` statements implement these actions:

```
if (this.canPickThing())
{  // There's a thing here, so this robot will turn
  if (this.countThingsInBackpack() > 0)
  { this.turnRight();
  } else
  { this.turnLeft();
  }
}
```

Any kind of statement can be nested within an `if` or `while` statement—including other `if` and `while` statements.

5.3.2 Nesting with Helper Methods

KEY IDEA

Use helper methods to simplify nested statements.

Nesting statements sometimes makes a method hard to understand, particularly if we use several levels of nesting or many steps within the `if` or `while` statement. When a method becomes too complicated, the appropriate approach is to use helper methods. For example, the `pickThingsToWall` method could have been written using helper methods, as shown in Listing 5-5.

Listing 5-5: *Using a helper method to simplify a method*

```
1  /** Pick up one thing (if there is a thing) from each intersection between this robot
2   *    and the nearest wall it is facing. */
3  public void pickThingsToWall()
4  { while (this.frontIsClear())
5    { this.move();
6      this.pickThingIfPresent();
7    }
8  }
9
10 /** Pick up one thing (if there is a thing) from the robot's intersection. */
11 private void pickThingIfPresent()
12 { if (this.canPickThing())
13   { this.pickThing();
14   }
15 }
```

This solution has more lines in total, but each method can be understood more easily than the larger version of `pickThingsToWall` in Listing 5-4.

5.3.3 Cascading-`if` Statements

Another useful form of nesting involves nesting `if-else` statements within `if-else` statements. If the nesting is always done in the else-clause, the effect is to choose at most one of a list of alternatives. For example, suppose that a robot should do exactly one of the actions shown in Table 5-2.

Situation	Action
Front is blocked	Turn around
Can pick a `Thing`	Turn right
Left is blocked	Turn left
Anything else	Move

(table 5-2)

Actions a robot performs in certain situations

It could be that more than one of these situations is true. For example, it could be that the robot's front and left are blocked. We still want the robot to perform only one action. We'll assume that the first matching situation listed in the table should be performed.

This could be coded in Java using a nested `if-else` construct, as follows:

```
 1  if (this.frontIsBlocked())
 2  { this.turnAround();
 3  } else
 4  { if (this.canPickThing())
 5    { this.turnRight();
 6    } else
 7    { if (this.leftIsBlocked())
 8      { this.turnLeft();
 9      } else
10      { this.move();
11      }
12    }
13  }
```

You should trace this code to convince yourself that only one of the actions listed in the table is executed. That is, only one of lines 2, 5, 8, or 10 is executed, no matter what situation the robot is in. Furthermore, when this code is read from top to bottom, the first test that returns `true` determines which statement is executed.

Figure 5-7 illustrates this code graphically using a flowchart.

Each of the else-clauses in this code fragment contain a single statement—another `if-else` statement. In this case, Java allows us to omit the braces. We can then rearrange the line breaks slightly to emphasize that only one of the actions is performed:

```
if (this.frontIsBlocked())
{ this.turnAround();
} else if (this.canPickThing())
{ this.turnRight();
} else if (this.leftIsBlocked())
{ this.turnLeft();
} else
{ this.move();
}
```

Cascading-`if`

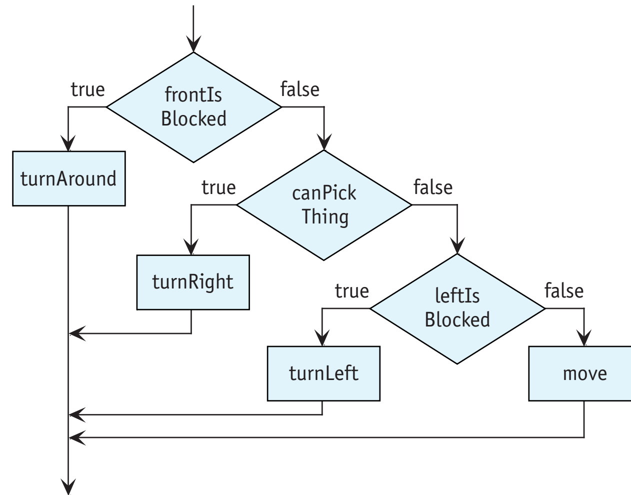
KEY IDEA

Use a cascading-`if` *to choose one of several groups of statements.*

When an `if-else` statement is structured in this way, it is called a **cascading-`if`**. This structure is a clear signal to the reader that only one of the expressions evaluated will cause an action to be taken. To be more specific, the action taken will be associated with the *first* expression that evaluates to `true`.

(figure 5-7)

Flowchart illustrating a nested `if` *statement*



Using the `switch` Statement (optional)

The `switch` statement is similar to the cascading-`if` statement in that both are designed to choose one of several alternatives. The `switch` statement is more restrictive in its use, however, because it uses a single value to choose the alternative to execute. The cascading-`if` can use any expressions desired. This restriction is also the `switch` statement's strength: the reader knows that the decision is based on a single value.

In Section 5.2.1, we used a series of `if` statements to direct a `Robot` based on the number of things on the intersection. Either a cascading-`if` or a `switch` statement would be a better choice because it makes clear that the `Robot` should perform only one of the actions.

The two code fragments shown in Figure 5-8 both implement a variant of the problem just described and behave exactly the same way when executed.

```
int numHere =
  this.countThingsHere();
if (numHere == 0)
{ this.move();

} else if (numHere == 1)
{ this.turnRight();

} else if (numHere == 2)
{ this.turnLeft();

} else
{ this.turnAround();

}
```

```
int numHere =
  this.countThingsHere();
switch (numHere)
{ case 0:
    this.move();
    break;

  case 1:
    this.turnRight();
    break;

  case 2:
    this.turnLeft();
    break;

  default:
    this.turnAround();
}
```

(figure 5-8)

Comparing a cascading-`if` statement and a `switch` statement

The `break` statement causes execution to continue after the end of the `switch` statement. If the `break` statement is not included, execution "falls through" to the next case of the `switch`. For example, in the following code, the `break` is omitted from the first case. The result is that a `Robot` on an intersection with zero `Things` will move and turn right because it "falls through" to the second case. However, a `Robot` on an avenue with one thing will only turn right.

```
switch (this.countThingsHere())
{ case 0:
    this.move();  // Fall though

  case 1:
    this.turnRight();
    break;
}
```

This behavior is sometimes useful if the robot should do exactly the same thing for two or more cases, but this is rare. In reality, the `break` is often forgotten and is a source of bugs. If you choose to use the `switch` statement, it is a good idea to use a compiler setting to warn you if you omit a `break` statement. If you deliberately omit a `break` statement, be sure to document why.

The `default` keyword may be used instead of `case` to indicate the group of actions that occurs if none of the cases match. It is equivalent to the last `else` in the cascaded-`if` statement.

The value used in a `switch` statement must be countable. Integers, as shown in Figure 5-8, fit this description. In later chapters, we will learn about characters and enumerations that also work in a `switch` statement.

5.4 Boolean Expressions

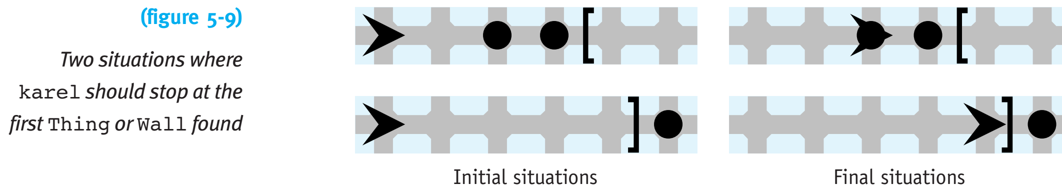
The test in `if` and `while` statements are Boolean expressions that give a yes or no, `true` or `false`, answer to a question. So far, our questions have been simple. As our programming skills grow, however, we will want to ask more complex questions, which will need more complex Boolean expressions to answer.

5.4.1 Combining Boolean Expressions

Consider a situation in which `karel` is facing east. It is known that in the distance are things and walls. We want `karel` to travel forward, stopping at the first thing or wall it comes to. Figure 5-9 shows two such situations.

(figure 5-9)

Two situations where `karel` *should stop at the first* `Thing` *or* `Wall` *found*



The robot might need to move zero or more times to reach the first thing or wall, so a `while` statement is appropriate. To construct it, we follow the four-step process discussed earlier. The first step is already apparent: the body of the loop should contain a `move` statement.

The second step in the process requires some thought. We want the robot to stop when it reaches a thing or a wall. We have a predicate, `canPickThing`, to determine if it is beside a thing. Another predicate, `frontIsClear`, will determine when a wall is reached. But we do not have a predicate that combines these two tests.

Fortunately, programming languages have **operators** that can combine Boolean expressions into more complex expressions. You are already familiar with operators from the mathematics you have studied: plus, minus, multiply, and divide are all operators that combine two arithmetic expressions to create a more complex expression. The equivalents for Boolean expressions are the **and** and **or** operators.

KEY IDEA

Both sides of an "and" must be true to do the action.

We often use Boolean operators in our everyday language. You might say, "I will go swimming if the weather is hot and sunny." From this statement, I know that if the weather is cloudy, you will not go swimming. Similarly, if the weather is cool, you will not go swimming. In order to go swimming, both expressions joined by "and" must be true.

On the other hand, if you say "I will go swimming if it is hot or sunny," we might question your sanity. With this statement, you might go swimming in a thunderstorm (if it happens to be hot that day) or you might go swimming in a frozen pond (if it happens to be a sunny winter day). The "or" operator requires a minimum of one of the two tests to be true. Of course, if it happens to be both hot and sunny, you would still go swimming.

KEY IDEA

Only one side of an "or" is required to be true to do the action.

Java's logical operators work in the same way except that instead of writing "and," we write `&&`, and instead of writing "or," we write `||`. Like English, an expression including `&&` is true only if both expressions it joins are true. For an expression including `||` to be true, one or both of the expressions it joins must be true.

KEY IDEA

In Java, write "and" as `&&` and write "or" as `||`.

