Chapter 4 Making Decisions

Chapter Objectives

After studying this chapter, you should be able to:

- Use an `if` statement to perform an action once or not at all.
- Use a `while` statement to perform an action zero or more times.
- Use an `if-else` statement to perform either one action or another action.
- Describe what conditions can be tested and how to write new tests.
- Write a method, called a predicate, that can be used in the test of an `if` or `while` statement.
- Use parameters to communicate values from the client to be used in the execution of a method.
- Use a `while` statement to perform an action a specified number of times.

In the preceding chapters, a robot's exact initial situation was known at the start of a task. When we wrote our programs, this information allowed robots to find things and avoid running into walls. However, these programs worked only in their specific initial situations. If a robot tried to execute one of these programs in a slightly different initial situation, the robot would almost certainly fail to perform the task.

To address this situation, a robot must make decisions about what to do next. Should it move or should it pick something up? In this chapter we will learn about programming language statements that test the program's current state and choose the next statement to execute based on what they find. One form of this capability is the `if` statement: *If* something is true, then execute a group of statements. *If* it is not true, then skip the group of statements. Another form of this capability is the `while` statement: *while* something is true, execute a group of statements.

4.1 Understanding Two Kinds of Decisions

So far, our programs have been composed of a sequence of statements executed in order. These statements have included creating new objects (the Object Instantiation pattern) and invoking their services (the Command Invocation pattern). The only deviation we've seen from this sequential order is in defining our own commands or methods. In that case, whenever one method includes a statement invoking another method, all the statements in the called method are executed in order before execution moves on to the next statement in the calling method.

The `if` and `while` statements are different. As the program is running, they can ask a question. Based on the answer, they choose the next statement or group of statements to execute. In a robot program, the question asked might be, "Is the robot's front blocked by a wall?" or "Is there something on this intersection the robot can pick up?" In the concert hall program from Chapter 1, questions asked by an `if` or `while` statement might include "Is the ticket for seat 22H still available?" or "Have all of the sold tickets been processed yet?"

KEY IDEA

`if` and `while` statements choose the next statement to execute by asking a yes/no question.

All of these questions have "yes" or "no" answers. In fact, `if` and `while` statements can only ask yes/no questions. Java uses the keyword `true` for "yes" and `false` for "no." These keywords represent Boolean values, just like the numbers `0` and `23` represent integer values.

KEY IDEA

`true` means "yes" and `false` means "no."

When the simplest form of an `if` statement asks a question and the answer is `true`, it executes a group of statements once and then continues with the rest of the program. If the answer to the question is `false`, that group of statements is not executed.

When a `while` statement asks a question and the answer is `true`, it executes a group of statements (just like the `if` statement). However, instead of continuing with the rest of the program, the `while` statement asks the question again. If the answer is still `true`, that same group of statements is executed again. This continues until the answer to the question is `false`.

The `if` statement's question is "Should I execute this group of statements *once*?" The `while` statement's question is "Should I execute this group of statements *again*?" This ability to ask a question and to respond differently based on the answer liberates our programs from always executing the same sequence of statements in exactly the order given. For example, these two statements will allow us to generalize the `Harvester` class shown in Listing 3-3 in the following ways:

KEY IDEA

`if` statements execute code once or not at all. `while` statements might execute the statements repeatedly.

- Harvest any number of things from the intersection.
- Have a single `goToNextRow` method that works at both ends of the row. The current solution has one method for the east end of the row and another method for the west end.
- Harvest fields of varying sizes.

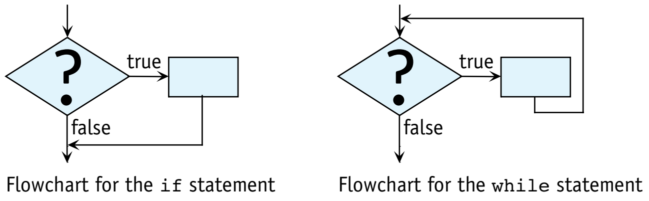
4.1.1 Flowcharts for `if` and `while` Statements

One way to illustrate the flow of control through the `if` and `while` statements is with a flowchart, as shown in Figure 4-1. The diamond represents the question that is asked. The box represents the statements that are optionally executed. The arrows show what the computer does next.

By tracing the arrows in the flowchart for `if`, you can easily verify that the statements in the box are executed once or not at all. Tracing the arrows for the `while` statement, however, shows that you can reach the statements in the box over and over again—or that they might not be executed at all.

(figure 4-1)

Flowcharts for the `if` and `while` statements


?
true
false
Flowchart for the if statement
?
true
false
Flowchart for the while statement

KEY IDEA

Decide between `if` and `while` by asking how many times the code should execute.

The key question you should ask when deciding whether to use an `if` statement or a `while` statement is "How many times should this code execute?" If the answer is once or not at all, choose the `if` statement. If the answer is zero or more times, then choose the `while` statement.

4.1.2 Examining an `if` Statement

Suppose that a robot, `karel`, is in a city that has walls. If `karel`'s path is clear, it should move and then turn left. Otherwise, `karel` should just turn left.

You can use an `if` statement to have `karel` make this decision. The `if` statement's question is "Is `karel`'s front clear of obstructions?" If the answer is `true` (yes), `karel` should move forward and then turn left. If the answer is `false` (no), `karel` should skip the move instruction and just turn left. This program fragment[1] is written like this:

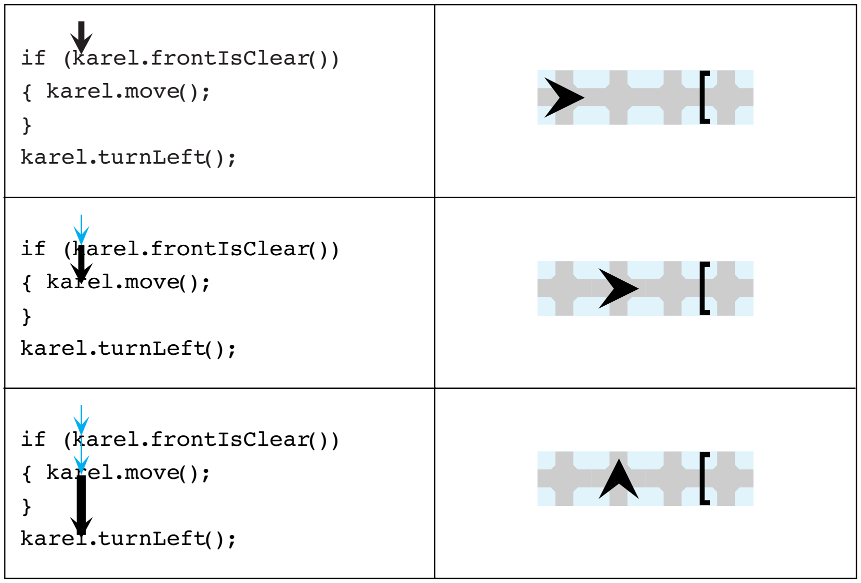
PATTERN

Once or Not at All

```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

[1] To conserve space, we will often demonstrate a programming idea without writing a complete program or even a complete method. Instead, we will write only the necessary statements, which are called a **program fragment**.

Consider two different initial situations. In Figure 4-2, the answer to the `if` statement's question is "Yes, `karel`'s front is clear of obstructions." As a result, `karel` performs the test, moves, and then turns left. These three actions are shown in the figure, where the heavy arrows show the statements that are executed to produce the situation shown on the right.



```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

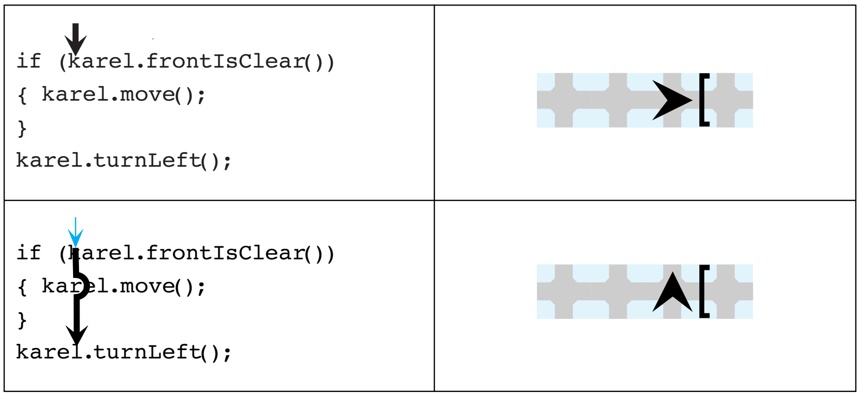
(figure 4-2)

Execution of an `if` statement when the robot's front is initially clear of obstructions

Suppose `karel` starts in the situation shown in Figure 4-3. Then the answer to the `if` statement's question is "No, `karel`'s front is *not* clear of obstructions" and the statement instructing `karel` to move is *not* executed. `karel` does not move, although it does turn left because the `turnLeft` command is outside the group of statements controlled by the `if` statement.

KEY IDEA

The `if` statement causes the robot to behave differently, depending on its situation.



```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

(figure 4-3)

Execution of an `if` statement when the robot's front is initially obstructed

Consider the code without the `if` statement:

```
karel.move();
karel.turnLeft();
```

In the first situation (shown in Figure 4-2), the result would be the same. However in the second situation, `karel` would crash into the wall and break.

Use an `if` statement when you want statements to execute once or not at all.

4.1.3 Examining a `while` Statement

Let's now consider a similar situation but control the `move` instruction with a `while` statement:

PATTERN

Zero or More Times

```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

KEY IDEA

The `while` statement repeatedly asks a question and performs an action until the answer is "no."

