The Lay of the Land

F

Appendix F can be used to either look ahead to what you will be learning, or as a review for what you have learned. This appendix provides a view of the lay of the land. It's like standing on a high hill at the beginning of a hike to see what kind of interesting places lie ahead – or surveying the view at the end of the hike to see where you've been.

If you are at the beginning of your hike and looking ahead, this chapter gives a glimpse of nine interesting places in the journey to becoming an object-oriented programmer. By getting an early view of them, you will know a little bit of what is to come, and be better able to integrate what you are learning into a cohesive whole. And, as any hiker knows, trudging up a mountain is easier if you can look forward to what lies at the top and on the other side. Obviously, there will be lots of details clarified later.

If you are at the end of your hike and looking backwards, this appendix can help you remember where all you've been and what you've learned. It could be used, for example, as part of your review before an exam.

Chapter Objectives

- To see how to extend classes with new services.
- To see how to perform a task repeatedly.
- To see how programs can decide whether to take an action.
- To see how to remember information temporarily.
- To see how to make services more flexible by receiving information when they are executed.
- To see how to interact with the program's user.
- To see how to remember information for an object's lifetime.
- To see how to send the same message to different kinds of objects, with each object behaving in a way appropriate for itself.
- To see how to gather similar information together with a collection.

F.1 Extending an Existing Class

Suppose that `karel` is working for a construction company that is paving several streets running east and west. At each end of the construction site are walls, blocking traffic from using the streets. Traffic continues to cross the construction site on the avenues, however. To warn the cross-traffic to slow down, flashers are placed on each intersection every night. `karel`, `jasmine`, and `pat` have the job of collecting them again in the morning. One robot begins on the west end of each street being paved. They each proceed to the east end, collecting the flashers along the way. When they have collected them all, they turn around. Their initial and final situations are shown in Figure F-1 and Figure F-2.

The flashers shown in Figure F-1 are special kinds of `Thing` objects that have a special appearance. They also have two additional services, one to turn on the flashing light and another to turn it off.

Figure F-1: The initial situation for picking up flashers on a construction project.

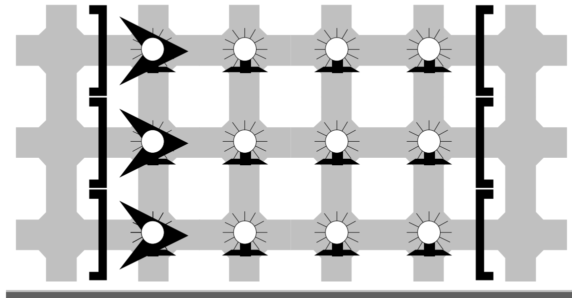
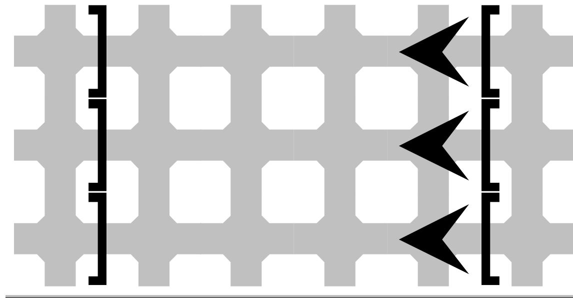


Figure F-2: The final situation.



F.1.1 Using Existing Techniques

Solving this problem is not difficult, but it is long and repetitive. The program has six lines of code placing walls, and twelve placing flashers. The instructions to each robot are nine lines long, and are repeated three times, once for each robot. Each set of instructions is the same except for the name of the robot. Much of the program is shown in Listing F-1. Some repetitive lines are omitted.

One of the primary tasks of programming is to find better abstractions for programs such as these, so they become easier to read, write, and understand.

Listing F-1: Much of the program to collect flashers from the construction site.