In the earlier problem, we want `karel` to stop when it's beside a thing or its front is blocked. This can be written in Java as follows:

```
karel.canPickThing() || karel.frontIsBlocked()
```

Step 2 of the four-step process says that we should negate this expression to find out when the loop should continue. We can negate the entire expression by wrapping it in parentheses and using the `!` operator, as follows:

```
1 while (!(karel.canPickThing() || karel.frontIsBlocked()))
2 { karel.move();
3 }
```

The Form of Legal Expressions

The informal descriptions of `&&` and `||` given previously mention the "expressions it joins." Let's be more precise about what constitutes a legal expression. There are four rules:

1. Literal values such as `true`, `false`, and `50` are legal expressions. The type of the expression is the type of the literal. For example, `boolean` and `int` in these examples.
2. A variable is a legal expression. The type of the expression is the type of the variable.
3. A method call whose arguments are legal expressions with the appropriate types is a legal expression. The type of the expression is the return type of the method.
4. An operator whose operands are legal expressions with the appropriate types is a legal expression. The type of the expression is given by the return type of the operator. Operators include `&&`, `||`, `!`, the comparison operators, and the arithmetic operators. Their **operands** are the expressions they operate on.

The first two rules just set the groundwork. The power is all in the last two rules, which let us combine expressions to any level of complexity. For example, within the `Robot` class, `this.canPickThing()` and `this.frontIsClear()` are both expressions (by rule 3). These two expressions can be joined with an operator such as `&&` to

make a more complex expression (rule 4):

```
this.canPickThing() && this.frontIsClear()
```

Two other expressions are `this.getAvenue()` and `0` (rules 3 and 1). They can be joined by the operator > to form a new expression (by rule 4). This expression has type `boolean` and can be combined with the previous expression by rule 4. For example, using `||` gives the following expression:

```
this.canPickThing() && this.frontIsClear() ||
this.getAvenue() > 0
```

This expression can also be combined with other expressions in ever-increasing complexity.

Evaluating Boolean Expressions

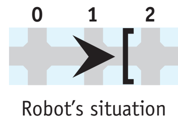
Suppose we have a complex expression. How can we evaluate it to find its value? We can use a technique we'll call **evaluation diagrams** to annotate the expression.

To construct an evaluation diagram, begin by drawing an oval around each literal, variable, and parameterless method call. Write the expression's type above the oval and value below the oval.

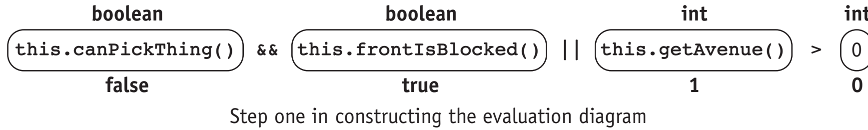
The second step is to repeatedly draw an oval around an operator with its operands or a method with its arguments until the entire expression is enclosed in a single oval. For each oval drawn:

- Verify that all operands and arguments enclosed in the new oval already have ovals around them, from either the first step or a previous iteration of the second step.
- Verify that the type of each operand or argument is appropriate for the operator or method call being enclosed in the new oval. For example, you may not draw an oval around the `&&` operator if one of its operands has type `int`. If such a situation occurs, it means that the expression as a whole is not well-formed and will be rejected by the Java compiler. Some operators, such as negation (`!`), use only one operand. In that case, the oval will include only the operator and one operand.
- Write the type returned by the operator or method above the oval and the value returned below the oval.

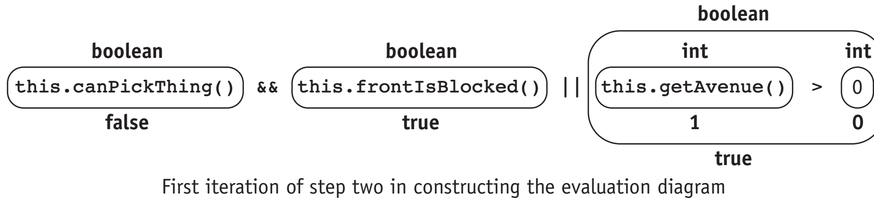
Figure 5-10 shows the process of constructing an evaluation diagram. The top cell of the diagram shows the robot's situation. The bottom four cells of the diagram illustrate the series of steps required to construct the diagram. From the last step we conclude that in the given situation, the expression returns `true`.



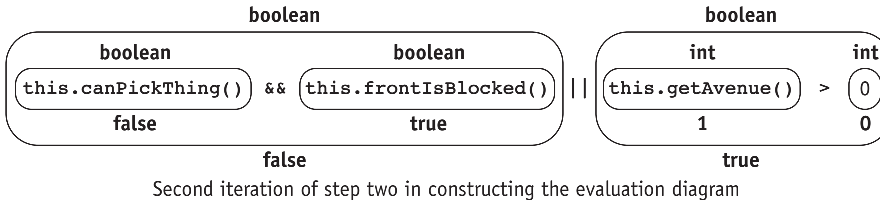
Robot's situation



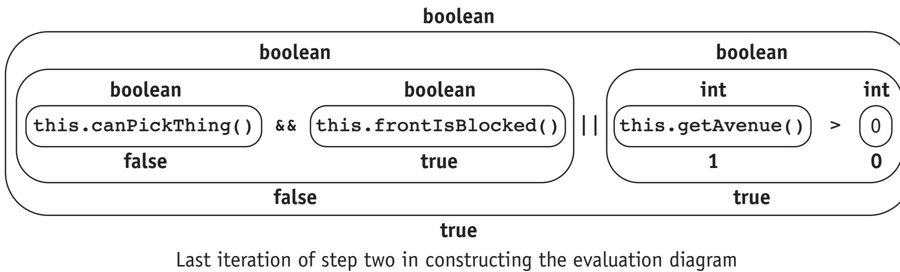
Step one in constructing the evaluation diagram



First iteration of step two in constructing the evaluation diagram



Second iteration of step two in constructing the evaluation diagram



Last iteration of step two in constructing the evaluation diagram

(figure 5-10)

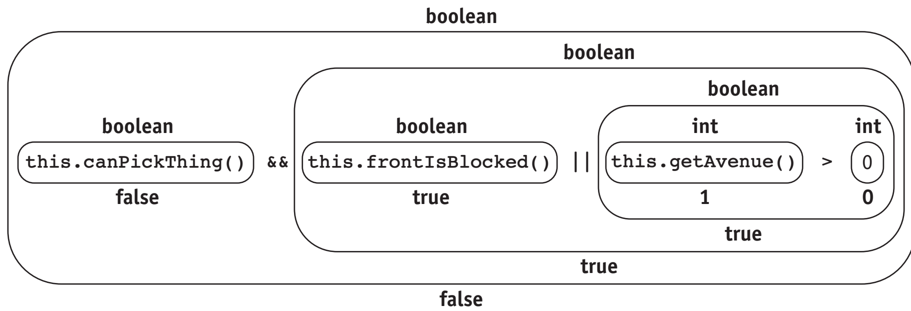
Evaluating a Boolean expression using an evaluation diagram

Operator Precedence

You may have noticed that some discretion was involved in choosing which operator to include in an oval. For example, in the second iteration of step two in Figure 5-10, we could have drawn the oval around the `||` operator instead of the `&&` operator. The resulting evaluation diagram is shown in Figure 5-11. Notice that the value of the expression as a whole is `false` rather than `true`.

(figure 5-11)

Alternative (and incorrect) evaluation diagram



The operators are chosen in order of their precedence. **Precedence** denotes the priority they are given when evaluating the expression. The operators we have encountered, from the highest precedence to the lowest, are listed in Table 5-3. We see that `&&` is listed before `||`. Therefore, the expression diagram in Figure 5-11 is incorrect because it drew an oval around `||` when `&&` should have been chosen.

(table 5-3)

Operator precedence, from highest to lowest, of the operators encountered so far

Operator	Precedence
methodName(*parameters*)	15
`!`	14
`*  /  %`	12
`+  -`	11
`<  >  <=  >=`	9
`==  !=`	8
`&&`	4
`||`	3

It may be that the normal precedence rules are not what you want. For example, you really do want the answer shown in Figure 5-11. In that case, override the precedence rules with parentheses—just like you would in an arithmetic expression. The following example has an expression diagram as shown in Figure 5-11:

```
this.canPickThing() &&
        (this.frontIsBlocked() || this.getAvenue() > 0)
```

LOOKING AHEAD

These rules are not yet complete. We will expand them in Chapter 7.

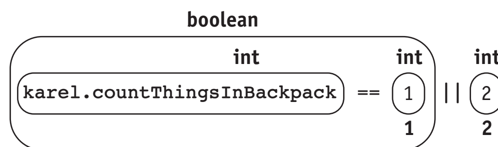
If an expression has two or more operators with equal precedence, circle them in order from left to right.

A Common Error in Combining Expressions

Perhaps we want karel to turn around if the number of things in its backpack is either 1 or 2. A direct translation of this English statement into Java might be as follows:

```
if (karel.countThingsInBackpack() == 1 || 2)
{ karel.turnAround();
}
```

If we attempt to diagram this expression, however, we will encounter the problem shown in Figure 5-12. The next iteration of the algorithm calls for drawing an oval around the || operator, which requires two Boolean operands. However, the evaluation diagram has one Boolean operand and one integer operand. This situation tells us that the expression is incorrectly formed and will be rejected by the Java compiler. Notice that we were able to determine this without knowing how many things are in karel's backpack. The analysis of the expression types, as recorded on top of the ovals, was sufficient.



(figure 5-12)

Evaluation diagram for an incorrectly formed expression

A correct expression for determining if karel has one or two things in its backpack is as follows:

```
if (karel.countThingsInBackpack() == 1 ||
    karel.countThingsInBackpack() == 2)
{ karel.turnAround();
}
```

5.4.2 Simplifying Boolean Expressions

Sometimes Boolean expressions can become quite complicated as they are combined and negated. Simplifying them can be a real service, both to yourself as the programmer and to others who need to understand your code.