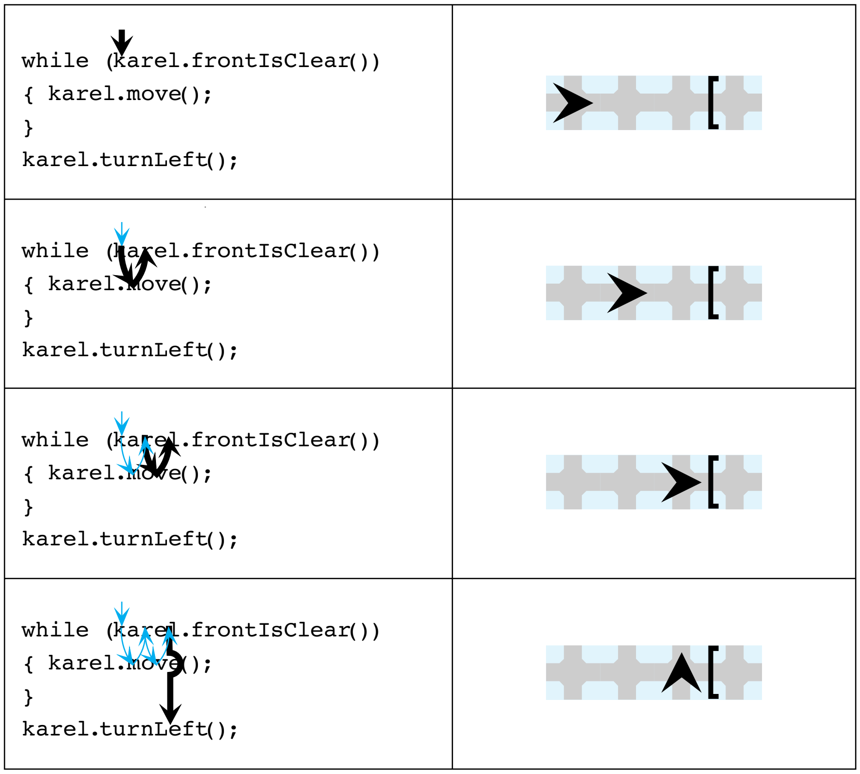
Recall that a `while` statement also asks a question. If the answer is `true`, the statements inside the braces are executed and then the question is asked again. This continues until the answer to the question is `false`. In the preceding code fragment, the question is "Is `karel`'s front clear of obstructions?"

Let's again consider `karel` in different initial situations. In Figure 4-4, `karel`'s front is clear and the answer to the `while` statement's question is "it is `true`, `karel`'s front is clear." `karel` moves and asks the question again—until the answer to the question is finally `false`. The heavy arrows in the code show the statements that are executed to reach the situation shown to the right of the code.

In this example, `karel` moves as many times as necessary to reach the wall. Then it turns. In the situation shown in Figure 4-4, the wall happens to be only two intersections away. It could be 20 or 2 million intersections away—`karel` would still move to the wall and then turn left with those same four lines of code.

If `karel` starts in a situation where its front is blocked, the answer to the question is immediately `false` and the move does not occur. Execution continues with the `turnLeft` instruction after the `while` statement. This situation is illustrated in Figure 4-5. Notice the similarities to the last two illustrations in Figure 4-4.

The `while` statement's test is always `false` after the statement finishes executing because the loop continues until the test becomes `false`. In fact, if nothing inside the `while` statement can make the test `false`, the statement will execute indefinitely.



```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

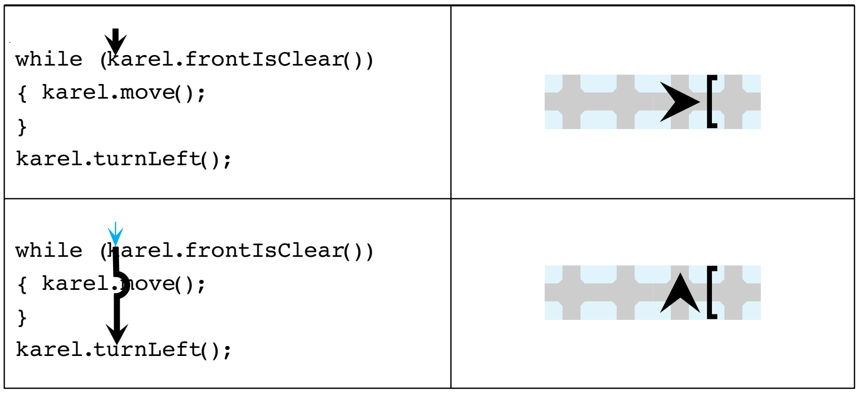
```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

(figure 4-4)

Illustrating the execution of a `while` *statement when the robot's front is initially clear of obstructions*



```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

(figure 4-5)

Illustrating the execution of a `while` *statement when the robot's front is initially obstructed*

The ability to go to a wall might be generally useful. The following method, inserted into a class extending `Robot`, provides such a service:

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

4.1.4 The General Forms of the `if` and `while` Statements

The general form of a statement marks the parts that can change, depending on the needs of the situation, leaving the parts that are always the same clearly identified.

The General Form of an `if` Statement

The `if` statement has the following general form:

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

The reserved word `if` signals the reader of the program that an `if` statement is present. The braces (`{` and `}`) enclose a list of one or more statements, *`«list of statements»`*. These statements are known as the **then-clause**. The statements in the then-clause are indented to emphasize that *`«list of statements»`* is a component of the `if` statement. Note that we do not follow the right brace of an `if` statement with a semicolon.

KEY IDEA

Boolean expressions ask true/false questions.

The *`«test»`* is a **Boolean expression** such as a query that controls whether the statements in the then-clause are executed. A Boolean expression always asks a question that has either `true` or `false` as an answer.

The General Form of a `while` Statement

The general form of the `while` statement is:

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

The reserved word `while` starts this statement. Like the `if` statement, the *`«test»`* is enclosed by parentheses and the *`«list of statements»`* is enclosed by braces[2]. The

[2] If the list of statements has only one statement, the braces can be omitted. More about this in Section 5.6.3.

list of statements is called the **body** of the statement. The Boolean expressions that can replace *«*`test`*»* are the same ones used in the `if` statements.

A statement that repeats an action, like a `while` statement, is often called a **loop**.

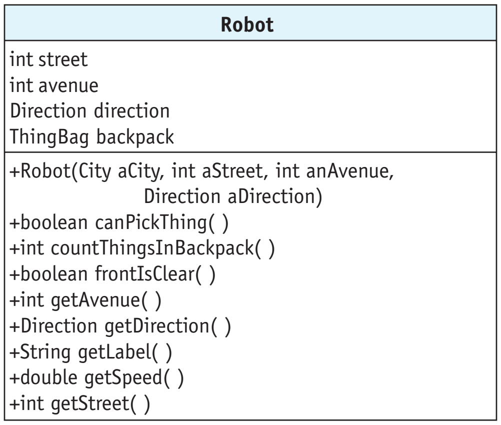
The `if` and `while` statements have similar **syntax**. That is, their structure, or the way they look, is similar. On the other hand, they have different **semantics**. That is, the way they behave is different. The `if` statement decides whether to execute a list of statements or to skip over them. The `while` statement decides how many times to execute a list of statements.

4.2 Questions Robots Can Ask

The `if` and `while` statements both ask a question to discover something about the current state of the program. In the previous section the question was whether the front of the robot was clear of obstructions. In this section we'll learn about other questions a robot can ask. The answers, of course, can be used to control the robot's behavior.

4.2.1 Built-In Queries

In Chapter 1, we briefly mentioned that one kind of service objects can provide is a query—a service that answers a question. Robots offer queries that answer questions such as "Which avenue are you on?", "Which direction are you facing?", "Can you pick up a `Thing` from the intersection you are currently on?", and "Is your front clear of obstructions?" The following class diagram, displayed in Figure 4-6, shows many of the queries robots can answer.



```
Robot
int street
int avenue
Direction direction
ThingBag backpack
+Robot(City aCity, int aStreet, int anAvenue,
             Direction aDirection)
+boolean canPickThing( )
+int countThingsInBackpack( )
+boolean frontIsClear( )
+int getAvenue( )
+Direction getDirection( )
+String getLabel( )
+double getSpeed( )
+int getStreet( )
```

(figure 4-6)

Class diagram showing many of the queries a robot can answer

KEY IDEA

Predicates are methods that return either `true` *or* `false`.

Each of the queries indicates what kind of answer it returns. `getAvenue`, for example, returns an integer value (abbreviated `int`) such as `1` for 1st Avenue or `9` for 9th Avenue. `canPickThing`, on the other hand, returns a `boolean`[3] value. If the robot is on the same intersection as a `Thing` it can pick up, `canPickThing` returns `true`; otherwise, it returns `false`. Queries that return a `boolean` answer are called **predicates**. The `frontIsClear` service described in the previous section is a predicate.

None of these queries change the state of the robot. The robot doesn't change in any way; it merely reports a piece of information about itself or its environment. This information is used in **expressions**. Expressions may be used in many ways, such as controlling `if` and `while` statements, passed as a parameter to a method, or saved in a variable. In this chapter, we will focus almost exclusively on expressions used to control `if` and `while` statements.

4.2.2 Negating Predicates

Sometimes we want a robot to do something when a test is *not* true, as in the following pseudocode:

```
if (karel cannot pick up a thing)
{ put a thing down
}
```

The `Robot` class does not provide a predicate for testing if the robot *cannot* pick up a `Thing`, only if it *can*.

KEY IDEA

Give a boolean expression the opposite value with `"!"`

PATTERN

Once or Not at All

Fortunately, any Boolean expression may be **negated**, or given the opposite value, by using the **logical negation operator**, "`!`". In English, this is usually written and pronounced as "not". The negation operator is placed immediately before the Boolean expression that is to be negated. Thus, the previous pseudocode could be coded as follows:

```
if (!karel.canPickThing())
{ karel.putThing();
}
```

Negation is our first exploration of **evaluating** expressions. You already have experience evaluating expressions from studying arithmetic. When you figure out that 5 + 3 * 2 is the same as 5 + 6 or 11, you are evaluating an arithmetic expression. The expression often includes an unknown, such as $5 + x * 2$. When you know the value of x, you can substitute it into the expression before evaluating it. For example, if x has the value 4, then the expression $5 + x * 2$ is the same as 5 + 4 * 2 or 13.

[3] Boolean values are named after George Boole, one of the early developers of logic.

Evaluating a Boolean expression, an expression that uses values of `true` and `false`, is similar to evaluating arithmetic expressions. The expression `!karel.canPickThing()` involves an unknown (`karel.canPickThing()`), similar to *x* in the arithmetic expression. Suppose the unknown has the value `true` (that is, `karel` is on the same intersection as a `Thing` it can pick up). Then the expression evaluates to `!true` ("not `true`") which is the same as `false`.

LOOKING AHEAD

In Section 5.4.1, we will look at combining expressions with "and" and "or", much like "+" and "" combine arithmetic expressions.*

4.2.3 Testing Integer Queries

The `if` and `while` statements always ask true-or-false questions. "Should I execute this code, `true` or `false`?" This approach works well for queries that return a `boolean` value, but how can we use queries that return integers? The solution is to compare the query's answer to another integer. For example, we could use the following code to ask if the robot is on 1st Street:

```
if (karel.getStreet() == 1)
{  // what to do if karel is on 1st street
}
```

PATTERN

Once or Not at All

We could also use the following loop to make sure `karel` has at least eight things in its backpack:

```
while (karel.countThingsInBackpack() < 8)
{ karel.pickThing();
}
```

PATTERN

Zero or More Times

A total of six **comparison operators** can be used to compare integers. They are shown in Table 4-1.

(table 4-1)

Java comparison operators

Operator	Name	Example	Meaning
`<`	less than	`karel.getAvenue() < 5`	Evaluates to `true` if `karel`'s current avenue is strictly less than 5; otherwise, evaluates to `false`.
`<=`	less than or equal	`karel.getStreet() <= 3`	Evaluates to `true` if `karel`'s current street is less than or equal to 3; otherwise, evaluates to `false`.
`==`	equal	`karel.getStreet() == 1`	Evaluates to `true` if `karel` is currently on 1st Street; otherwise, evaluates to `false`.
`!=`	not equal	`karel.getStreet() != 1`	Evaluates to `true` if `karel` is not currently on 1st Street; if the robot *is* on 1st Street, evaluates to `false`.

(table 4-1) *continued*

Java comparison operators

Operator	Name	Example	Meaning
>=	greater than or equal	`5 >= karel.getAvenue()`	Evaluates to `true` if 5 is greater than or equal to `karel`'s current avenue. Most people find this easier to understand when written as `karel.getAvenue() <= 5`.
>	greater than	`karel.getAvenue() > 5`	Evaluates to `true` if `karel`'s current street is strictly greater than 5; otherwise, evaluates to `false`.

The examples in Table 4-1 always show an integer on only one side of the comparison operator. Java is much more flexible than this, however. For example, it can have a query on both sides of the operator, as in the following statements:

```
if (karel.getAvenue() == karel.getStreet())
{ ...
}
```

This test determines whether `karel` is on the diagonal line of intersections (0, 0), (1, 1), (2, 2), and so on. It also includes intersections with negative numbers such as (-3, -3).

Java also allows a more complex arithmetic expression on either side. The following code tests whether the robot's avenue is five more than the street. Locations where this test returns `true` include (0, 5) and (1, 6).

```
if (karel.getAvenue() == karel.getStreet() + 5)
{ ...
}
```

KEY IDEA

Assignment (=) is not *the same as equality (==).*

One common error is writing `=` instead of `==`. The assignment statement, such as `Robot karel = new Robot(…)` uses a single equal sign. Comparing integers, on the other hand, uses two equal signs. Fortunately, Java usually catches this error and issues a compile-time error. Some other languages, such as C and C++, do not.

4.3 Reexamining Harvesting a Field

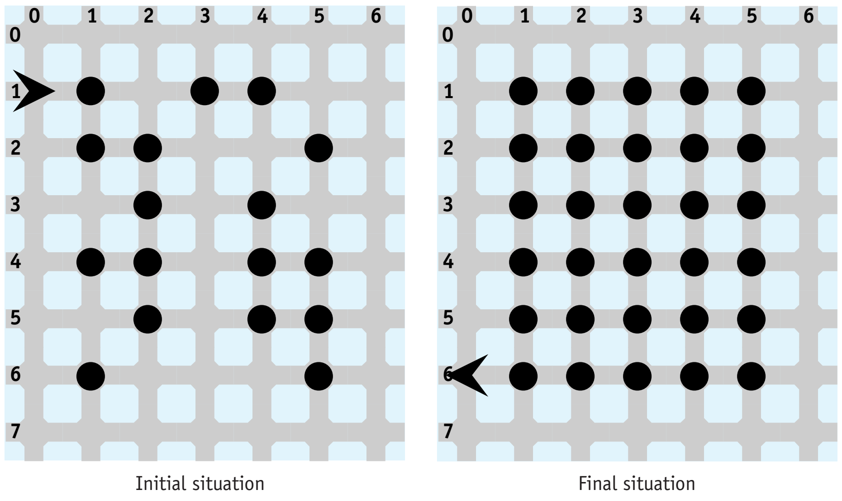
Let's return to the primary example from Chapter 3, traversing a field of `Thing` objects. In the following examples we use `if` and `while` statements to solve variations of that problem. The original program is in Listing 3-3.

We again use the dialogue format introduced in Chapter 3 to reveal the thinking that leads to the final solution.

4.3.1 Putting a Missing Thing

Instead of harvesting a field, consider planting a field. Such a robot class, `PlanterBot`, is the same as Listing 3-3 except for renaming the methods to use "plant" instead of "harvest" and, in the `plantIntersection` method, using `putThing` instead of `pickThing`.

Now, consider a minor variation: someone has already planted some intersections, but not all. Our `PlanterBot`, `karel`, must go through the field and put a `Thing` on only those intersections that don't already have one. The initial and final situations are shown in Figure 4-7. Of course, `karel` must either be created with a supply of `Thing` objects in its backpack (see the documentation for an alternate constructor) or pick up a supply before it starts.



(figure 4-7)

Initial and final situations for planting intersections that don't have a `Thing`

Expert What does `karel` have to do?

Novice It must traverse the entire field, as before. Each time it comes to an intersection it must ensure that the intersection has a `Thing` before the `PlanterBot` leaves.

Expert Does it always perform the same action at each intersection?

Novice No. Its actions depend on whether a `Thing` is already there.

Expert Does the `PlanterBot` perform its actions once or not at all? Or does it perform them zero or more times?

Novice It either puts a `Thing` down or it doesn't, depending on whether a `Thing` is already present on the intersection. So it does an action, putting a `Thing`, once or not at all.

Expert Can you write that in pseudocode?

Novice I had a feeling that was coming.... Performing an action once or not at all uses an `if` statement, as follows:

```
if (there isn't a thing on this intersection)
{ put a thing on this intersection
}
```

Expert How can you express the test for the `if` statement in Java?

Novice We haven't seen a test for the absence of a `Thing` on an intersection. The closest test we've studied is `canPickThing`—can the robot pick up a thing from this intersection. If it can, there must be a `Thing` present. If it can't, there isn't a `Thing` present.

I think the test we want is `if (`*not a thing that can be picked up*`)`. "Not", in Java, is written with an exclamation point. Therefore, we want `!this.canPickThing`.

The definition of `PlantThing` follows the pseudocode closely and is shown in Listing 4-1.

Listing 4-1: *The* `plantIntersection` *method*

```
/** Ensure that there is one thing on this intersection. */
public void plantIntersection()
{ if (!this.canPickThing())
  { this.putThing();
  }
}
```

4.3.2 Picking Up a Pile of Things

Suppose that instead of a single `Thing`, each of the field's intersections may have many `Things`. The original program in Listing 3-3 has a method, `harvestIntersection`, declared as follows:

```
/** Harvest one intersection. */
public void harvestIntersection()
{ this.pickThing();
}
```

In the revised version of the program, we want this method to pick up all of the `Things` on the intersection.

Expert What does `karel` have to do?

Novice Pick up all the `Thing` objects that are on the same intersection as itself.

Expert Can it pick them all up with a single instruction?

Novice The `pickThing` instruction picks up one `Thing` at a time.

Expert Does the robot always perform the same actions to pick up all the `Things`?

Novice No. Its actions depend on how many `Things` are on the intersection. It must use a test to decide what to do.

Expert Is the decision to do the action once or not at all? Or is the decision to repeat the action zero or more times?

Novice The robot should repeat an action (picking up a `Thing`) zero or more times—until there is nothing left to pick up.

Expert Can you express this idea in pseudocode?

Novice Sure:

```
while (this robot can pick up a Thing object)
{ pick up the Thing object
}
```

Expert How can you express the test in Java?

Novice With `this.canPickThing()`.

This pseudocode can be expressed in Java and placed in a revised version of the `harvestIntersection` method as shown in Listing 4-2.

Listing 4-2: *A version of* `harvestIntersection` *that harvests all the* `Things` *there*

```
/** Harvest one intersection. */
public void harvestIntersection()
{ while (this.canPickThing())
  { this.pickThing();
  }
}
```

FIND THE CODE

ch04/harvest/

The `while` statement will continue to ask the question "Can this robot pick up a `Thing`?" until the answer is "no" or `false`. As long as the answer is "yes" or `true`, it will pick up a `Thing`. This method also works if some of the intersections don't have any `Things` on them. In that case, the first time the question is asked, the answer is "no, there is nothing here that can be picked up" and the body of the loop is not executed. In either case, when the loop is finished executing there will be nothing on the intersection that the robot can pick up.

4.3.3 Improving `goToNextRow`

The original `Harvester` class has two methods moving the robot one street south. One, `goToNextRow`, is used on the east side of the field. The other, `positionForNextHarvest`, is used on the west side of the field. The existing definitions of these two methods are as follows:

```
/** Go one row south and face west. */
public void goToNextRow()
{ this.turnRight();
  this.move();
  this.turnRight();
}

/** Go one row south and face east. */
public void positionForNextHarvest()
{ this.turnLeft();
  this.move();
  this.turnLeft();
}
```

It would be preferable to have a single method that will work correctly at either end of the row. Now, the distinction between `goToNextRow` and `positionForNextHarvest` is not clear from the names of the methods. It would be easier for people reading and writing the code to have only one descriptive name like `goToNextRow`.

Expert What does the robot have to do?

Novice When it is at the east end of the row, it must turn right to move to the next row. When it is at the west end it must turn left.

Expert So the robot must decide if it is at the east end of the row or the west end and the action it carries out is to turn. Is the action performed once or not at all, or is it performed zero or more times?

Novice The method is called many times—once at the end of each row. So in that sense the action is performed many times.

Expert Hmm.... That's not what I had in mind. Let's focus on only one invocation of `goToNextRow`. The robot is at the end of one particular row and needs to perform an action. Is that action performed once or not at all or is it performed zero or more times?

Novice It's once or not at all. It performs a group of actions (turn right, move, turn right) once if it is at the east end of the row and not at all if it isn't. Similarly, it performs a group of actions once if it is at the west end and not at all if it isn't.

Expert What statement can we use to control the robot's actions?

Novice It's the `if` statement that performs an action once or not at all. But I'm confused, because it isn't a single test. We need one test for the east end of the row and another test for the west end of the row.

Expert Perhaps it's not only two tests we need, but two complete `if` statements.

Novice So using pseudocode, it would be as follows:

```
if  (this robot is at the east end of the row)
{  turn right, move, and turn right again
}
if  (this robot is at the west end of the row)
{  turn left, move, and turn left again
}
```

Expert Exactly. Now, how can you determine if the robot is at the east end of the row?

Novice Looking at Figure 3-2, the east end of the row is on Avenue 5 and the west end of the row is on Avenue 1. In the first `if` statement, we can compare `this.getAvenue` to 5 and in the second `if` statement we can compare `this.getAvenue` to 1.

This pseudocode and the insight into the tests can be turned into the required Java method, as shown in Listing 4-3.

Listing 4-3: *A revised version of* `goToNextRow` *that will work at either end of the row.*

```
/** Go one row south. The robot must be on either Avenue 1 or Avenue 5. */
public void goToNextRow()
{ if (this.getAvenue() == 5)       // at the east end of the row
  { this.turnRight();
    this.move();
    this.turnRight();
  }
  if (this.getAvenue() == 1)       // at the west end of the row
```

FIND THE CODE

ch04/harvest/

LOOKING AHEAD

Written Exercise 4.3 focuses on this method.

Listing 4-3: *A revised version of* `goToNextRow` *that will work at either end of the row.* (continued)

```
  { this.turnLeft();
    this.move();
    this.turnLeft();
  }
}
```

4.4 Using the `if-else` Statement

The `if` statement performs an action once or not at all. Another version of the `if` statement, the `if-else` statement, chooses between two groups of actions. It performs one or it performs the other based on a test. Unlike the `if` statement, the `if-else` statement always performs an action. The question is, which action?

The general form of the `if-else` is as follows:

PATTERN

Either This or That

```
if («test»)
{ «statementList1»
} else
{ «statementList2»
}
```

The form of the `if-else` statement is similar to the `if` statement, except that it includes the keyword `else`, *`«statementList2»`* and another set of braces. Note the absence of a semicolon before the word `else` and at the end.

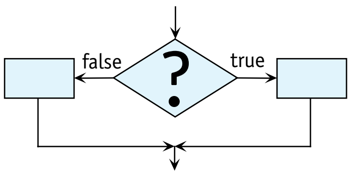
An `if-else` is executed in much the same manner as an `if`. First, the *`«test»`* is evaluated to determine whether it is `true` or `false` in the current situation. If the *`«test»`* is `true`, *`«statementList1»`* is executed; if the test is `false`, *`«statementList2»`* is executed. Thus, depending on the current situation, either *`«statementList1»`* or *`«statementList2»`* is executed, but not both. The first statement list is called the **then-clause**, just like an `if` statement. The second statement list is called the **else-clause**.

When the else-clause is empty, the `if-else` statement behaves just like the `if` statement. In fact, the `if` statement is just a special case of the `if-else` statement.

The flowchart for an `if-else` statement is shown in Figure 4-8.

(figure 4-8)

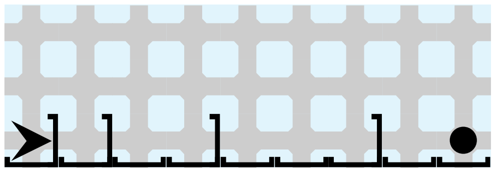
Flowchart for an `if-else` *statement*



4.4.1 An Example Using `if-else`

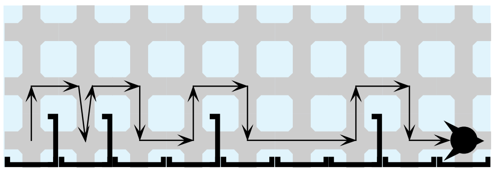
Let’s look at an example that uses the `if-else` statement. Suppose that we want to program a `Racer` robot to run a hurdle race, where vertical wall sections represent hurdles. The hurdles are only one block high and are randomly placed between any two intersections in the race course. The finish line is marked with a `Thing`. One of the many possible race courses for this task is illustrated in Figure 4-9. Figure 4-10 shows the final situation and the path the robot should take for this particular race course.

Here we think of the city as being vertical, with down being south. To run the fastest race possible, we require the robot to jump if, and only if, it is faced with a hurdle.



(figure 4-9)

Hurdle-jumping robot’s initial situation



(figure 4-10)

Hurdle-jumping robot’s final situation and the path it took

We will assume that a stepwise refinement process is being used and that the `Racer` class is partially developed, as shown in Listing 4-4. We need to continue the process by developing the `raceStride` method. It should move the robot forward by one intersection.

Listing 4-4: *A partially developed implementation of* `Racer`

```
import becker.robots.*;

/** A class of robots that runs a hurdles race (steeplechase).
 *
 *  @author Byron Weber Becker */
public class Racer extends RobotSE
{
   /** Construct a new hurdle-racing robot. */
   public Racer(City aCity, int str, int ave, Direction aDir)
   { super(aCity, aStreet, anAvenue, aDir);
   }
```

Listing 4-4: *A partially developed implementation of* `Racer` (continued)

```

   /** Run the race by repeatedly taking a raceStride until the finish line is crossed. */
   public void runRace()
   { while (!this.canPickThing())
     { this.raceStride();
     }
   }
}
```

We could easily develop a class of robots that run this race by jumping between every pair of intersections. Although this strategy is simple to program, it doesn't meet the requirements of running the fastest race possible. Instead, we must program the robot to move straight ahead when it can, and jump over hurdles only when it must.

Expert So, assume the `Racer` is on an intersection of the racetrack and ready to take its next stride. What should it do?

Novice It needs to move forward to the next intersection.

Expert Does the robot always perform the same actions?

Novice No, they depend on the situation. If there is a hurdle, it needs to be jumped. If there isn't a hurdle, the `Racer` can just move.

Expert Can you express these thoughts using pseudocode?

Novice

```
if (facing a hurdle)
{ jump the hurdle
}
move
```

Expert You seem to be thinking that the robot should always move. The only question is whether it jumps a hurdle first. Have you considered what would happen if there are two consecutive hurdles? The first pair of hurdles in Figure 4-9 shows that kind of a situation.

Novice Well, it would jump the first hurdle, landing right before the second one. Then it would move… and crash into the hurdle. I guess we need a different plan.

Expert So, what does the robot need to do?

Novice It should either jump the hurdle or move (but not both), depending on whether it is facing a hurdle. In pseudocode,

```
if (facing a hurdle)
{ jump the hurdle
} else
{ move
}
```

Putting these ideas into a method results in the following code. It should be added to Listing 4-4. The `jumpHurdle` method can be developed using the stepwise refinement techniques found in Section 3.2.

```
public void raceStride()
{ if (!this.frontIsClear())
  { this.jumpHurdle();
  } else
  { this.move();
  }
}
```

PATTERN

Either This or That

4.5 Writing Predicates

In Section 4.2.1, we learned about eight queries that are built in to the `Robot` class and examples of how to use them. But what if we frequently need to check if a robot's front is *not* clear? We could write the following code, which includes a negation:

```
if (!this.frontIsClear())
{ // what to do if this robot's front is blocked
}
```

PATTERN

Simple Predicate

However, the following positive statement is easier to understand:

```
if (this.frontIsBlocked())
{ // what to do if this robot's front is blocked
}
```

Fortunately, Java allows us to define our own predicates. Recall that a predicate is a method that returns one of the Boolean values, either `true` or `false`. Returning a value has two requirements:

- The method's return type must be indicated in its declaration. The type `boolean` is appropriate for predicates. It replaces the keyword `void` we have used so far.

KEY IDEA

A return statement always causes a method to stop immediately and return to its caller.

- The new predicate must indicate what value to return. To do so, we need a new kind of statement: the `return` statement. The form of the `return` statement is the reserved word `return`, followed by an expression. Because our method's return type is `boolean`, the expression must evaluate to a Boolean value, either `true` or `false`. Executing a `return` statement immediately terminates the execution of the method.

4.5.1 Writing `frontIsBlocked`

A Boolean expression that can be used as a test in an `if` or `while` statement can be easily turned into a predicate by inserting it into the following template:

PATTERN

Simple Predicate

```
«accessModifier» boolean «predicateName»(«optParameters»)
{ return «booleanExpression»;
}
```

where

- *«accessModifier»* is `public`, `protected`, or `private`.
- *«predicateName»* is the name of the predicate. Valid names are the same as for any other method.
- *«optParameters»* provide additional information from the client. Parameters are optional; many predicates do not have them.
- *«booleanExpression»* evaluates to either `true` or `false` and is the expression that could be placed in the test of an `if` or `while` statement.

The Java method for the `frontIsBlocked` predicate is as follows:

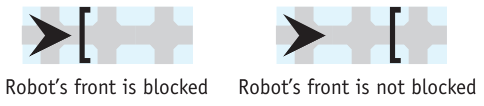
PATTERN

Simple Predicate

```
public boolean frontIsBlocked()
{ return !this.frontIsClear();
}
```

When this method is called, it evaluates the Boolean expression `!this.frontIsClear()`. In the situation shown on the left side of Figure 4-11, `frontIsClear` evaluates to `false` and, because of the `!`, the expression as a whole evaluates to `true`. This value is returned. It is `true` that the robot's front is blocked.

On the other hand, consider the situation shown on the right side of Figure 4-11. `frontIsClear` evaluates to `true`, but is negated by the `!`, resulting in the entire expression evaluating to `false`—the robot's front is *not* blocked.



(figure 4-11)

Evaluating `frontIsBlocked` *in two different situations*

4.5.2 Predicates Using Non-Boolean Queries

When developing the `goToNextRow` method (see Listing 4-3) we included the following `if` statement and comment:

```
3  { if (this.getAvenue() == 5)      // at the east end of the row
4    { this.turnRight();
5      this.move();
6      this.turnRight();
7    }
```

With an appropriate predicate, line 3 could be rewritten as follows:

```
3  { if (this.atRowsEastEnd())
```

Predicates such as `atRowsEastEnd` can help us produce self-documenting code. The goal of **self-documenting code** is to make the code so readable that internal comments explaining the code are not needed. In this case, `atRowsEastEnd` tells us the intention of the test nearly as well as the comment, enabling us to remove the comment. It's a good idea to replace comments with self-documenting code because comments are often overlooked as the code changes. When this happens comments can become incomplete, misleading, or wrong.

KEY IDEA

Appropriately named predicates lead to self-documenting code.

Having a predicate such as `atRowsEastEnd` is also an advantage if the test must be done at many places in the program. With a predicate, when the problem changes to have rows that end at a different place or a bug is discovered, there is only one easily identified place to change.

Coding this predicate follows the same procedure as before: take the Boolean expression that would be included in the `if` or `while` statement and place it inside a method. The method is as follows:

```
protected boolean atRowsEastEnd()
{ return this.getAvenue() == 5;
}
```

PATTERN

Simple Predicate

The query `getDirection` is similar to `getAvenue` except that it returns one of the special values such as `Direction.NORTH` or `Direction.EAST`. These values can be compared using `==` and `!=`, but not <, >, and so on.

KEY IDEA

Enumerated types such as `Direction` can be tested for equality only.

This fact can be used to create the predicate `isFacingSouth` as follows:

```
protected boolean isFacingSouth()
{ return this.getDirection() == Direction.SOUTH;
}
```

This predicate, along with `isFacingNorth`, `isFacingEast`, and `isFacingWest`, are used often enough that they have been added to the `RobotSE` class.

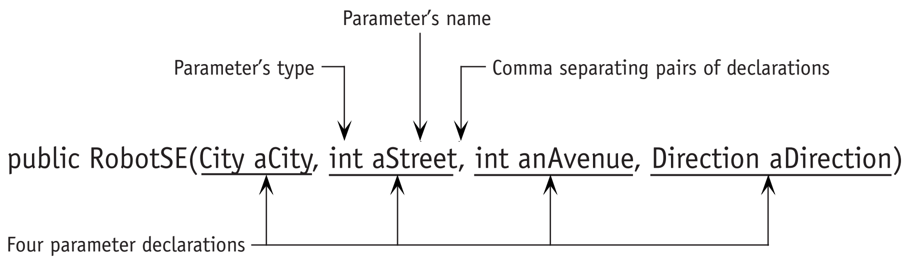
4.6 Using Parameters

The ability to make decisions with `if` and `while` statements gives our methods a tremendous amount of flexibility. They can have even more flexibility when we use parameters. We have already used parameters by passing them arguments to specify where to place robots when they are created, how large to draw a rectangle, and which icon a `Thing` should use to display itself.

We have declared parameters every time we extended the `Robot` class and wrote a constructor, as well as in Section 3.7.1 where we used parameters to place a stick figure at a precise location. From these contexts, we know that each parameter declaration has a type, such as `int`, and a name. Parameter declarations are placed between the parentheses following the method's name. If there is more than one declaration, consecutive pairs are separated with commas. These points are illustrated in Figure 4-12.

(figure 4-12)

Declaring parameters


Parameter's name
Parameter's type
Comma separating pairs of declarations
public RobotSE(City aCity, int aStreet, int anAvenue, Direction aDirection)
Four parameter declarations

The type of the parameter determines what kind of values can be used as arguments. If the parameter's type is `int`, the arguments must be integers such as 15 or -23. Similarly, only an object of type `City` can be passed as an argument to a parameter of type `City`.

Inside the constructor or method, the name of the parameter can be used to reference the value passed to it as an argument. Let's use an example to understand how this works.

Suppose we want a subclass of `Robot` that can easily tell us if it has gone past a particular avenue, say Avenue 50. We could use the `getAvenue` method and compare it to 50, but our code is more self-documenting with a predicate, as follows:

```
if (this.isPastAvenue(50))
{  // what to do when the robot has strayed too far
```

The `isPastAvenue` method is written as follows:

PATTERN

Simple Predicate

```
private boolean isPastAvenue(int anAvenue)
{ return this.getAvenue() > anAvenue;
}
```

KEY IDEA

The parameter refers to the value passed as an argument.

Inside the `isPastAvenue` method, `anAvenue` refers to the value passed as an argument. In the preceding example, that value is 50 and the Boolean expression is evaluated as `this.getAvenue() > 50`. However, if the argument is 100, as in `if (this.isPastAvenue(100))`, then inside `isPastAvenue` the parameter `anAvenue` will refer to the value 100. This one method can be used with any avenue—a tremendous amount of flexibility compared to methods without parameters.

4.6.1 Using a `while` Statement with a Parameter

A parameter can also be used in the test controlling a `while` or `if` statement—and can make the method more flexible, as well. For example, the following method moves a robot east to Avenue 50:

```
/** Move the robot east to Avenue 50. The robot must already be facing east and
*   must be on an avenue that is less than 50. */
public void moveToAvenue50()
{ while (this.getAvenue() < 50)
  { this.move();
  }
}
```

This method is extremely limited—it is only useful to move the robot to Avenue 50. With a parameter, however, it can be used to move the robot to any avenue east of its current location. The following method includes a parameter with an appropriate documentation comment:

```
/** Move the robot east to destAve. The robot must already be facing east and
*   must be on an avenue that is less than destAve.
*   @param destAve      The destination avenue to move to. */
public void moveToAvenue(int destAve)
{ while (this.getAvenue() < destAve)
  { this.move();
  }
}
```

The statement `karel.moveToAvenue(50)` moves `karel` to Avenue 50 while `karel.moveToAvenue(5000)` moves `karel` much farther. In both cases the argument, 50 or 5000, is referred to inside the method as `destAve`.

4.6.2 Using an Assignment Statement with a Loop

Consider the following ill-advised method:

```
public void step(int howFar)
{ while (howFar > 0)
  { this.move();
  }
}
```

KEY IDEA

Something in the body of the loop must change how the test is evaluated.

Why is it ill advised? Consider telling `karel` to step four times with `karel.step(4)`. The `while` statement evaluates the expression `howFar > 0`, concluding that it is `true`—four is larger than zero. The statement executes the `move` method and evaluates `howFar > 0` again. `howFar` is still four, four is still greater than zero, and so the `move` method is executed again. The value of `howFar` does not change in the body of the loop, the test will always be true, and the loop will execute "forever."

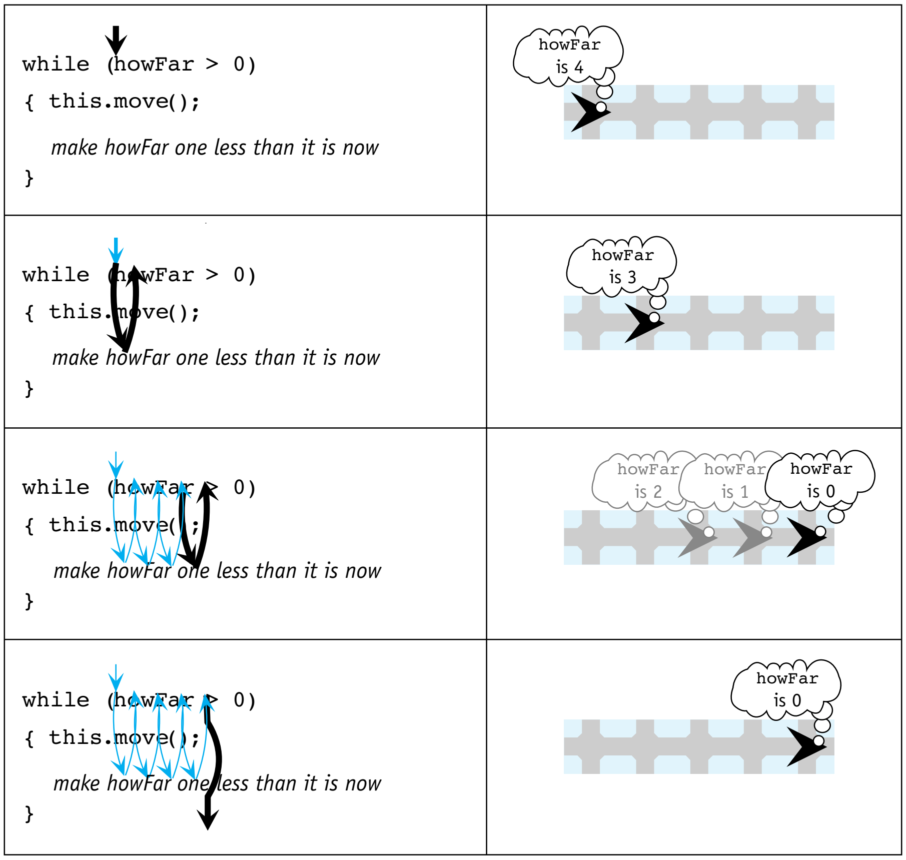
Suppose, however, that we could decrease the value of `howFar` in the body of the loop, as indicated by the following pseudocode:

PATTERN

Count-Down Loop

```
public void step(int howFar)
{ while (howFar > 0)
  { this.move();
    make howFar one less than it is now
  }
}
```

That is, `howFar` starts with the value `4`, then has the value `3`, then `2`, and so on—assuming that `step` was called with an argument of `4`, as in the preceding code. Now we have a useful method, as illustrated in Figure 4-13. When `howFar` reaches the value 0, the loop stops and the robot has traveled four intersections. If we want `karel` to take four steps, we write `karel.step(4)`. If we want `karel` to take 400 steps, it's as easy as writing `karel.step(400)`.