Find The Code: layofland/extend

```
 1 import becker.robots.*;
 2
 3 public class Main extends Object
 4 {
 5   public static void main(String[] args)
 6   { City site = new City();
 7
 8     // create the walls at the end of each street's construction zone
 9     Wall detour0 = new Wall(site, 0, 1, Directions.EAST);
10     Wall detour1 = new Wall(site, 0, 2, Directions.EAST);
11     Wall detour2 = new Wall(site, 0, 3, Directions.EAST);
12     Wall detour3 = new Wall(site, 5, 1, Directions.WEST);
13     Wall detour4 = new Wall(site, 5, 2, Directions.WEST);
14     Wall detour5 = new Wall(site, 5, 3, Directions.WEST);
15
16     // create 12 flashers, four on each street
17     Flasher flash00 = new Flasher(site, 1, 1, true);
18     Flasher flash01 = new Flasher(site, 2, 1, true);
19     Flasher flash02 = new Flasher(site, 3, 1, true);
20     Flasher flash03 = new Flasher(site, 4, 1, true);
21     Flasher flash04 = new Flasher(site, 1, 2, true);
       ... // seven similar lines omitted

29
30     // create the three robot workers
31     Robot karel = new Robot(site, 1, 1, Directions.EAST);
32     Robot jasmine = new Robot(site, 1, 2,Directions.EAST);
33     Robot pat = new Robot(site, 1, 3, Directions.EAST);
34
35     // display the construction site
36     CityFrame frame = new CityFrame(site, 6, 5);
37
38     // tell each robot what to do
30     karel.pickThing();
40     karel.move();
41     karel.pickThing();
42     karel.move();
43     karel.pickThing();
44     karel.move();
45     karel.pickThing();
46     karel.turnLeft();
47     karel.turnLeft();
48
49     jasmine.pickThing();
50     jasmine.move();
       ... // Seven instructions to jasmine, similar to lines 42-48, omitted.
           // Nine instructions to pat omitted.
68   }
69 }
```

F.1.2 Creating A New Kind of Robot

This program would be so much easier to understand if the robots could do more than just pick things up, move, turn left, and put things down. For example, what if we had a new kind of robot that can also collect a row of flashers, and turn around? With this new kind of robot, nine lines of instruction for each robot could be reduced to two:

```
karel.collectFlashers();
karel.turnAround();
```

Besides making the main program shorter and easier to understand, defining a new kind of robot also allows us to reuse code. Not only can `karel` use these two services, but so can `jasmine` and `pat`.

Define a new kind of robot with new services for long or complex tasks, or tasks that need to be done several times.

We can and should create a new kind of robot with new services whenever a complex task has distinct parts or the same task is performed several times, either by the same robot or several robots.

Listing F-2 shows how to define a new kind of robot named `Collector` with two new services: `collectFlashers`, and `turnAround`. This code should be in its own file, named `Collector.java`.

Each of the two new services, defined in lines 9-18 and 20-24, follow a regular pattern: the keywords `public` and `void` are followed by the name of the service and a pair of parentheses. After this, between { and }, are the instructions that tell the robot how to carry out the new command.

When defining a new service, use `this` instead of the name of a robot.

When a robot named `karel` is told to move, we name the robot and then tell it to move – for example, `karel.move()`. This approach doesn't work inside a new kind of robot like `Collector`, because we don't know what the robot will be named. It could be named `karel`, `jasmine`, `pat`, or something else. So we tell "this robot" to move by writing `this.move()`.

The name of the class, `Collector`, must be used to construct these new kinds of robots. Replace lines 31, 32, and 33 in Listing F-1 with

```
Collector karel = new Collector(
             site, 1, 1, Directions.EAST, 0);
Collector jasmine = new Collector(
             site, 1, 2, Directions.EAST, 0);
Collector pat = new Collector(
             site, 1, 3, Directions.EAST, 0);
```

`karel`, `jasmine` and `pat` have all the capabilities of ordinary robots. They can move, turn left, pick things up, and put things down. They can also collect a row of four flashers and turn around, thanks to the definitions contained in Listing F-2. And so, we can replace lines 39 to 47 in Listing F-1 with just two lines:

```
karel.collectFlashers();
karel.turnAround();
```

A similar replacement can be made for `jasmine` and `pat`.