Simplifying Negations

Many simplifications are common sense—for example, double negatives. `!!karel.frontIsClear()` is the same as `karel.frontIsClear()`. Other such equivalencies are shown by example in Table 5-4.

(table 5-4)

Examples of equivalent, simplified expressions

Expression	Simplification
`!!karel.frontIsClear()`	`karel.frontIsClear()`
`!karel.frontIsBlocked()`	`karel.frontIsClear()`
`!(this.getAvenue() == 0)`	`this.getAvenue() != 0`
`!(this.getAvenue() != 0)`	`this.getAvenue() == 0`

De Morgan's Laws

When negations involve more complex expressions, it's easy to get mixed up. Faced with this problem, Augustus De Morgan (1806–1871) introduced what have become known as De Morgan's Laws, which formalize the process of finding the opposite form of a complex test. De Morgan's Laws state the following equivalencies (≡ means that the expression on the left is equivalent to the expression on the right):

```
!(b1 && b2) ≡ !b1 || !b2    (1st law)
!(b1 || b2) ≡ !b1 && !b2    (2nd law)
```

where `b1` and `b2` are arbitrary Boolean expressions.

These laws can be used to simplify the following expression:

```
!(karel.canPickThing() ||
      (karel.leftIsBlocked() && karel.rightIsBlocked()))
```

This code is equivalent to the following by De Morgan's second law:

```
!karel.canPickThing() &&
      !(karel.leftIsBlocked() && karel.rightIsBlocked())
```

This can be further simplified by applying the first law:

```
!karel.canPickThing() &&
      (!karel.leftIsBlocked() || !karel.rightIsBlocked())
```

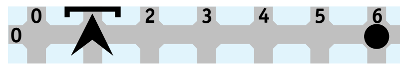
This can be simplified again by restating each negated predicate, using new predicates if necessary:

```
!karel.canPickThing() &&
      (karel.leftIsClear() || karel.rightIsClear())
```

5.4.3 Short-Circuit Evaluation

Suppose you have a robot in the situation shown in Figure 5-13 that is about to execute the following code fragment:

```
if (this.frontIsClear() && this.thingOnSixthAvenue())
{ this.putAllThings();
}
```



(figure 5-13)

Time-consuming test

We can observe two things. First, the robot can find out quickly if its front is clear. On the other hand, it will take a relatively long time to move all the way to Sixth Avenue to find out if a `Thing` is there. Second, when the robot is in a situation like this, it doesn't need to waste its time checking Sixth Avenue. The definition of "and" says that if the first part of the test is false (the robot's front is *not* clear), then the entire test will be false. It doesn't matter whether the second part of the test is true or false.

With these two observations, we can conclude that the following is a more efficient way to write the previous code fragment:

```
if (this.frontIsClear())
{ if (this.thingOnSixthAvenue())
  { this.putAllThings();
  }
}
```

This fragment will only cause the robot to check Sixth Avenue if that test will really make a difference to the robot's behavior.

KEY IDEA

Java only performs a test if it needs to.

However, running these two code fragments in the situation shown in Figure 5-13 results in exactly the same behavior. In neither case does the robot check Sixth Avenue. This is because Java uses **short-circuit evaluation**. When evaluating a Boolean expression `test1 && test2`, Java will only execute `test2` if `test1` is true. If `test1` is false, Java knows that executing `test2` is a waste of time and doesn't do it.

Similarly, in the expression `test1 || test2`, `test2` will only be executed if `test1` is false. If `test1` is true, the entire expression will be true regardless of whether `test2` is true or false.

5.5 Exploring Loop Variations

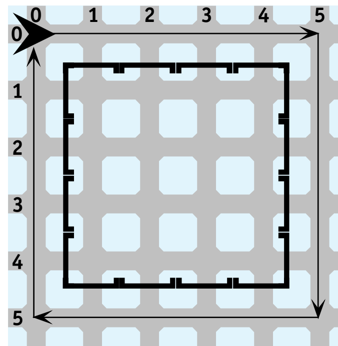
The `while` loop is only one of several ways that Java can execute a code fragment repeatedly. In this section, we will explore the `for` statement and two variations of the `while` statement.

5.5.1 Using a `for` Statement

Sometimes we know before a loop begins exactly how many times we want it to execute. For example, consider a problem in which a robot named `suzie` must move clockwise around a square defined by walls, as shown in Figure 5-14.

(figure 5-14)

Moving around a square



To solve the problem, `suzie` must traverse exactly four sides of the square—no more, no less. For each side, `suzie` must move exactly five times. At each corner, `suzie` must turn left exactly three times.

KEY IDEA

Use `while` when the number of iterations is unknown. Use `for` when the number is known.

A `while` loop works well when statements must be repeated an unknown number of times—while some condition is true. However, `suzie`'s situation is different. Here, we know exactly how many times the statements must be executed even before the loop begins. Java includes the `for` statement just for such situations.

The Form of the `for` Statement

The form of the `for` statement used to repeat statements a fixed number of times is as follows:

```
for (int «counter» = 0; «counter» < «limit»;
                                    «counter» = «counter» + 1)
{ «statements to repeat»
}
```

where

- *«statements to repeat»* are the instructions to be executed each time through the loop. They are called the body of the loop, the same term used for the statements in the `while` loop.
- *«counter»* is an identifier or name, such as `numTurns` or `sideCount`.
- «limit» is the number of times *«statements to repeat»* should be executed.

Here is an example of `turnRight` implemented with a `for` loop:

```
public void turnRight()
{ for (int turns = 0; turns < 3; turns = turns + 1)
  { this.turnLeft();
  }
}
```

Counted Loop

In the `for` loop, `turns` is the «counter» that keeps track of how many times the `turnLeft` method has been executed. The «limit», or total number of times we want `turnLeft` to execute, is 3.

The `for` statement is nothing more than an abbreviation of a particular form of the `while` loop. The component parts of the `for` statement can be rearranged to create a `while` loop that behaves in exactly the same way.

KEY IDEA

A `for` statement is a shortcut for a common form of the `while` loop.

```
{
  int «counter» = 0;
  while («counter» < «limit»)
  { «statements to repeat»
    «counter» = «counter» + 1;
  }
} // Note: «counter» is not available beyond this closing brace
```

Examples of the `for` Statement

To gain further comfort with the `for` statement, let's solve the problem illustrated in Figure 5-14. We will extend `Robot` to create the class `SquareMover`. We can use stepwise refinement and pseudocode to solve the problem. To move around the square, the robot needs to move along four sides:

```
To move around a square:
 for (4 times)
 { move along one side
 }
```

To move along one side, the robot needs to move five times:

```
To move along one side:
 for (5 times)
```

```
{ move
}
turn right
```

Finally, to turn right, it needs to turn left three times:

```
To turn right:
  for (3 times)
  { turn left
  }
```

In each of these refinements, the robot must perform an action a number of times—and that number is known before the loop begins executing. Such circumstances are ideal for using a `for` statement. The class definition corresponding to this pseudocode is shown in Listing 5-6.

FIND THE CODE

ch05/squareMover/

Counted Loop

Listing 5-6: *A class of robot that moves around squares*

```
1  import becker.robots.*;
2
3  /** A class of robot that goes around squares.
4  *
5  *     @author Byron Weber Becker */
6  public class SquareMover extends Robot
7  {
8    public SquareMover(City c, int str, int ave, Direction dir)
9    { super(c, str, ave, dir);
10   }
11
12   /** Move around a square by traversing each of its four sides. */
13   public void moveAroundSquare()
14   { for (int side = 0; side < 4; side = side + 1)
15     { this.moveAlongSide();
16     }
17   }
18
19   /** Move along one side of the square by moving 5 times. */
20   private void moveAlongSide()
21   { for (int moves = 0; moves < 5; moves = moves + 1)
22     { this.move();
23     }
24     this.turnRight();
25   }
26
27   /** Turn right by turning left three times. */
28   private void turnRight()
```

Listing 5-6: *A class of robot that moves around squares* (continued)

```
29    { for (int turns = 0; turns < 3; turns = turns + 1)
30      { this.turnLeft();
31      }
32    }
33  }
```

Java provides a shortcut for *«counter»* = *«counter»* + 1. This statement occurs so frequently that Java allows the abbreviation *«counter»*++, which means "add 1 to the value stored in *«counter»*." Another abbreviation is *«counter»* += *«expression»*. It means to add the value on the right side to the variable on the left.

LOOKING AHEAD

This, and other shortcuts, are discussed in Section 7.2.5.

Finally, it should be noted that the `for` statement is more flexible than implied by these examples. The *«counter»* need not start at 0; any Boolean expression can be used for the test; and *«counter»* = *«counter»* + 1 can be replaced by a more general statement. In particular, the `for` statement's template can be generalized as follows:

```
for («initialization»; «test»; «update»)
{ «statements to repeat»
}
```

For example, the `turnRight` method could also be written as

```
private void turnRight()
{ for (int turns = 3; turns > 0; turns = turns - 1)
  { this.turnLeft();
  }
}
```

5.5.2 Using a `do-while` Loop (optional)

The `while` loop always performs its test before the body of the loop is executed. If the test happens to be false right away, the loop's body may not be executed at all. Another loop, the `do-while` loop, performs its test after the loop body executes. This means that it will always execute at least once.

KEY IDEA

A `do-while` loop always executes at least once.

The general form of the `do-while` loop can be expressed as follows:

```
do
{ «statements to repeat»
} while («test»);
```

The loop begins by executing *«statements to repeat»*. After each execution, the *«test»* is evaluated. If it is `true`, execution resumes at the `do` keyword and the body of the loop is executed again. If the test is false, execution resumes with the statement after the `while` keyword.

It is unusual to have a loop that *always* executes at least once, and so the `do-while` loop itself is unusual. A search of three projects[2] totaling more than 20,000 lines of code revealed not even one `do-while` loop.

5.5.3 Using a `while-true` Loop (optional)

The `while-true` loop is the most flexible loop available in Java. The other looping forms, including the `for` loop, test either at the beginning of the loop's body or at the end. The `while-true` can test any place you want—and can even test several times during the body's execution.