```
while (howFar > 0)
{ this.move();
   make howFar one less than it is now
}
```

```
while (howFar > 0)
{ this.move();
   make howFar one less than it is now
}
```

```
while (howFar > 0)
{ this.move();
   make howFar one less than it is now
}
```

```
while (howFar > 0)
{ this.move();
   make howFar one less than it is now
}
```

(figure 4-13)

Illustrating the execution of a count-down loop

A `while` statement that counts from a number down to zero is called a **count-down loop**.

We still need to explain, of course, how to make `howFar` be one less than it is now. This change is accomplished with an **assignment statement**. An assignment statement evaluates an expression and assigns the resulting value to a variable. A parameter is one kind of variable.

The following is an assignment statement that decreases `howFar`'s value by one:

```
howFar = howFar – 1;
```

When this assignment statement is executed, it evaluates the expression on the right side of the equal sign by subtracting one from the current value of `howFar`. When `howFar` refers to the value 4, `howFar – 1` is the value 3. The value 3 is then assigned to `howFar`. The parameter will refer to this new value until we change it with another assignment statement or the method ends. Parameters are destroyed when the method declaring them completes its execution.

LOOKING AHEAD

Other kinds of variables will be discussed in Chapters 5 and 6.

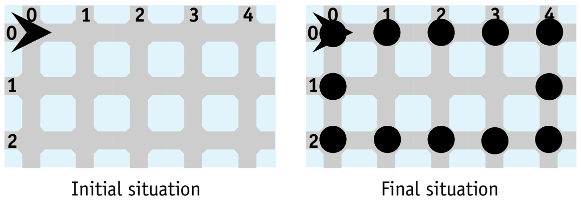
KEY IDEA

A count-down loop performs actions a specified number of times.

A count-down loop, like the one used in `step`, can be used to do many different activities a specified number of times: picking up a specified number of things; turning a specified number of times; and, with a more complicated loop body, harvesting a specified number of rows from a field.

4.6.3 Revisiting Stepwise Refinement

A count-down loop can have a body that is more complex than a single `move`. The body could contain more statements, or preferably, a call to a helper method. Furthermore, a method can have more than one parameter and parameters can be passed as arguments to helper methods. In this section we will develop a class of robots that illustrate all of these principles. Our robots will plant `Things` in the shape of a hollow rectangle. The width and height of the rectangle is specified with parameters when the `plantRect` method is invoked. A sample initial and final situation is shown in Figure 4-14.



(figure 4-14)

One pair of many possible initial and final situations for planting a rectangle

Our class is called `RectanglePlanter` and has a single service, `plantRect`. We want the robot to be able to plant many different sizes of rectangles, a kind of flexibility that is well-suited for using parameters. Two parameters are needed—one for the rectangle's width and one for the height. This usage follows the `setSize` command for a `JFrame` and the `drawRect` command in the `Graphics` class. The following code fragment instructs `karel` to plant a rectangle five `Things` wide and three `Things` high, as shown in Figure 4-14. One notable feature is that the constructor allows specifying the number of things initially in the robot's backpack. Here it is set to 50.

```
RectanglePlanter karel = new RectanglePlanter(
                                garden, 0, 0, Direction.EAST, 50);
...
karel.plantRect(5, 3);
```

Implementing `plantRect`

The `plantRect` method requires two integer parameters, one for the width and one for the height, corresponding to the arguments `5` and the `3` in the previous code fragment. The beginning of the class, including a stub for `plantRect` and the constructor allowing the initial number of `Things` in the backpack to be set, is as shown in Listing 4-5.

Listing 4-5: *Beginning the* `RectanglePlanter` *class*

FIND THE CODE

ch04/rectanglePlanter/

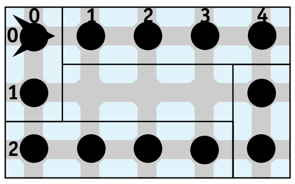
```
import becker.robots.*;

/** A class of robots that plants Things in the form of a hollow rectangle.
 *
 *  @author Byron Weber Becker */
public class RectanglePlanterextendsRobotSE
{
   /** Create a new rectangle planter.
    *   @param aCity          The robot's city.
    *   @param aStreet        The robot's initial street.
    *   @param anAvenue       The robot's initial avenue.
    *   @param aDir           The robot's initial direction.
    *   @param numThings      The number of things initially in the robot's backpack. */
   public RectanglePlanter(City aCity, int aStreet,
           int anAvenue, Direction aDir, int numThings)
   { super(aCity, aStreet, anAvenue, aDir, numThings);
   }

   /**  Plant a hollow rectangle of Things. The robot must be positioned in the
    *   rectangle's upper-left corner facing east.
    *   @param width        The number of avenues wide.
    *   @param height       The number of streets high. */
   public void plantRect(int width, int height)
   {
   }
}
```

To implement `plantRect` we need a strategy. Assuming we don't want two `Things` on each corner, one strategy is to plant a side of the rectangle by planting one less than the length of the side. This strategy is illustrated in Figure 4-15. To make a side of length five, for example, the robot will plant four things beginning with the next intersection and then turn right. In general, the number of `Things` it plants is one less than the length of the side.

Planting a side four times, with the appropriate lengths for each side, results in the desired rectangle.



(figure 4-15)

Strategy for planting a rectangle

To carry this out, we can create a helper method, `plantSide`, that takes a parameter specifying how long that side of the rectangle should be. Assuming that we can write this helper method, the `plantRect` method can be completed as follows:

```
public void plantRect(int width, int height)
{ this.plantSide(width);
  this.plantSide(height);
  this.plantSide(width);
  this.plantSide(height);
}
```

Note that the parameters, `width` and `height`, are passed as arguments to the helper method. However, the `plantSide` method only requires one parameter because it is only concerned with the length of a side and not the overall dimensions of the rectangle.

Implementing `plantSide`

The strategy for `plantSide` was already determined when outlining the overall strategy. We already know, from the way it was used in `plantRect`, that it has a single, integer parameter. The parameter can have any name, but we will call it `length` because it determines the length of the side.

`plantSide` plants a line that is one less than the length of the side and then turns right. In pseudocode, this is as follows:

```
length = length - 1
plant a line of Things that is length Things long
turn right
```

The Java translation of this pseudocode uses of an assignment statement to decrease the value passed to the parameter by one and a helper method to plant a line of things. The completed method follows:

```
/** Plant one side of the rectangle with Things, beginning with the next intersection.
*   @param length       The length of the line. */
protected void plantSide(int length)
{ length = length - 1;
  this.plantLine(length);
  this.turnRight();
}
```

Implementing `plantLine`

Planting a line of `Things` requires repeating actions zero or more times (a `while` statement). It's not a question of performing actions once or not at all (an `if` statement) or performing either this action or that action (an `if-else` statement).

What are the actions we must repeat? To plant a line of three things beginning with the next intersection, for example, we must perform the following actions:

```
move
plant a thing
move
plant a thing
move
plant a thing
```

The actions that are repeated are moving and planting a thing. They form the body of the `while` statement. We want them to be performed a specific number of times, as specified by the parameter. This is an ideal application for a count-down loop. This method is, in fact, identical to the `step` method developed earlier except that we also need to plant a `Thing` on the intersection. The code for the method follows:

```
/** Plant a line of Things beginning with the intersection in front of the robot.
*   @param length          The length of the line. */
protected void plantLine(int length)
{ while (length > 0)
  { this.move();
    this.plantIntersection();
    length = length - 1;
  }
}
```

PATTERN

Count-Down Loop

Finally, `plantIntersection` is a method to make future change easy. It contains a single call to `putThing`, as shown in the following code:

```
/** Plant one intersection. */
protected void plantIntersection()
{ this.putThing();
}
```

This completes the implementation of the `RectanglePlanter` class. In the course of its development we have demonstrated:

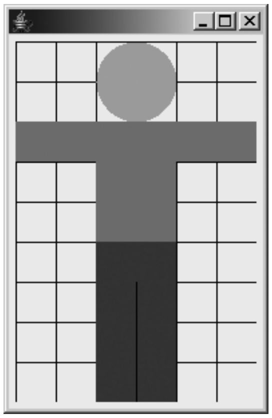
- A method with more than one parameter
- Passing parameters as arguments to helper methods
- A count-down loop with a more complex set of actions

4.7 GUI: Scaling Images

The graphical user interface section of Chapter 2 introduced us to drawing pictures similar to the one shown in Figure 4-16. In this section, we'll see how to use queries in the `JComponent` class to make our image adapt to different sizes.

(figure 4-16)

Stick figure and a grid used to design it



The code for the main method is shown in Listing 2-13. `StickFigure` is the class that does the actual drawing. It was originally shown in Chapter 2 and is reproduced in Listing 4-6. Notable points are that it sets the preferred size for the component in the constructor at lines 8 and 9, and overrides `paintComponent` to draw the actual image.

FIND THE CODE

ch02/stickFigure/

Listing 4-6: *A class to draw a stick figure*

```
 1 import java.awt.*;
 2 import javax.swing.*;
 3 
 4 public class StickFigure extends JComponent
 5 {
 6   public StickFigure()
 7   { super();
 8     Dimension prefSize = new Dimension(180, 270);
 9     this.setPreferredSize(prefSize);
10   }
11 
12   // Draw a stick figure.
13   public void paintComponent(Graphics g)
14   { super.paintComponent(g);
15 
16     // head
17     g.setColor(Color.YELLOW);
18     g.fillOval(60, 0, 60, 60);
19 
20     // shirt
21     g.setColor(Color.RED);
22     g.fillRect(0, 60, 180, 30);
23     g.fillRect(60, 60, 60, 90);
```

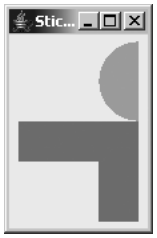
Listing 4-6: *A class to draw a stick figure* (continued)

```

        // pants
        g.setColor(Color.BLUE);
        g.fillRect(60, 150, 60, 120);
        g.setColor(Color.BLACK);
        g.drawLine(90, 180, 90, 270);
    }
}
```

Now, suppose that we wanted the image to be a different size. The preferred size set in the `StickFigure` constructor could be replaced with, for example, `new Dimension(90, 135)` to make the image half as big in each dimension.

There is a problem, however. Making only this one change results in an image similar to the one shown in Figure 4-17. Unfortunately, all of the calculations to draw the stick figure were based on the old size of 180 pixels wide and 270 pixels high.



(figure 4-17)

Naively changing the size of the stick figure

4.7.1 Using Size Queries

We can make the image less sensitive to changes in size by drawing it relative to the current size of the component. We can obtain the current size of the component, in pixels, with the `getWidth` and `getHeight` queries. To paint a shape that covers a fraction of the component, multiply the results of these queries by a fraction. For example, to specify an oval that is two-sixths of the width of the component and two-ninths of the height, we can use the following statement:

```
g.fillOval(0, 0, this.getWidth()*2/6, this.getHeight()*2/9);
```

The first two parameters will place the oval at the upper-left corner of the component.

The original stick figure was designed on a grid six units wide and nine units high. Figure 4-16 shows this grid explicitly and makes it easy to figure out which fractions to multiply by the width or the height. For example, the head can be painted with