Listing F-2: The definition of a new kind of robot that has two new services for collecting flashers and turning around.

Find The Code: layofland/extend

```
 1 import becker.robots.*;
 2 
 3 public class Collector extends Robot
 4 {
 5   public Collector(City city, int ave, int str, int dir)
 6   { super(city, ave, str, dir);
 7   }
 8 
 9   // Collect a row of four flashers
10   public void collectFlashers()
11   { this.pickThing();
12     this.move();
13     this.pickThing();
14     this.move();
15     this.pickThing();
16     this.move();
17     this.pickThing();
18   }
19 
20   // turn around and face the opposite direction
21   public void turnAround()
22   { this.turnLeft();
23     this.turnLeft();
24   }
25 }
```

F.1.3 Creating Other New Kinds of Robots

Looking back to Chapter 1, we could have used new kinds of robots many times. In one end-of-chapter problem the robot moved around a square. It could have used a new service named `moveAlongSide`. The robot in many of the problems could have used a new service named `turnRight`, and in another problem two different robots could have each used `move3`. In each of these situations, one or more robots performed the same sequence of instructions several times, or a complicated action could have been broken down into several steps.

New kinds of robots, customized for these situations, can be defined with the template shown in Listing F-3. Most of the code is the same for every new kind of robot. You need to replace *«className»*, and *«newService»* to customize the template for your unique needs. *«className»* follows rules

that we learned in Chapter 1. Use the same name both places that *«className»* appears.

Listing F-3: A template for creating a new kind of robot with a new service. Additional services may also be added.

```
 1 import becker.robots.*;
 2
 3 public class «className» extends Robot
 4 {
 5   public «className»(City city, int ave, int str, int dir)
 6   { super(city, ave, str, dir);
 7   }
 8
 9   «newService»
10 }
```

In fact, this same technique can also be used to create a new kind of city that has new services to place walls and flashers for the construction site. The new kind of city might be called a `ConSite`, short for "construction site." It has two new services, one for placing barriers (walls) and another for placing flashers.

Using a `ConSite` city and `Collector` robots significantly shortens our `main` program, making it easier to read and understand. The revised program is shown in Listing F-4. This program behaves exactly the same as the 69 line program shown in Listing F-1, but is much simpler and easier to read.

Find The Code:
layofland/extend

Listing F-4: A new version of the program using a new kind of robot and a new kind of city.

```
 1 import becker.robots.*;
 2
 3 public class NewMain extends Object
 4 {
 5   public static void main(String[] args)
 6   { ConSite site = new ConSite();
 7     site.placeBarriers();
 8     site.placeFlashers();
 9
10     Collector karel =
11               new Collector(site, 1, 1, Directions.EAST);
12     Collector jasmine =
13               new Collector(site, 1, 2, Directions.EAST);
14     Collector pat =
15               new Collector(site, 1, 3, Directions.EAST);
16
17     CityFrame frame = new CityFrame(site, 6, 5);
18
```

continued…

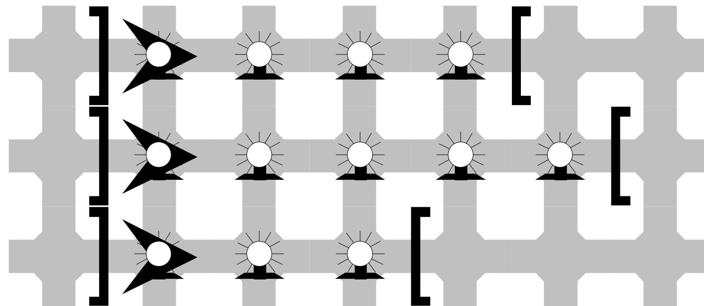
Listing F-4 *continued*

```
19      karel.collectFlashers();
20      karel.turnAround();
21
22      jasmine.collectFlashers();
23      jasmine.turnAround();
24
25      pat.collectFlashers();
26      pat.turnAround();
27    }
28  }
```