A Brief History Lesson on Structured Programming

All modern programmers advocate some form of **structured programming** whereby a control structure such as a loop, `if`, or a method restricts the program to entering at only a single point and often also restricts how it exits. This is in stark contrast to early programming languages that did not have such restrictions and permitted programmers to write **spaghetti code**, which was as hard to untangle as a bowl of spaghetti.

Why is this history lesson relevant? The `while-true` loop is less structured than the other loops because it allows multiple exits from the loop. Your instructor may feel uncomfortable with that and not want you to use such loops. On the other hand, many programmers believe that it strikes an excellent balance between the rigor of one-entry/one-exit structured programming and the flexibility to solve problems easily. One such person is Eric Roberts, a noted computer science educator who wrote a paper[3] on the topic. Another is the designer of the Turing programming language, in which `while-true` is used for all forms of looping except the `for` loop.

[2] The `becker` library (10,500 lines), a testing tool named `junit3.8.1` (5,000 lines), and an implementation of a marine biology simulation (5,000 lines).

[3] "Loop exits and structured programming: reopening the debate," pages 268–272 in *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, ACM Press, March 1995.

The Form of a `while-true` Loop

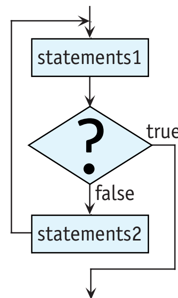
At first glance, a `while-true` loop looks like it will execute forever. That's because the test is the value `true`—which can never be `false` and cause the loop to end. That observation tells us there *must* be another way out of the loop.

The loop uses an `if` statement combined with a `break` statement to end the loop. It's common for the `break` to be the only statement in the `if`, so it can all be put on a single line. It uses the following form:

```
1  while (true)       // use a break statement to exit!
2  { «optional statements1»
3    if («test»)    {     break;   }
4    «optional statements2»
5  }
```

The test in line 1 is always `true` and so the program always enters the loop. When execution reaches the `if` statement in line 3, it performs its test. If the test is `false`, execution resumes with the optional statements in line 4. If the test is `true`, the `break` instruction causes execution to resume after the end of the loop.

This flow of control is summarized in Figure 5-15. All the statements in the loop are executed until the *«test»* is `true`. At that point, the `break` statement causes the loop to end.



(figure 5-15)

Flowchart for the `while-true` *loop*

The statements before the test (line 2 in the preceding code) might be omitted, in which case the loop is like a standard `while` loop but with the test negated. On the other hand, if the statements after the test (line 4 in the preceding code) are omitted, the loop is like a `do-while` loop.

The loop can also have several tests. This may make the loop easier to understand than an equivalent `while` loop with a compound Boolean expression for the test.

LOOKING AHEAD

See Programming Exercise 5.7.

An Example

Consider again the fence-post problem shown in Figure 5-2. We wanted a robot to pick up all of the things between its current location and a wall. We solved it with the following method, noting that we needed an extra call to `pickThing` after the loop.

Loop-and-a-Half

```
public void clearThingsToWall()
{ while (this.frontIsClear())
  { this.pickThing();
    this.move();
  }
  this.pickThing();
}
```

The extra call to `pickThing` is needed because we need the robot to pick up four things but move only three times.

Here is the same problem solved with a `while-true` loop:

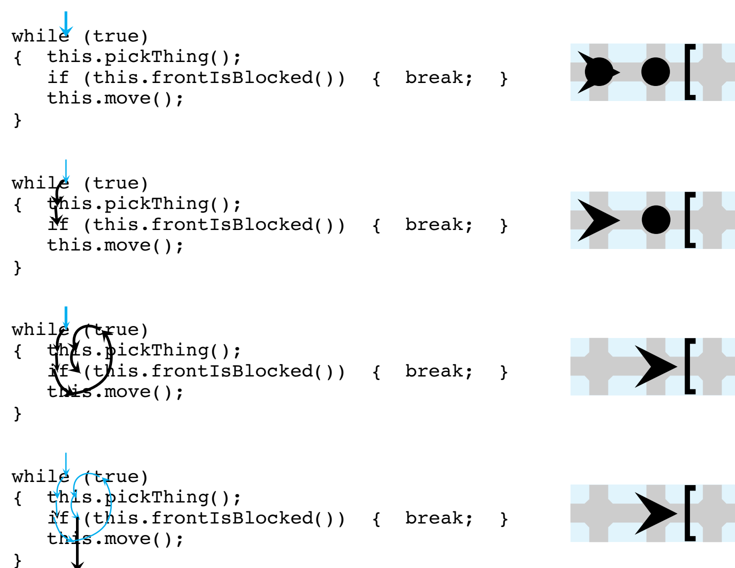
Loop-and-a-Half

```
public void clearThingsToWall()
{ while (true)
  { this.pickThing();
    if (this.frontIsBlocked())     {      break; }
    this.move();
  }
}
```

When only two things remain to be picked up, this code executes as illustrated in Figure 5-16.

(figure 5-16)

Illustrating the execution of a `while-true` loop



The `while-true` loop provides a general solution to the fence-post problem, also known as the **loop-and-a-half** problem. Solving these kinds of problems with a traditional `while` statement requires repeating some of the code, because the loop must execute an extra "half" iteration to perform the last action. In this case, the repeated code is the call to `pickThing`. By putting the test inside the body of the loop, the repeated code is no longer needed.

5.5.4 Choosing an Appropriate Looping Statement

We have studied a number of different kinds of looping statements. Table 5-5 provides guidelines on when to use each kind.

(table 5-5)

Guidelines for choosing a looping statement

If...	Then...
a parameter refers to the number of times the loop will execute and the value is not needed for other purposes...	use a count-down loop or a `for` statement.
the number of times the loop will execute is known before the loop is entered...	use a `for` statement.
the loop might execute zero times...	use a `while` statement.
the loop has a relatively simple test that appears at the top of the loop...	use a `while` statement.
the loop always executes at least once...	use a `do-while` statement.
the loop executes an extra "half" time for a fence-post problem...	use a `while-true` loop.
the loop has multiple exit tests or a complex test that can be more easily understood as separate tests...	use a `while-true` loop.

5.6 Coding with Style

As we have seen in previous sections, the style of our code makes a difference in how easily it can be understood. Selection and repetition statements such as `if`, `while`, and `for` are no different in this sense—we must consider the style of these statements to make sure they are easy to interpret.

The most important elements of style, stated briefly, are:

- Use stepwise refinement to avoid having deeply nested statements or long sequences of statements.
- Use positively stated, simple Boolean expressions.
- Indent your code so the visual structure reflects the logical structure.

The following subsections explore these ideas more carefully.

5.6.1 Use Stepwise Refinement

A long list of statements inside the body of a loop or `if` statement can cause the reader to lose track of the loop or `if` statement as a whole. It should be easy for the reader to remember when a loop terminates or what case is being handled by an `if` statement. Long bodies in either structure make doing so difficult.

Using stepwise refinement naturally breaks long bodies into smaller steps. Some programmers put the entire body into a helper method so that the loop or `if` statement contains only one line—the invocation of the helper method.

5.6.2 Use Positively Stated Simple Expressions

One of the most crucial aspects of good style is to keep tests easy to understand. First, avoid negations if you can because people usually find them harder to understand than positive statements. It is easier to understand `while (this.frontIsBlocked())` than `while (!this.frontIsClear())`, for example. This style may mean that you need to define your own predicate to put the test in a positive form.

A second way to keep tests easy to understand is to use predicates with descriptive names. For example, `if (this.isFacingSouth())` is easier to understand than `if (this.getDirection() == Direction.SOUTH)`.

If you will be using the same test several times in the program, writing the predicate is particularly worthwhile.

A third possibility, applicable to `if-else` statements, is to rewrite them with the goal of making them simpler and easier to understand. Rewriting an `if` statement should not change the execution of the program, only the way in which it is written. These techniques are explored in the following subsections.

Test Reversal

Consider the following code:

```
if (!this.frontIsClear())
{ this.turnLeft();
} else
{ this.move();
}
```

When the robot's front is not clear, it turns left. Otherwise it moves forward. This can be rewritten to use the opposite test if we interchange the then-clause and the else-clause:

```
if (this.frontIsClear())
{ this.move();
} else
{ this.turnLeft();
}
```

KEY IDEA

An `if` statement executes the same way if you negate the test and swap the then-clause with the else-clause.

This code is easier to read and understand, primarily because we eliminated the negation in the test. Another way to make this code easier to read is to replace `!this.frontIsClear()` with a new predicate, `this.frontIsBlocked()`.

Occasionally, we may write an `if-else` statement with an empty then-clause:

```
if (this.canPickThing())
{ // do nothing
} else
{ this.turnLeft();
}
```

Test reversal allows us to rewrite this code fragment as follows:

```
if (!this.canPickThing())
{ this.turnLeft();
} else
{ // do nothing
}
```

When written this way, it's easy to see that we can drop the else-clause and use only the Once or Not At All pattern.

```
if (!this.canPickThing())
{ this.turnLeft();
}
```

Bottom Factoring

Compare the following two fragments of code.

```
if (this.canPickThing())
{ this.pickThing();

  this.turnAround();
} else

{ this.putThing();
  this.turnAround();
}
```

```
if (this.canPickThing())
{ this.pickThing();

} else
{ this.putThing();

}
this.turnAround();
```

Both code fragments result in the same final situation. In both fragments, the robot finishes by turning around. The code on the right, however, makes this more obvious by moving `this.turnAround()` outside of the `if-else` statement. Only the actions that actually depend on the test are left inside the `if-else` statement.

Moving identical lines of code that appear at the end of both the then-clause and the else-clause to just after the `if-else` statement is called **bottom factoring**.