```
g.fillOval(this.getWidth()*2/6, this.getHeight()*0/9,
           this.getWidth()*2/6, this.getHeight()*2/9);
```

The first pair of parameters says the head's bounding box should start 2/6th of the component's width from the left edge and 0/9th of the component's height from the top. The second parameter could be replaced by 0.

Converting the remaining method calls to use the `getWidth` and `getHeight` queries follows a similar pattern. It is tedious, however. Fortunately, there is a better approach.

4.7.2 Scaling an Image

Using `getWidth` and `getHeight` to scale an image follows a very predictable pattern, as shown in the last section. Fortunately, the designers of Java have provided a way for us to exploit that pattern with much less work on our part. They have provided a way for the computer to automatically multiply by the width or the height of the component and divide by the number of units on our grid. All we need to do is supply the numerator of the fraction that places the image or says how big it is. For example, in the previous section we wrote the following statements:

```
g.fillOval(this.getWidth()*2/6, this.getHeight()*0/9,
           this.getWidth()*2/6, this.getHeight()*2/9);
```

Java's drawing methods can be set up so that this method call is replaced with the following statement:

```
g.fillOval(2, 0, 2, 2);
```

Using this approach requires three things: Using a more capable version of the `Graphics` object, setting the scale to be used in drawing, and scaling the width of the lines to use in drawing. All are easy and follow a pattern consisting of the following four lines inserted at the beginning of `paintComponent`:

PATTERN
Scale an Image

```
// Standard stuff to scale the image
Graphics2D g2 = (Graphics2D)g;
g2.scale(this.getWidth()/6, this.getHeight()/9);
g2.setStroke(new BasicStroke(1.0F/this.getWidth()));
```

Explaining this code in more detail requires advanced concepts; however, the overview is as follows:

- Line 2 makes a larger set of capabilities in `g` available. More about this in Chapter 12.
- Line 3 tells the `Graphics` object how to multiply values to scale our paintings appropriately.
- Line 4 makes the width of a line, also called a stroke, proportional to the scaling performed in line 3.

Whether or not we understand exactly what these lines of code do, using them is easy:

- Import the package `java.awt.*`.
- Copy these four lines to the beginning of your `paintComponent` method.
- Decide on the size of your grid, and change the "6" and "9" in the call to `scale` accordingly. For a 50 x 100 grid, change the 6 to 50 and the 9 to 100.
- Use `g2` instead of `g` to do the painting.

The resulting `paintComponent` method is shown in Listing 4-7.

Listing 4-7: *Painting a stick figure by scaling the image*

```
public void paintComponent(Graphics g)
{ super.paintComponent(g);

   // Standard stuff to scale the image
   Graphics2D g2 = (Graphics2D)g;
   g2.scale(this.getWidth()/6, this.getHeight()/9);
   g2.setStroke(new BasicStroke(1.0F/this.getWidth()));

   // head
   g2.setColor(Color.YELLOW);
   g2.fillOval(2, 0, 2, 2);

   // shirt
   g2.setColor(Color.RED);
   g2.fillRect(0, 2, 6, 1);
   g2.fillRect(2, 2, 2, 3);

   // pants
   g2.setColor(Color.BLUE);
   g2.fillRect(2, 5, 2, 4);
   g2.setColor(Color.BLACK);
   g2.drawLine(3, 6, 3, 9);
}
```

FIND THE CODE

ch04/stickFigure/

4.8 Patterns

4.8.1 The Once or Not at All Pattern

Name: Once or Not at All

Context: You are in a situation where executing a group of one or more statements may or may not be appropriate, depending on the value of a Boolean expression. If the expression is true, the statements are executed once. If the expression is false, they are not executed at all.

Solution: Use the `if` statement, as in the following two examples:

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

if (this.numStudentsInCourse() < 100)
{ this.addStudentToCourse();
}
```

In general, use the following template:

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

If possible, state the test positively. Easily understood predicate names contribute to easily understood code. For example, if the statement `if (this.numStudentsInCourse() < this.maxEnrollment())` is really checking if there is room in the course for one more student to be added, then using a predicate such as `if (this.roomInCourse())` makes the code easier to understand.

Consequences: Programs are able to respond differently, depending on the situation. In particular, an action is executed either once or not at all.

Related Patterns:

- The Either This or That pattern executes one of two actions. This pattern is a special case of that one.
- The Zero or More Times pattern is useful if an action is to be executed repeatedly rather than once or not at all.
- The Simple Predicate pattern is often used to create more easily understood *«test»*s.

4.8.2 The Zero or More Times Pattern

Name: Zero or More Times

Context: You are in a situation where a group of one or more statements must be executed a (usually) unknown number of times. It might be as few as zero times, or possibly many times. Whether to repeat the statements again can be determined with a Boolean expression.

Solution: Use a `while` statement to control the execution of the statements. When the test evaluates to `true`, the statements are executed and the test is performed again. This continues until the test evaluates to `false`. The following example is an example of the pattern:

```
while (this.frontIsClear())
{ this.turnLeft();
}
```

This loop turns the robot until it is facing a wall. If there is no wall blocking one of the four directions, it will turn forever.

In general, use the following template:

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

As with the `if` statement, a `while` statement is easiest to read and understand if the test is stated positively.

Consequences: The list of statements may be executed as few as zero times or they may execute forever. Such infinite loops are not desirable and should be guarded against.

Related Patterns:

- The Count-down Loop pattern is a special case of Zero or More Times.
- The Once or Not at All pattern executes an action either zero or one times rather than zero or more times.
- The Simple Predicate pattern is often used to create more easily understood *«test»*s.

4.8.3 The Either This or That Pattern

Name: Either This or That

Context: You have two groups of statements. Only one group should be executed and which one depends on the result of a Boolean expression.

Solution: Use an `if-else` statement to perform the test and govern which group of statements is executed, as in the following example:

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

Following is the general form of an `if-else` statement:

```
if («test»)
{ «statementGroup1»
} else
{ «statementGroup2»
}
```

If *«test»* evaluates to `true`, *«statementGroup1»* is executed and *«statement-Group2»* is not executed. If *«test»* evaluates to `false`, *«statementGroup2»* is executed and *«statementGroup1»* is not.

Consequences: The pattern allows programs to choose between two courses of action by evaluating a Boolean expression.

Related Patterns:

- The Once or Not at All pattern is a special case of this pattern useful for when there is only one action that may or may not be executed.
- If there are more than two actions, only one of which is executed, consider the Cascading-If pattern, described in Section 5.8.6.
- The Simple Predicate pattern is often used to create more easily understood *«test»*s.

4.8.4 The Simple Predicate Pattern

Name: Simple Predicate

Context: You are using a Boolean expression that is not as easy to read or understand as is desirable. Perhaps it is a complicated expression or perhaps the names of the queries don't match the problem.

Solution: Define a new method that performs the processing to find the required result, returning `true` or `false` to its client. Such methods are called predicates. For example, the following code defines a predicate named `frontIsBlocked`:

```
public boolean frontIsBlocked()
{ return !this.frontIsClear();
}
```

With this predicate, we could write `while (this.frontIsBlocked())` instead of `while (!this.frontIsClear())`.

Following is a template for such a predicate:

```
«accessModifier» boolean «predicateName»()
{ return «a boolean expression»;
}
```

Consequences: Statements that use a predicate, such as `this.frontIsBlocked()`, are easier to understand than those that use the equivalent test, such as `!this.frontIsClear()`. A predicate can be easily used many times, reducing the total time to code, test, and debug the program.

Related Patterns:

- 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 specialization of the more general Predicate pattern discussed in Section 5.8.5.

4.8.5 The Count-Down Loop Pattern

Name: Count-Down Loop

Context: You must perform an action a specified number of times. The number is often given via a parameter.

Solution: Write a `while` statement that uses a variable, often a parameter variable, to count down to zero. When the value reaches zero, the loop ends. The general form of the count-down loop is as follows:

```
while («variable» > 0)
{ «list of statements»
  «variable» = «variable» - 1;
}
```

A concrete example of the count-down loop is the `plantLine` method which puts a row of `Thing`s, the length of which is determined by the parameter.

```
public void plantLine(int length)
{ while (length > 0)
  { this.move();
    this.putThing();
    length = length - 1;
  }
}
```

Consequences: The Count-Down Loop pattern gives programmers the ability to perform an action a specified number of times, even if the number is large. Putting the count-down loop inside a method and using a parameter provides even more flexibility.

Related Patterns: The Count-Down Loop pattern is a special case of the Zero or More Times pattern.

4.8.6 The Scale an Image Pattern

Name: Scale an Image

Context: An image drawn by the `paintComponent` method in a subclass of `JComponent` needs to scale to different component sizes.

Solution: Draw the image based on a predefined grid for the coordinates. Then use the following code template to use that coordinate grid while drawing.

```
public void paintComponent(Graphics g)
{ super.paintComponent(g);

   // Standard stuff to scale the image
   Graphics2D g2 = (Graphics2D)g;
   g2.scale(this.getWidth()/«gridWidth»,
            this.getHeight()/«gridHeight»);
   g2.setStroke(new BasicStroke(1.0F/this.getWidth()));

   «statements using g2 to draw the image»
}
```

Consequences: The parameters provided to methods in `Graphics2D` such as `drawRect` are multiplied by either `this.getWidth()/«gridWidth»` or `this.getHeight/«gridHeight»`, as appropriate. The changes made to `g2` will persist even if it is passed to a helper method.