F.2 Repeating Statements

The paving project where `karel`, `jasmine`, and `pat` work has been progressing smoothly. However, an influential resident on `jasmine`'s street has convinced city council to pave one extra block of that street. To offset the cost, only two blocks of `pat`'s street will be paved. The new situation is shown in Figure F-3.

Figure F-3: Collecting flashers on streets of differing lengths.



Now we have a problem – we apparently need three different kinds of robots. `karel`, the top robot in Figure F-3, can still be a `Collector`, as we defined in Listing F-2. `jasmine`, the middle robot, must collect the flasher from an extra intersection. `pat`, the bottom robot, will malfunction unless we instruct it to collect flashers from only three intersections rather than four.

Creating three different kinds of robots that do almost the same task is a poor solution. Fortunately, there is a better way. We know that each robot has completed its task when it reaches the wall at the opposite end of the street. In between the starting position and that wall, each robot does the same steps over and over: collect a flasher, and move to the next intersection.

Many tasks do the same steps over and over, until the task is finished.

If we can define the `Collector` robots this way, then `karel`, `jasmine`, and `pat` can all be the same kind of robot.

Java's `while` statement can be used to repeat statements over and over. The `while` statement can be used whenever a task is composed of identical steps that are repeated until the task is done. In this case, the identical steps are collecting a flasher and moving to the next intersection. These steps are repeated until the opposite wall is reached. Using this algorithm, each of the three robots will perform correctly even though they are collecting flashers on different lengths of street.

A version of `collectFlashers` that uses this idea is shown in Listing F-5. The `while` statement extends from line 11 to line 14 and consists of three parts.

- The keyword `while` signals to Java that something is going to be repeated.
- The *condition* determines if the statements should be repeated again. In this example, the condition is `(this.frontIsClear())` – is this robot's front clear of anything that can prevent it from moving (like a wall)?
- The *body* of the `while` statement, the part between { and }, is the code that is repeated.