Top Factoring

When identical code appears at the beginning of the then-clause and the else-clause, we may be able to **top factor**. Top factoring means moving identical code from the beginning of the then- and else-clauses to just before the `if-else` statement. For example:

```
if (this.canPickThing())
{ this.turnAround();
  this.pickThing();
} else
{ this.turnAround();
  this.putThing();
}
```

```
this.turnAround();
if (this.canPickThing())
{
  this.pickThing();
} else
{
  this.putThing();
}
```

Both versions of this code will always result in the same final situation. In both versions, the robot always turns around, regardless of the test's result.

Top factoring is not as simple as bottom factoring, however. If the identical lines of code affect the outcome of the test, they *cannot* simply be moved. Consider the following example:

```
if (this.isFacingNorth())
{ this.turnAround();
  this.pickThing();
} else
{ this.turnAround();
  this.putThing();
}
```

```
this.turnAround();
if (this.isFacingNorth())
{
  this.pickThing();
} else
{
  this.putThing();
}
```

KEY IDEA

Top factor only if the code moved outside the `if` statement has no effect on the test.

Suppose the robot's initial situation is facing north on an intersection with a thing. Executing the code on the left leaves the robot facing south and having picked up one thing. Executing the code on the right also leaves the robot facing south, but this time the robot has put a thing down rather than picking a thing up.

5.6.3 Visually Structure Code

Another important stylistic rule is to line up braces vertically and indent the bodies of loops. This rule is the same as appropriately indenting methods.

If you read code written by someone else, you may notice that sometimes braces are omitted in an `if` or `while` statement. When the body consists of a single statement, the braces surrounding it are optional. For example, both of the following statements are legal:

```
if (this.frontIsClear())
  this.move();
else
  this.turnRight();
```

```
while (this.frontIsClear())
  this.move();
```

There are, however, dangers in leaving out the braces. The first comes from adding code. Suppose that after executing the `if` statement we realize that if the front is not clear, the robot should turn right and move. We might add an extra statement, as in the following example:

```
if (this.frontIsClear())
  this.turnLeft();
else
  this.turnRight();
  this.move();
```

In spite of the indentation, the `move` will occur whether the front is clear or not, which is not what was desired. Why? Braces should group the new line with the instruction to turn right. Without the braces, a compiler interprets the preceding code as follows:

```
if (this.frontIsClear())
{ this.turnLeft();
} else
{ this.turnRight();
}
this.move();
```

The compiler is interpreting the code correctly. The mistake is the programmer's in using white space to imply an incorrect program structure.

The second danger is called a **dangling else**. If braces are not included, where the `else` goes can be confusing. For example, consider the following fragment:

```
if (this.frontIsClear())
  if (this.canPickThing())
    this.pickThing();
else
  this.turnLeft();
}
```

The question is, which `if` goes with the `else`? The indentation seems to say the `else` should go with the first `if` statement. In fact, an `else` goes with the closest unmatched `if`. That is, the code is equivalent to the following:

```
if (this.frontIsClear())
{ if (this.canPickThing())
    this.pickThing();
  else
    this.turnLeft();
}
```

If we want to write code that does what the indentation implies, we are forced to add braces so that the `if` without an `else` is clearly identified, as follows:

```
if (this.frontIsClear())
{ if (this.canPickThing())
    this.pickThing();
} else
    this.turnLeft();
```

5.7 GUI: Using Loops to Draw

LOOKING BACK

See Sections 2.7.1 (a `main` *method), 2.7.2 (overriding* `paintComponent`*), and 4.7.2 (scaling images).*

In previous chapters, we learned how to draw a figure, such as a line, by writing a class extending `JComponent` and overriding the `paintComponent` method. An instance of this class is set as the content pane of a `JFrame`. The following simple class overrides `paintComponent` to scale the image and the stroke, and then draws a single line from the upper-left to the lower-right (see Listing 5-7).

FIND THE CODE

ch05/lineArt/

Listing 5-7: *Drawing a single diagonal line*

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  /** Create a component that paints our "art."
5   *   @author Byron Weber Becker */
6  public class ArtComponent extends JComponent
7  {
8    public ArtComponent()
9    { super();
10     this.setPreferredSize(new Dimension(300,300));
11   }
12
13   /** Paint the component with our "art." */
14   public void paintComponent(Graphics g)
15   { super.paintComponent(g);
```

Listing 5-7: *Drawing a single diagonal line* (continued)

```
16
17        // Standard stuff to scale the image.
18        Graphics2D g2 = (Graphics2D) g;
19        g2.scale(this.getWidth()/11, this.getHeight()/11);
20        g2.setStroke(new BasicStroke(1.0F/this.getWidth()));
21
22        // draw our "art"
23        g2.drawLine(1, 1, 10, 10);
24    }
25  }
```

With a `for` loop, we can draw a shape over and over again. But if we replace line 23 with the following loop, we draw the same line repeatedly in the same place, having no visible effect.

```
for (int line = 1; line <= 10; line = line + 1)
{ g2.drawLine(1, 1, 10, 10);
}
```

What we need is a way to change the position of the line in each iteration of the loop.

LOOKING AHEAD

The number 10 has a special significance in this code. In Chapter 6 we will see a better way to handle it using named constants.

5.7.1 Using the Loop Counter

One way to change the position of the line with each iteration of the loop is to use `line`, the loop counter, as a parameter to `drawLine`. The parameters to `drawLine` are integers, and `line` holds integers. The value of this integer changes from 1 to 10 as the loop executes. With each iteration of the loop, a different value is passed to `drawLine`, changing the position of each of the 10 lines.

For example, we can replace line 23 in Listing 5-7 with the following loop:

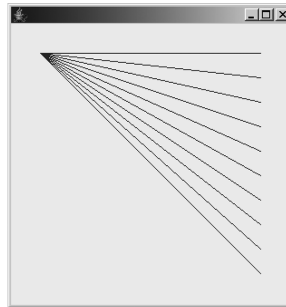
```
for (int line = 1; line <= 10; line = line + 1)
{ g2.drawLine(1, 1, 10, line);
}
```

Counted Loop

This loop yields the image shown in Figure 5-17. Each of the 10 iterations of the loop draws a line. The location of the left end-point is fixed, but the right end-point's location varies according to the current value stored in `line`, the loop's counter variable.

The loop counter can be used for more than one of drawLine's parameters. What would be the effect of the following code fragment?

```
for (int line = 1; line <= 10; line = line + 1)
{ g2.drawLine(1, line, 10, line);
}
```



(figure 5-17)

Image resulting from using a loop to control drawing lines

5.7.2 Nesting Selection and Repetition

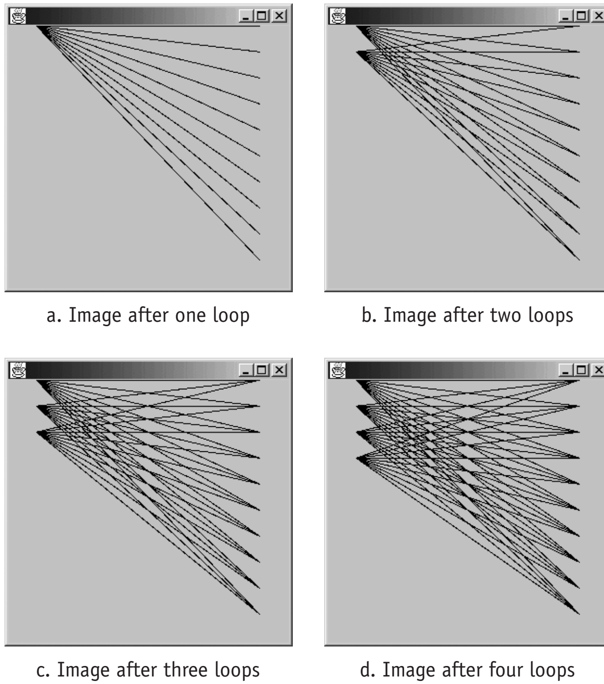
In Section 5.3 we saw that `if` statements and `while` statements can be nested—that is, one can be placed inside the other. `for` statements control any number of `if`, `while`, and `for` statements. As such, `for` statements may be nested, just like `if` and `while`. We could, for example, replace the single `drawLine` command at line 23 in Listing 5-7 with the following **nested loop**.

Counted Loop

```
for (int left = 1; left <= 5; left = left + 1)
{  for (int right = 1; right <= 10; right = right + 1)
   { g.drawLine(1, left, 10, right);
   }
}
```

These five lines cause a total of 50 lines to be drawn. The outer loop executes five times. In each of the five iterations of the outer loop, the inner loop executes completely, performing 10 iterations each time.

The image drawn after one iteration of the outer loop looks like Figure 5-18a. After two iterations of the outer loop, it looks like Figure 5-18b, and so on. Each iteration of the outer loop draws one more spray of lines (see Figures 5-18c and d). Each spray is drawn by the inner loop. In each iteration through the outer loop, the variable `left` has a value one larger than the previous iteration. When passed as an argument to `drawLine`, the coordinates of the left end of the line change.

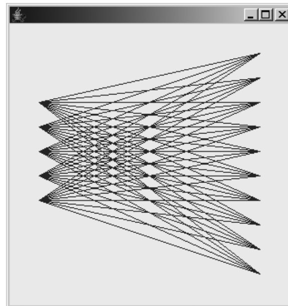


(figure 5-18)

Images produced by a nested loop after 1, 2, 3, and 4 iterations of the outer loop

The initial value in a `for` loop need not be 0. For example, the following nested loop starts the outer loop at 3 instead of 0. The result is shown in Figure 5-19.