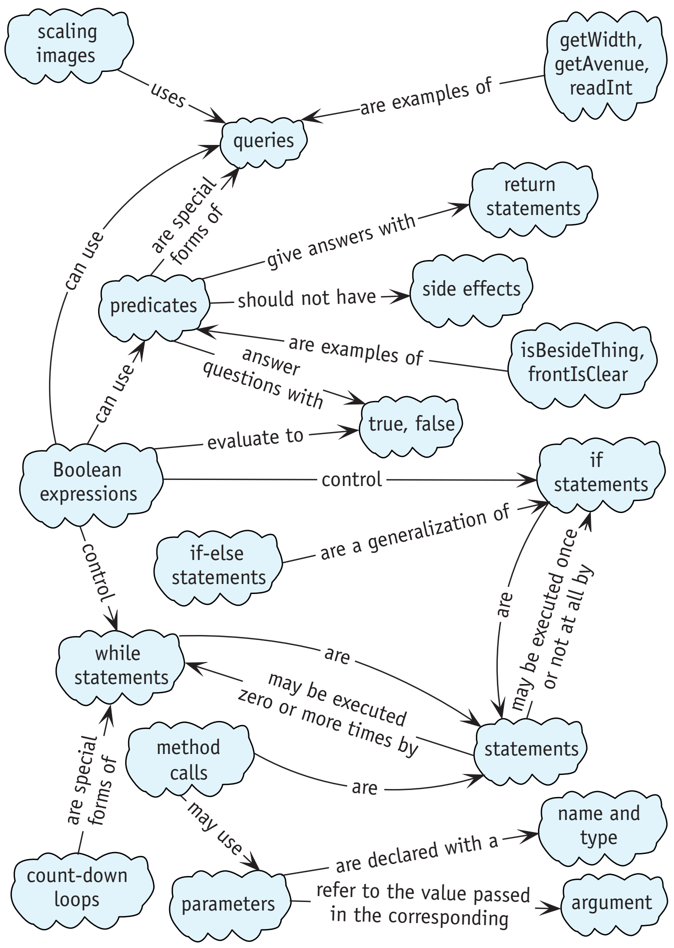
Related Patterns: This pattern is a specialization of the Draw a Picture pattern.

4.9 Summary and Concept Map

Programs that always execute the same set of statements, one after another, are fairly limited. Introducing Boolean expressions that can be evaluated—such as whether a robot is facing north or is on Avenue 8—makes programs much more flexible. They can execute statements once or not at all with an `if` statement or execute statements zero or more times with a `while` statement. An `if-else` statement uses the test to determine which of two actions to execute and a count-down loop can execute an action a specified number of times.

Using predicates in the tests used by `if` and `while` statements can make them easier to understand, debug, test, and maintain, all of which increases the quality of programs.

Boolean expressions used in `if` and `while` statements allow a program to respond to varying situations. Using parameters can make a program even more flexible. A parameter receives a value passed as an argument from the client code. That value can then be used inside the method to control its execution.



4.10 Problem Set

Written Exercises

4.1 Evaluate the following Boolean expressions for a `Robot`. Assume the robot is on intersection (1, 5) facing north. There is a `Wall` immediately in front of it. In each case your answer will be either *true* or *false*.

a. `this.getAvenue() > 0`

b. `this.getAvenue() <= 5`

c. `this.getStreet() != 1`

d. `!(this.getStreet() == 1)`

e. `this.frontIsClear()`

f. `!this.frontIsClear()`

g. `!!this.frontIsClear()`

h. `this.frontIsClear() == false`

4.2 Consider the following `if-else` statements. Do they behave the same way or differently? Justify your answer.

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

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

4.3 Consider the `goToNextRow` method developed in Section 4.3.3 and shown in Listing 4-3.

a. In a table similar to Table 1-2, trace the method when the robot is on (1, 1) facing west, again when it is on (1, 3) facing west, and once again when the robot is on (1, 5) facing east.

b. Describe what happens if the method is modified for rows of length 1. That is, the west end of the row is on Avenue 1 and the east end is also on Avenue 1.

c. Rewrite the method using an `if-else` statement.

d. The current method requires the rows to start and end on specified avenues. Rewrite the method using a different test to remove this restriction. The new method will allow the method to be used in any size field without modification.

4.4 Figure 4-13 illustrates the execution of a `while` loop. Trace the loop using a table similar to Table 1-2. Include columns for the robot's street, avenue, direction, and the parameter `howFar`. Assume the robot begins on (2, 5) facing east and that the `step` method was called with an argument of `4`.

Programming Exercises

4.5 Write methods named `turnLeft`, `pickThing`, and `putThing` that allow the client to specify how many times the robot turns, picks, and puts, respectively.

4.6 Write a pair of methods, as follows:

a. `carryExactlyEight` ensures a robot is carrying exactly eight things in its backpack. Assume the robot is on an intersection with at least eight things that can be picked up

b. Generalize `carryExactlyEight` to `carryExactly`. The new method will take a parameter specifying how many `Things` the robot should carry.

4.7 Write a new robot method, `faceNorth`. A robot that executes `faceNorth` will turn so that `getDirection` returns `Direction.NORTH`.

a. Write `faceNorth` so that the robot turns left until it faces north. Use several `if` statements.

b. Write `faceNorth` so that the robot turns left until it faces north. Use a `while` statement.

c. Write `faceNorth` so that the robot turns either left or right, depending on which direction requires the fewest turns to face north.

4.8 Write a robot method named `face`. It takes a single direction as a parameter and turns the robot to face in that direction. The robot does *not* need to use the minimal number of turns.

4.9 Code and run brief examples of the following errors and report how your compiler handles them.

a. A method with a return type of `void` that includes the statement `return !this.canPickThing();`.

b. A method with a return type of `boolean` that does not include a `return` statement.

c. A method named `experiment` that takes a single integer argument. Call it without an argument, with two arguments, and with `Direction.NORTH`.

Programming Projects

4.10 Finish the implementation of the `Racer` class shown in Listing 4-4. Demonstrate your class with at least two different race courses.

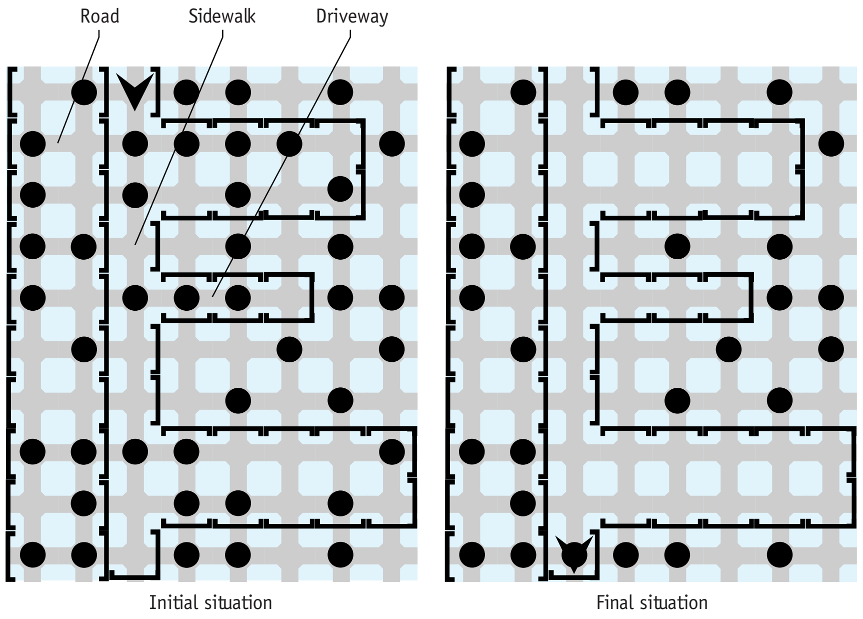
4.11 Listing 3-3 is the complete implementation of the `Harvester` class, a class of robots designed to harvest a field of things. Implement the class again using your knowledge of `if` and `while` statements. Your new class of robots should be able to harvest a rectangular field of any size provided that things cover the field completely and the field is bordered by intersections that do not have any things on it (in particular, there are no walls bordering the field). Note that the original solution required the field to have an even number of rows. Your solution should not have that restriction. Assume the upper-left corner of the field is at (1, 1). Demonstrate your robot harvesting at least two fields with different sizes.

4.12 `karel` and `tina`, instances of `ShovelBot`, are in business together as snow shovelers. `karel` shovels the snow (`Things`) from the driveways, placing them on the sidewalk. Then, while `karel` rests, `tina` moves all the snow left on the sidewalk to the end of the sidewalk.

An initial situation with its corresponding final situation is shown in Figure 4-18. It is known that `karel` and `tina` always start at one end of the sidewalk. The sidewalk always extends beyond the first and last driveways, but it is not known how many driveways there are, the width of the driveways, or the length of the driveways.

(figure 4-18)

Neighborhood that requires snow to be shoveled



4.13 Implement a `Guard` class of robots that can guard either King Java's castle or King Caffeine's castle, plus other castles with similar layouts but different sizes. See Programming Projects 3.12 and 3.13 for descriptions of these castles. You may assume that the corner turrets are each one wall square and that the central courtyard of the castle has at least one wall on each side. Create several files specifying different sizes of castles to use in testing your program.

4.14 A method named `goToOrigin` could use the following algorithm to move a robot to intersection (0, 0):

```
face avenue 0
move to avenue 0
face street 0
move to street 0
```

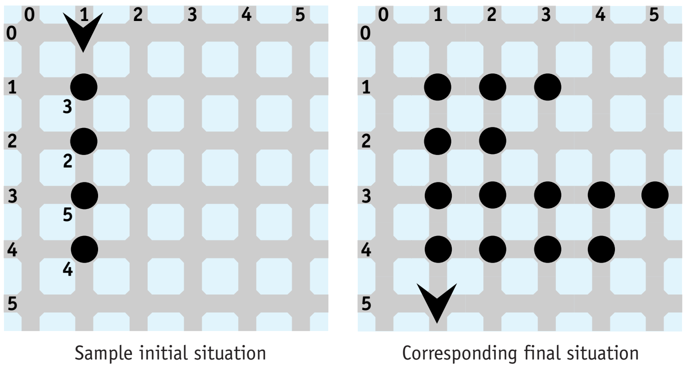
Assume the city has no obstructions such as `Walls`.

a. Implement `goToOrigin` using the given algorithm.

b. Write a method named `goTo` that allows the programmer to specify the intersection the robot is to go to.

4.15. Suppose that data from a poll is represented by `Thing` objects. There is one `Thing` object on intersection (1, 1) for each person who selected response "a". Two people selected response "b" in the poll, resulting in two `Thing` objects on intersection (1, 2). Similarly, the number of objects on the remaining intersections represents the number of people selecting particular responses.

Write `HistogramBot` to create a histogram (commonly called a bar chart) for the data. An instance of `HistogramBot` will pick up the things on each intersection and spread them out (one per intersection) to form a bar. Sample initial and final situations are shown in Figure 4-19. Test your program with different numbers of things on each pile as well as with different numbers of piles.



(figure 4-19)

Sample pair of initial and final situations for a `HistogramBot`

16. Sketch a scene on graph paper that uses a combination of ovals, rectangles, lines, and perhaps strings (text). Write a program that paints your scene, using the scaling techniques in Section 4.7.