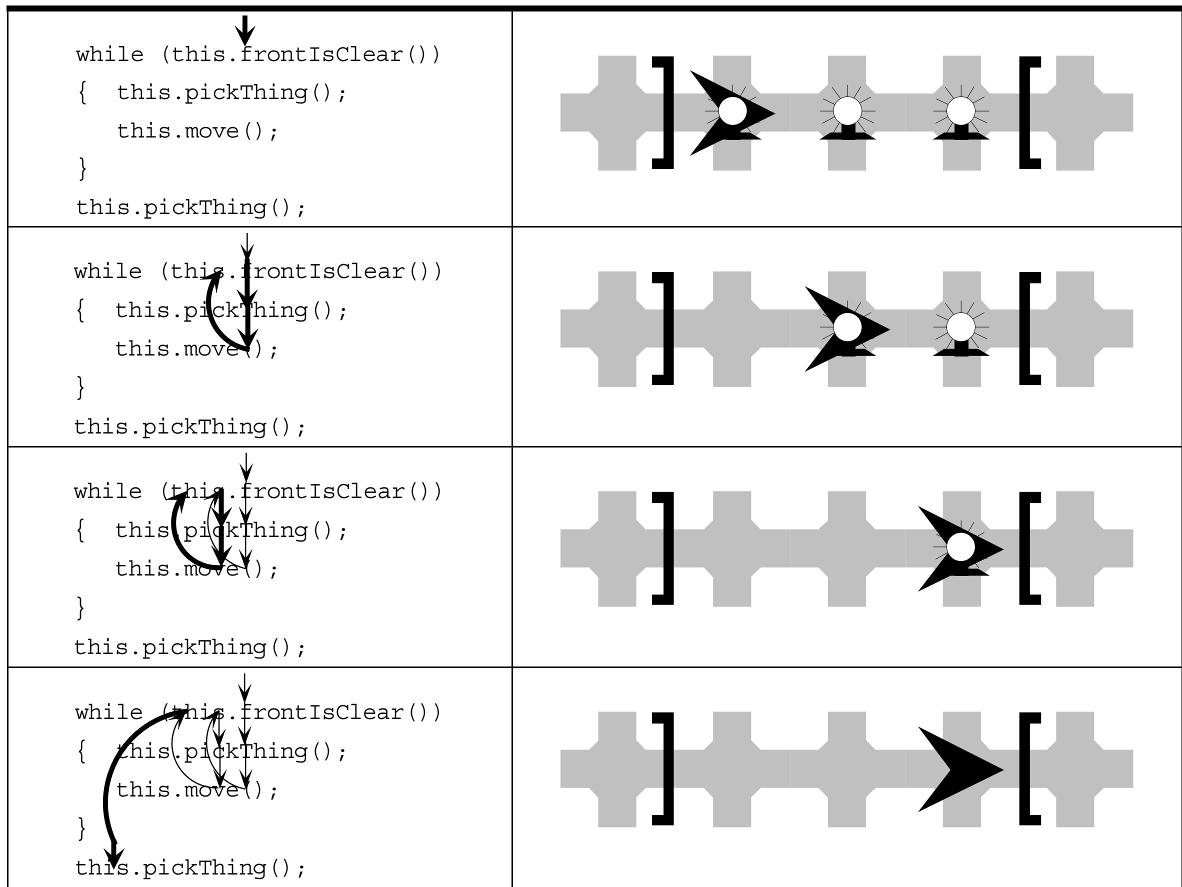
Find The Code: layofland/repetition

Listing F-5: The `Collector` class, defined with a `while` loop in `collectFlashers`.

```
 1 import becker.robots.*;
 2
 3 public class Collector extends Robot
 4 {
 5   public Collector(City city, int ave, int str, int dir)
 6   { super(city, ave, str, dir);
 7   }
 8
 9   // Collect flashers as long as the front of the robot is not blocked.
10   public void collectFlashers()
11   { while (this.frontIsClear())
12     { this.pickThing();
13       this.move();
14     }
15     this.pickThing();
16   }
17
18   // turn around and face the opposite direction
19   public void turnAround()
20   { this.turnLeft();
21     this.turnLeft();
22   }
23 }
```

How does this `while` loop work? When a robot is told to `collect-Flashers`, the `while` loop first checks if the robot's front is clear. That is, it checks if it is blocked from moving forwards. If its front is clear, then it does everything in the braces at lines 12-14. After it picks up a flasher and moves to the next intersection, execution returns to line 11. The robot again checks if its front is clear. If it is, everything in the braces is executed. This keeps happening – check if the front is clear, and if it is, do everything in the braces – until the front is *not* clear. Then execution resumes at line 15 – the line following the `while` loop's closing brace. Figure F-4 illustrates this process.

Figure F-4: Illustrating the execution of a `while` loop. The dark lines in the code indicate the statements that are executed to arrive at the situation shown on the right.



The `while` loop contains one `pickThing` instruction and one `move` instruction, so the robot will always pick something up just as often as it moves. However, the initial situation shows that it needs to move twice but pick up

three flashers. Thus, there must be one `pickThing` instruction after the loop to ensure that the extra flasher is picked up.

Use a `while` loop to execute the same code an unknown number of times.