```
for (int left = 3; left <= 7; left = left + 1)
{ for (int right = 1; right <= 10; right = right + 1)
  { g.drawLine(1, left, 10, right);
  }
}
```



(figure 5-19)

Starting the outer loop at 3

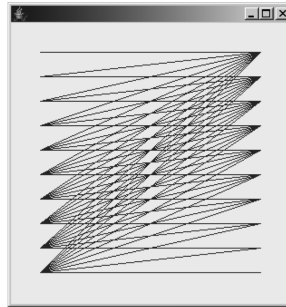
The control variable from the outer loop can also be used as a starting value or a limiting value in the inner loop. Here, the test for the inner loop is `right <= left`:

```
for (int left = 1; left <= 10; left = left + 1)
{ for (int right = 1; right <= left; right = right + 1)
  { g.drawLine(1, left, 10, right);
  }
}
```

The first time the outer loop executes, the variable `left` has a value of 1. This value limits the inner loop to executing 1 time. The second time through the outer loop, `left` has a value of 2. The inner loop draws a spray consisting of two lines. The third time through the outer loop, `left` has a value of 3 and so the inner loop draws a spray of 3 lines. See Figure 5-20.

(figure 5-20)

Limiting the inner loop with the outer loop's control variable

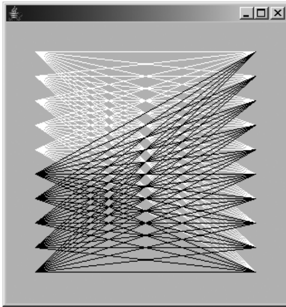


We can also add `if` statements inside a loop. Consider this program fragment:

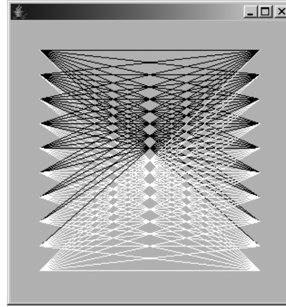
```
for (int left = 1; left <= 10; left = left + 1)
{ if (left <= 5)
  { g2.setColor(Color.WHITE);
  } else
  { g2.setColor(Color.BLACK);
  }

  for (int right = 1; right <= 10; right = right + 1)
  { g.drawLine(1, left, 10, right);
  }
}
```

The `if` statements test the loop control variable against an integer, just as we tested the result of an integer query such as `getAvenue()` against an integer. The drawing color is set based on the test's outcome. The result is shown in Figure 5-21a, in which the first five sprays are white and the last five are black. The background is set to a darker shade of gray to show the white lines more effectively.



a) Color according to the value of `left`



b) Color according to the value of `left + right`

(figure 5-21)

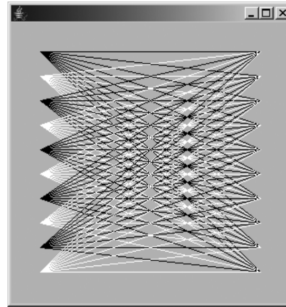
Sprays of lines with varying colors, colored according to the sum of `inner` *and* `outer`

One more possibility is to perform a slightly more complex test for the color. It is possible to compare two integer expressions in the `if` statement's test. In the following example, the `if` statement is moved into the inner loop. It makes the line white if the sum of the values contained in `left` and `right` is greater than 10, and black otherwise. The result is shown in Figure 5-21b.

```
for (int left = 1; left <= 10; left = left + 1)
{ for (int right = 1; right <= 10; right = right + 1)
  { if (left + right > 10)
    { g2.setColor(Color.WHITE);
    } else
    { g2.setColor(Color.BLACK);
    }

    g.drawLine(1, left, 10, right);
  }
}
```

We can also use Java's **remainder operator** (`%`) in the test. The remainder operator gives the remainder when one number is divided into another. For example, `4 % 2` is 0 because 2 goes into 4 an even number of times. On the other hand, `5 % 2` is 1 because when 5 is divided by 2, 1 is left over. This mathematical relationship gives us an easy test for whether a number is even or odd. For example, if `left % 2` is 0, then the value contained in `left` is even. If `left % 2` is 1, then the value contained in `left` is odd. In the following program fragment, this fact is used to color alternating sprays differently. The result is shown in Figure 5-22.



(figure 5-22)

Alternating the color of each spray

```
for (int left = 1; left <= 10; left = left + 1)
{ if (left % 2 == 0)
  { g2.setColor(Color.WHITE);
  } else
  { g2.setColor(Color.BLACK);
  }

  for (int right = 1; right <= 10; right = right + 1)
  { g.drawLine(1, left, 10, right);
  }
}
```

As you can see, selection and repetition statements such as `if`, `while`, and `for` can be combined in many ways.

5.8 Patterns

5.8.1 The Loop-and-a-Half Pattern

Name: Loop-and-a-Half

Context: A loop is used for a variation of the fence-post problem; that is, some of the repeated actions (the "fence-post actions") must be performed one more time than the other repeated actions (the "fence-section actions").

Solution: There are two standard solutions. The first repeats part of the code either before or after the loop, as appropriate. Templates for two variants follow:

```
«fencePost actions»
while («booleanExpression»)
{ «fenceSection actions»
  «fencePost actions»
}
```

```
while («booleanExpression»)
{ «fencePost actions»
  «fenceSection actions»
}
«fencePost actions»
```

The second solution avoids the repeated code with a `while-true` loop, as shown in the following template:

```
while (true)
{ «fencePost actions»
  if («booleanExpression»)  {  break;  }
  «fenceSection actions»
}
```

Consequences: The *«fencePost actions»* are executed one more time than the *«fenceSection actions»*. The *«fencePost actions»* are always executed at least once.

Related Pattern: This pattern is a variation of the Zero or More Times pattern. That pattern is used when all of the repeated steps are executed an equal number of times.

5.8.2 The Temporary Variable Pattern

Name: Temporary Variable

Context: You need to store a value that is used in the task being performed rather than as an attribute of the object. The value is only used in one method and perhaps in the methods it invokes—for example, a variable to control a loop, to store a temporary result for use in later calculations, or to accumulate a value to return to a client.

Solution: Use a temporary variable. For example, a `Robot` might be extended with the following method, which uses a temporary variable, `numWalls`:

```
public int numBlockedDirections()
{ int numWalls = 0;
  for(int turns = 0; turns < 4; turns = turns + 1)
  { if (!this.frontIsClear())
    { numWalls = numWalls + 1;
    }
    this.turnLeft();
  }
  return numWalls;
}
```

The general form for declaring a temporary variable is:

```
«type» «name» = «initialValue»;
```

where *«type»* is the type of value stored, such as `int`, `double`, or even the name of a class; *«name»* is the name used for the variable; and *«initialValue»* is the first value used for the variable. The initial value is optional; however, it must be assigned before the variable is first used, and it is good practice to initialize the variable when it is declared.

Consequences: A variable is declared that may only be used within the smallest enclosing block of code. Because it is only used locally, the reader's burden of remembering the name and purpose of the variable is significantly reduced, speeding the comprehension of the program and reducing errors.

Related Patterns: This pattern always occurs within an instance of a method pattern, such as the Helper Method, Query, or Parameterized Method patterns.

5.8.3 The Counting Pattern

Name: Counting

Context: You need to count a number of events, such as the number of times a `Thing` is picked up, the number of moves a robot makes, or the number of times a test returns `true`.

Solution: Increment a temporary variable each time the event occurs. Initialize the variable to zero before counting begins.

```
int «counter» = 0;
while («booleanExpression»)
{ «statements»
  «counter» = «counter» + 1;
}
```

Variations of this template may increment *«counter»* only if a certain test is met or may use a different looping strategy.

Consequences: *«counter»* will record the number of events that have occurred since it was initialized.

Related Patterns:

- This pattern uses a loop, typically the Zero or More Times pattern and the Temporary Variable pattern.
- This pattern is often placed in an instance of the Query pattern.

5.8.4 The Query Pattern

Name: Query

Context: A calculation that yields a single value is required. This pattern is particularly applicable if:

- the calculation involves a number of steps
- the calculation is complicated
- program readability is improved by giving the calculation a name
- the calcualtion is used more than once in the program

Solution: Write a method with a return value of the required type that uses a `return` statement to identify the calculation's answer. In general:

```
«accessModifier» «returnType» «queryName»(«optParameters»)
{ «optionalStatements»
  «returnType» answer = «expression»;
  «optionalStatements»
  return answer;
}
```

An example is `countThingsHere`, shown in Listing 5-2. Another example is to calculate the distance from a given street, as follows:

```
private int distanceFromStreet(int targetStr)
{ int answer;
  if (this.getStreet() > targetStr)
  { answer = this.getStreet() - targetStr;
  } else
  { answer = targetStr - this.getStreet();
  }
  return answer;
}
```

Queries should avoid side effects.

Consequences: Queries make code easier to understand because they name a calculation. They also make the calculation easier to reuse.

Related Patterns:

- The Query pattern is a specialization of the method creation patterns, such as the Parameterless Command, Helper Method, and Parameterized Method patterns.
- The Simple Predicate and Predicate patterns are specializations of this pattern.

5.8.5 The Predicate Pattern

Name: Predicate

Context: You are using a Boolean expression that is not as easy to read or understand as is desired, or a test is needed that can't be written as a Boolean expression because it requires extra processing.

Solution: Use the Query pattern where the *«returnType»* is `boolean`. Such a query is called a predicate. The predicate may have parameters to make it more flexible.

Consequences: The processing required for the test is encapsulated in a reusable method. With appropriate naming, the code using the predicate is more readable.

Related Patterns:

- The Predicate pattern is a specialization of the Query pattern.
- The Predicate pattern is often used to define predicates used in the Once or Not At All, Zero or More Times, and Either This or That patterns, among others.
- The Simple Predicate pattern is a simplified version of this pattern that does not use a temporary variable or the optional statements.

5.8.6 The Cascading-`if` Pattern

Name: Cascading-`if`

Context: You have a situation in which exactly one of several groups of statements should be executed based on a sequence of tests.

Solution: Order the tests from the most specific test to the most general test, pairing each test with the appropriate group of actions. Format the tests and actions to emphasize the pairings:

```
if («test1»)
{ «statementGroup1»
} else if («test2»)
{ «statementGroup2»
...
} else if («testN»)
{ «statementGroupN»
} else
{ «defaultStatements»
}
```

Consequences: The tests are executed in order from 1 to N. The first one that returns `true` will cause the associated statement group to be executed once. The final `else` and *«defaultStatements»* are optional. They will be executed if none of the tests return `true`.

Related Patterns:

- If there is only one test and one group of statements, this pattern becomes the Once or Not At All pattern. Similarly, if there is only one test but two groups of statements, this pattern becomes the Either This or That pattern.
- The `switch` statement, while not included as a pattern, solves similar kinds of problems when the decision of which group of statements to execute is based on a single value.

5.8.7 The Counted Loop Pattern

Name: Counted Loop

Context: You have a group of statements that must be executed a specific number of times, a number that is known when the loop begins execution.

Solution: Use a `for` statement, as in the following example:

```
// move to avenue 0; assumes the robot is on an avenue west of 0 and is facing West
int howFar = this.getAvenue();
for (int i=0; i<howFar; i = i + 1)
{ this.move();
}
```

The general form for this pattern is shown in Section 5.5.1. There is an equivalent form of the `while` statement, also shown in that section.

Consequences: The body of the `for` statement is executed zero or more times, depending on the specifics of the loop.

Related Pattern: The Counted Loop pattern is a specialization of the Zero or More Times pattern.

5.9 Summary and Concept Map

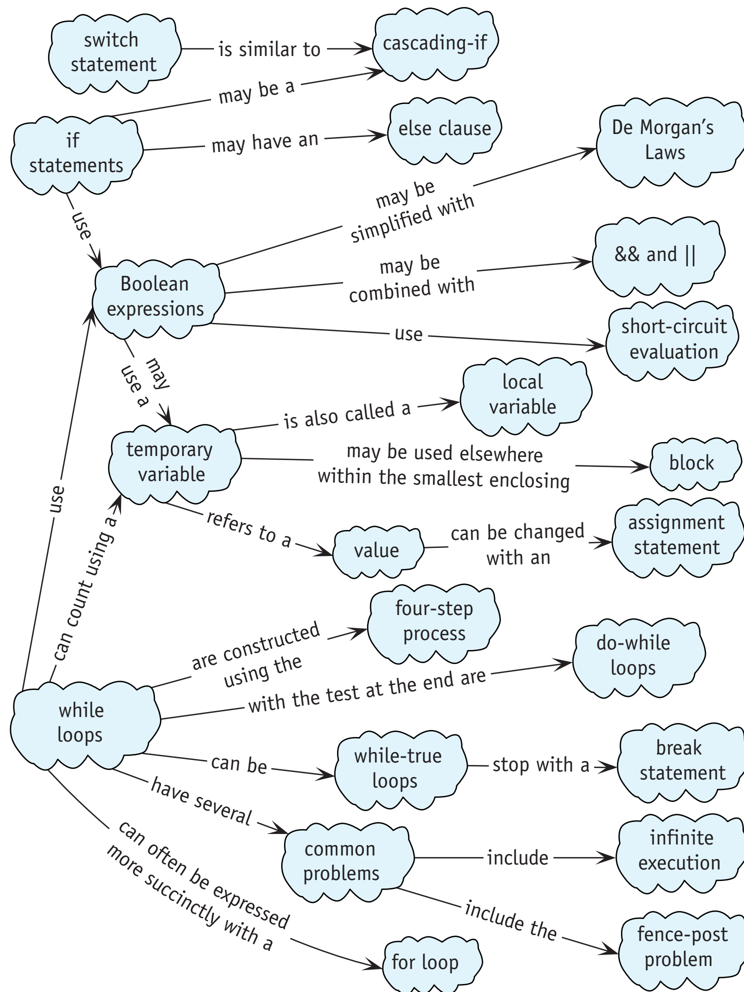
A `while` statement is used to execute a group of statements zero or more times. Writing such a loop correctly can be made easier by following a formal four-step process. There are also specialized forms of the `while` loop. For example, when it should execute a known number of times, a `for` statement is the preferred alternative. Loops that always execute at least once may use the `do-while` statement while the `while-true` variation is particularly useful for solving fence-post problems using the Loop-and-a-Half pattern.

Statements may be nested, for example, by putting an `if` statement within a `while` or `for` statement. When only one of several groups of statements should be executed, a particular pattern of nesting `if` statements called a cascading-`if` is useful. Complicated nesting structures should be avoided by using helper methods.

Temporary or local variables are used to remember a value for later use in the same method. They can simplify many methods and enable techniques such as the Counting pattern. Temporary variables are also useful to remember a value to be returned from a query. The value a temporary variable refers to can be changed with an assignment statement in which the expression on the right side is evaluated and the resulting value is assigned to the variable on the left side.

Boolean expressions may be combined using "and" (&&) and "or" (||). As such expressions become complicated, encapsulating them in a predicate and simplifying them, perhaps with De Morgan's Laws, can make the program easier to understand.

The statements discussed in this chapter and the previous chapter can significantly increase the complexity of our programs, making appropriate style important. Writing helper methods identified with stepwise refinement, using positively stated tests, and visually structuring the code are all important techniques.



5.10 Problem Set

Written Exercises

5.1 Show the four steps used to derive a `while` loop for the following situations:

a. A robot must pick up all the things on the intersection it occupies.

b. A robot must pick up the same number of things as it has in its backpack.

c. A robot facing east must move until it is on the nearest street that is divisible by eight. (*Hint*: Use the remainder operator (`%`) discussed in the code used for Figure 5-22.)

d. A robot moves until it arrives at an intersection with a thing and a wall on the right edge.

e. A robot moves between consecutive intersections picking up one thing from each intersection, beginning with the one it is on. If there is still a thing on the intersection after it has picked one up, the robot stops.

f. A variable, `maxToPick`, holds the maximum number of things a robot should pick up. It picks up that many things from its current intersection unless there aren't enough things present. In that case, it picks up as many as it can.

5.2 The `pickThingsToWall` method, shown in Listing 5-4 and illustrated in Figure 5-6, instructs a robot to move to a wall, picking one `Thing` from each intersection that has one. Describe the changes required to make the robot pick up an entire pile of things from those intersections that have them.

5.3 For each subproblem, write a predicate that returns the same value as the given Boolean expression. That is, you could use your predicate instead of the Boolean expression in a program with no difference in the overall behavior of the program. Do not use `&&`, `||`, or `!` inside the predicate. (*Hint*: Use `if` and `if-else` statements with a temporary variable and possibly helper methods.)

a. `this.getAvenue() > 5 && this.getAvenue() < 10`

b. `this.countThingsInBackpack() > 10 && !this.frontIsClear()`

c.
```
(this.getDirection() == Direction.NORTH ||
 this.getDirection() == Direction.SOUTH) &&
 this.frontIsClear()
```

5.4 For each subproblem, draw an oval diagram for the given expression, assuming the robot is in the described situation.

a. `this.getAvenue() > 5 && this.getAvenue() < 10`
(the robot is on avenue 5)

b. `this.countThingsInBackpack() > 10 && !this.frontIsClear()`
(the robot has 12 things in its backpack and is facing a wall)

c.
```
(this.getDirection() == Direction.NORTH ||
 this.getDirection() == Direction.SOUTH) &&
 this.frontIsClear()
```
(the robot is facing north and its front is clear)

Programming Exercises

5.5 Use the cascading-`if` statement to write a method named `faceNorth` that always turns a robot to face north.

5.6 A `HomingRobot`'s "home" is at 4th Street and 3rd Avenue in a city that has no obstructions, such as walls. `HomingRobot` contains a method named `goHome`, which returns the robot to (4, 3) no matter where the robot is in the city. `goHome` is written as follows:

```
public void goHome()
{ while (!this.atHome())
  { this.faceHome();
    this.move();
  }
}
```

a. Write the predicate `atHome`.

b. Use a cascading-`if` to write `faceHome`.

5.7 Consider again the situation shown in Figure 5-9 in which a robot should stop at a thing or a wall, whichever comes first.

a. Solve the problem using a `while-true` loop with one `break` statement.

b. Solve the problem using a `while-true` loop with two `break` statements.

5.8 Consider again the problem of shifting things from one intersection to another, as illustrated in Figure 5-4. Solve the problem using a `while-true` loop.

5.9 Use techniques presented in Section 5.6.2 to improve the following code fragments. If they can't be improved, explain why.

a.

```
if (this.isFacingNorth())
{ this.turnAround();
  this.pickThing();
} else
{ this.turnAround();
  this.putThing();
}
```

b.

```
if (this.getStreet() != 5)
{ this.turnleft();
} else
{ this.turnRight();

}
```

c.

```
if (this.canPickThing())
{ this.move();
  this.turnLeft();
} else
{ this.move();
  this.turnRight();
}
```

d.

```
if (count != 5 &&
    !this.frontIsClear())
{ this.turnRight();
  count = count + 1;
} else
{ this.turnLeft();
  count = count + 1;
}
```

e.

```
int n = this.thingsHere();
if (n == 0)
{ this.turnLeft();
  this.move();
} else if (n == 1)
{ this.turnRight();
  this.move();
} else if (n == 2)
{ this.turnAround();
  this.move();
} else
{ this.move();
}
```

f.