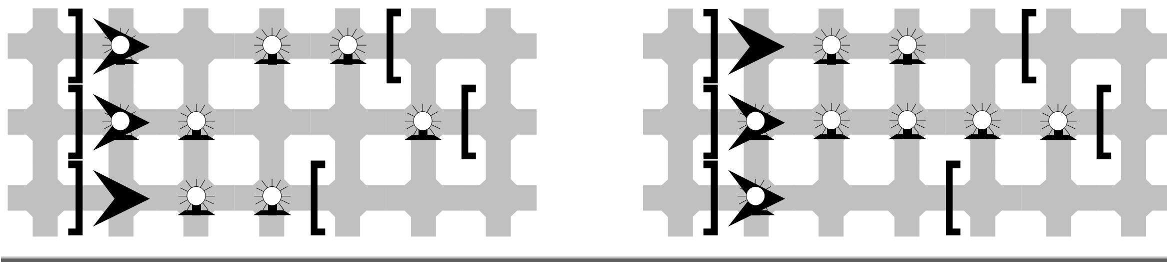
A `while` loop is useful to repeat some code over and over. In this example, the repeated code picked something up and moved. A robot could also use a `while` loop to move until a streetlight is encountered, to pick up all the flashers on a corner, to turn left until its front is clear, and so on.

F.3 Making Decisions

Work has been proceeding nicely on the construction site. `karel`, `jasmine`, and `pat` are still collecting the flashers every morning. One morning, however, they were unable to complete their jobs. It appears that several flashers were stolen during the night. The robots malfunctioned when they reached the empty intersections and tried to pick up the missing flashers. The construction project is already over budget, and so the decision has been made to distribute the eight remaining flashers randomly on the intersections.

Figure F-5 shows just two of the many possible initial situations. As you can see, the robots can no longer automatically try to pick up a flasher from every intersection. They must be reprogrammed to first check if a flasher is present.

Figure F-5: Two possible initial situations.



A robot may need to make decisions in other contexts, as well. It may need to detect whether its way is blocked by a wall. If so, turn. It may need to check whether it is facing south, and if it is, turn around. It may need to check whether there are enough things in its backpack to carry out a job. If not, go to a supply depot and get some more.

Use an `if` statement to choose whether or not to execute some code, based on a condition.

In each of these contexts, the robot should use an `if` statement. An `if` statement tests a condition. If the condition is true, some additional code is executed. If the condition is false, the additional code is skipped. For example, in

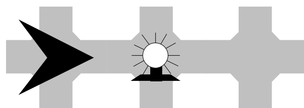
```
if (karel.isBesideThing())
{ karel.pickThing();
}
```

the additional code is `karel.pickThing()` and the condition is `karel.isBesideThing()`. If the condition is true – `karel` is, in fact, beside a `Thing` (a flasher) – then the additional code is executed and the thing is picked up. If `karel` is not beside a `Thing`, then `karel` won't even try to pick something up.

Figures F-6 and F-7 show two different initial situations and how `karel` behaves when the code shown is executed. In each case, arrows show how execution proceeds through the code.

Figure F-6: When there is a thing to pick up.

Initial Situation:



Code Executed:

```
karel.move();
if (karel.besideThing())
{   karel.pickThing();
}
karel.move();
```

Final Situation:

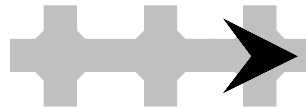
Figure F-7: When there isn't a thing to pick up.

Initial Situation:

Code Executed:

```
karel.move();
if (karel.besideThing())
{   karel.pickThing();
}
karel.move();
```

Final Situation:



Now, we need to apply this knowledge to keep `karel`, `jasmine`, and `pat` from malfunctioning when they do their jobs. The code we need to fix is the `collectFlashers` service in Listing F-5. Each use of `this.pickThing()` must be replaced with three lines:

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

We again use this instead of a robot's name because we are defining a new kind of robot that might be given the name karel, jasmine, pat – or a completely new name.