```
if (this.frontIsClear())
{ if (this.canPickThing())
  { this.pickThing();
    this.move();
  } else
  { this.move();
  }
} else
{ this.turnLeft();
  this.move();
}
```

5.10 Assume that a `Prospector` robot is on an intersection with either one or two things. Write a new method named `followTrail` that commands the robot to face north if it is on an intersection with one thing and to face south if it is on an intersection with two things. The robot must leave the same number of things on the intersection as it found originally.

5.11 Write a predicate that returns `true` if and only if a robot is completely surrounded by walls and unable to move in any direction. Of course, the predicate should not have side effects.

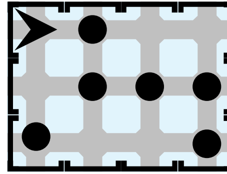
5.12 Implement the pile-shifting robot described in Section 5.1.2.

Programming Projects

5.13 `karel` is in a completely enclosed rectangular room that has, unfortunately, litter strewn all over it (see Figure 5-23). Create a new class of robot that can pick up the litter. The size of the room is unknown and the amount of litter on each intersection is also unknown. However, its top-left corner is always on intersection (1, 1), and `karel` always starts there, facing east. `karel` should return to its starting position when its task is complete. Make use of stepwise refinement and helper methods. Create files representing different rooms to test your program.

(figure 5-23)

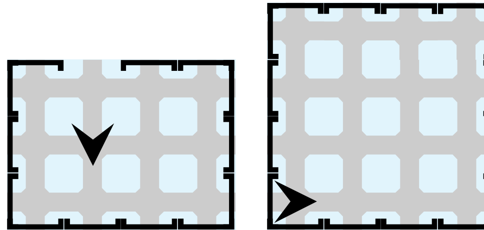
Littered room



5.14 Write a new `Robot` class, `Houdini`, that includes a method named `escapeRoom`. It will cause the robot to search for an exit to a rectangular room—a break in the wall. Such an exit always exists and is never in a corner. The robot may start anywhere in the room, but it will not be facing the exit. When the exit is found, the robot will move through the exit and then stop. See Figure 5-24 for two of many possible initial situations.

(figure 5-24)

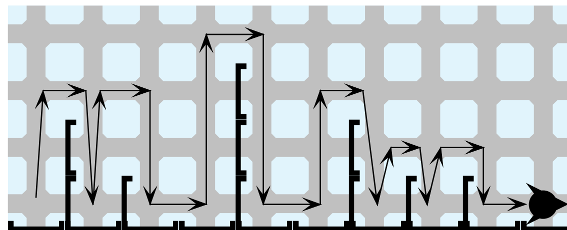
Two possible rooms to escape



5.15 Program a robot to run a mile-long steeplechase. The steeplechase course is similar to the hurdle race (see Section 4.4.1), but here, the barriers can be one, two, or three walls high. One sample situation is shown in Figure 5-25. The robot begins the race on the lower-left corner facing east and follows the path shown. Call the class of this new robot `SteepleChaser`. It should have `Racer` as a parent class (see Section 4.4.1). Override appropriate statements of `Racer` to implement the new behavior.

(figure 5-25)

One possible steeplechase situation



5.16 Extend the `RobotSE` class to create a `MazeWalker`. `MazeWalker` has a single public method, `followWallRight`. Assume that when it executes, the robot has a wall directly to its right. By calling this method repeatedly, the robot will eventually find its way out of a maze.

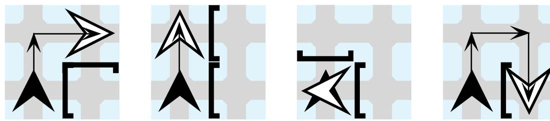
Study the online documentation for the `MazeCity` class to learn how to construct a city with a maze in it. Place your `MazeWalker` at (0, 0) facing south and a `Thing` someplace within the city for it to find. Call `followWallRight` repeatedly until the thing is found.

a. One strategy for `followWallRight` involves four different position changes, as shown in Figure 5-26. The dark robot signifies the initial position and the light robot signifies its position after `followWallRight` is invoked.

b. Another strategy for `followWallRight` is for the robot to make exactly one move each time the method is called.

c. Develop a solution that minimizes the number of "useless" turns the robot makes to determine if its right or left side is blocked.

Comments: Option (a) is easy because there are hints in Figure 5-26, but it's hard to get the robot to stop at the right place. Option (b) is hard because it has no hints, but it's easy to get the robot to stop at the right place.



(figure 5-26)

Movements of a `MazeWalker` *robot*

5.17 Implement a class named `TrailBot` that extends `RobotSE` and contains a single public method, `followTrail`. A `TrailBot` follows a trail to a destination. Trails begin at (0, 0) with the robot facing south. Trails consist of various signs that indicate how to continue following the trail. The robot must leave the trail signs as they were found. One way to test this is to have two robots follow the same trail. The robots may or may not start with `Things` in their backpacks.

a. The trail signs consist of piles of one or more things. The number of `Things` in the pile instruct the robot how far to move forward. After moving that distance, the robot may find another trail sign (a pile of `Things`). If so, the number present instructs the robot how far to go to find the next trail sign. Finding a pile and moving a distance equal to its size continues until the robot arrives at an empty intersection (the end of the trail). There may be things between the piles that instruct the robot how far to go. If so, they should be ignored.

b. The trail signs are as follows:

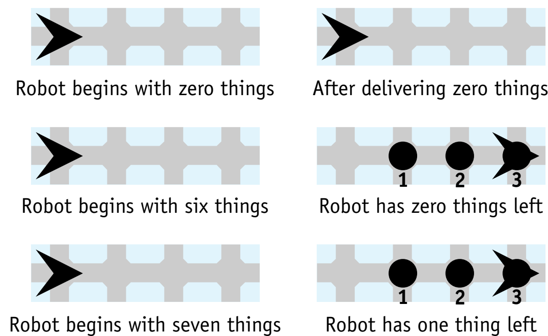
- A `Wall` and one `Thing`: end of the trail
- One `Thing`: move one intersection to the right
- A `Wall`: move one intersection to the left
- Empty intersection: move one intersection forward.

c. Design your own set of trail signs and create a robot to follow it.

5.18 `karel` is an instance of `DeliveryBot` and has a unique delivery task. It starts with some number of `Things` in its backpack. When its `deliverThings` method is called, it begins to place `Things` on consecutive intersections. On the first intersection it places one thing. On the next intersection it places two things, and on the next intersection, three things. Each intersection receives one more thing than the previous intersection. Each intersection receives its full allotment of things or none at all. Figure 5-27 shows several pairs of sample initial and final situations.

(figure 5-27)

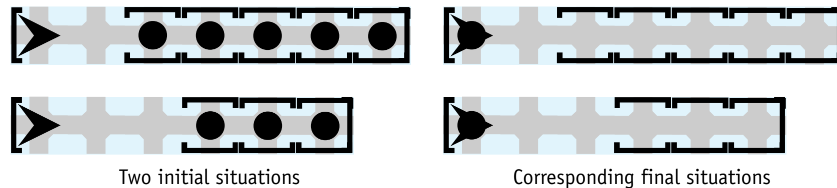
Pairs of initial and final situations for a `DeliveryBot`



5.19 An instance of `ClearTunnelBot` is facing a tunnel that has at least one `Thing` on each intersection. When given the `clearTunnel` command, the robot should remove all of the `Things`, placing them as shown in the final situation. The robot may carry at most one `Thing` at a time and may not make any trips back to the tunnel once all the `Things` have been removed. Figure 5-28 shows two typical situations and their corresponding final situations. The robot will always start with a wall behind it, marking where the things should be placed. The distance to the tunnel and the length of the tunnel may vary.

(figure 5-28)

Tunnel-clearing situation and its corresponding final situation

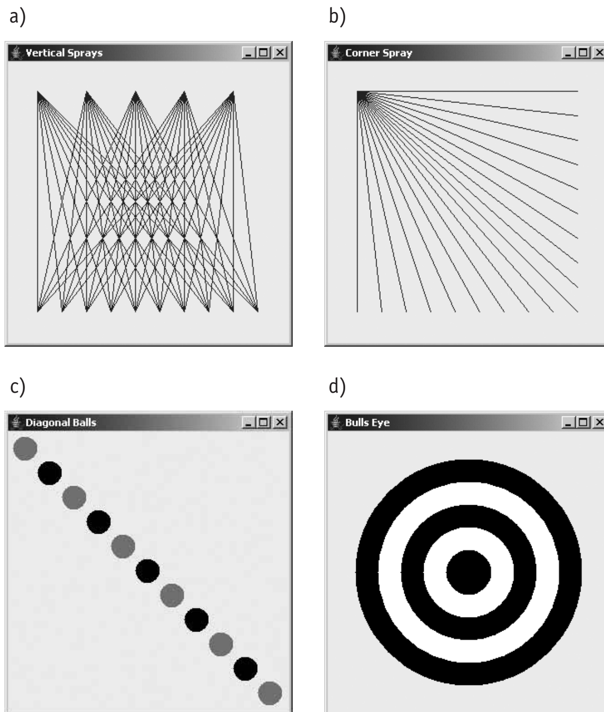


5.20 Modify the program in Listing 5-7 as follows:

a. Draw sprays of lines, starting at the top of the image and extending down, as shown in Figure 5-29a.

b. Draw lines from the upper-left corner to evenly spaced points on the bottom and right edges, as shown in Figure 5-29b. *Hint*: You only need one loop, but each iteration draws two lines.

c. Draw a line of circles that alternate in color, as shown in Figure 5-29c.
d. Draw a bull's eye, as shown in Figure 5-29d. You will need a single loop. Use the loop counter to specify the top left corner of the circles and a second variable for the size of the circles.
e. Fill the entire component with circles, as shown in Figure 5-29e.
f. Fill the entire component with circles, as shown in Figure 5-29f. Define a predicate returning `true` if the given row and column are part of the cross, and false otherwise.
g. Fill the entire component with circles, as shown in Figure 5-29g. Define a predicate returning true if the given row and column are part of the cross, and false otherwise.
h. Create a checkerboard, as shown in Figure 5-29h.



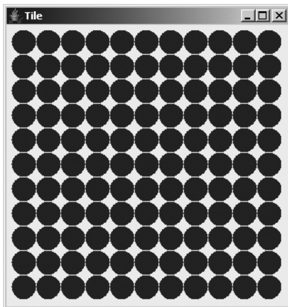
(figure 5-29)

Various fine pieces of art

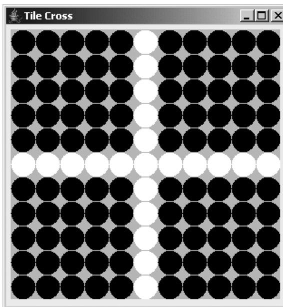
(figure 5-29) *continued*

Various fine pieces of art

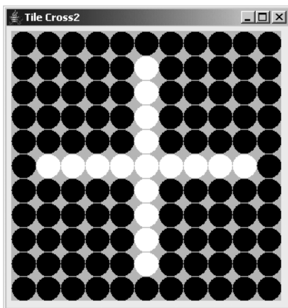
e)



f)



g)



h)