Checking whether or not a robot is beside a thing is just one use of the if statement. It is useful in many situations, wherever a program must determine whether or not some code should be executed. Use it to test whether or not a robot should pick something up, or whether or not something should be put down. Eventually, we will use the if statement to test whether or not a credit card's balance is low enough to make a debit, or whether or not a name is too long to print in the allotted space, to give just a few examples.

The if statement and the while statement are sometimes confused by beginning programmers. Both include a test, but they are used for fundamentally different things. A while statement tests if code should be executed again. The code in the braces might be executed many, many times. An if statement tests whether to execute code exactly once or not at all.

F.3.1 Testing for Specific Kinds of Things

The test for being beside a Thing is very general. Flashers are Things, but so are ordinary Things! In addition, the techniques we used in Section G.1 to extend the Robot class can also be used to create new kinds of Things.

What if some other kinds of Things are on the construction site? For example, suppose one of the intersections has a "tool" (represented by a Thing), but no flasher. Then, when karel comes to that intersection, karel first tests if it is beside a Thing. The "tool" will cause the condition to be true, and karel will pick it up.

To solve this problem, we can write

```
if (this.isBesideThing(Predicate.aFlasher))
{ this.pickThing(Predicate.aFlasher);
}
```

The new part, Predicate.aFlasher, tells isBesideThing and pickThing that we are only interested in Things that happen to be Flashers. isBesideThing should only test if the robot is beside a Flasher, and pickThing should only attempt to pick up Flashers. This restriction to Flashers only applies to the isBesideThing and pickThing where Predicate.aFlasher is included. It is not "remembered" for the next time.

F.3.2 Helper Methods

The part of `collectFlashers` that picks flashers up is becoming more complicated. It now includes a test to check whether the robot is actually beside a `Thing`, and whether that `Thing` is, in fact, a flasher. Replacing the simple statement `this.pickThing()` in Listing F-5 with the three lines shown above obscures the logic of the `while` loop.

One way to make `collectFlashers` easier to understand is to create another service to handle the complexity of picking up a flasher. This new service might be named `pickFlasherIfPresent`. Services that exist just to simplify another method are sometimes called *helper methods*.

A helper method often makes code easier to understand.

Listing F-6 shows a version of the `Collector` class that defines `pickFlasherIfPresent`. `collectFlashers` uses the method with the statement

```
this.pickFlasherIfPresent();
```

By defining and using this helper method, the intricacies of picking up a flasher only need to be written once instead of twice (once in the `while` loop and once more right after the loop). It also retains the original simplicity of the `collectFlashers` method.

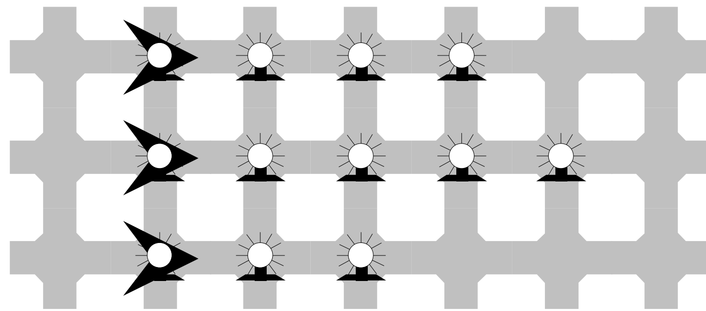
Listing F-6: One method, `pickFlasherIfPresent`, can help another method, `collectFlashers`, carry out its task.

```
 1 import becker.robots.*;
 2
 3 public class Collector extends Robot
 4 {
 5   public Collector(City city, int ave, int str, int dir)
 6   { super(city, ave, str, dir);
 7   }
 8
 9   // Collect flashers as long as the front of the robot is not blocked.
10   public void collectFlashers()
11   { while (this.frontIsClear())
12     { this.pickFlasherIfPresent();
13       this.move();
14     }
15     this.pickFlasherIfPresent();
16   }
17
18   // pick up a flasher, if one is present on the current intersection
19   public void pickFlasherIfPresent()
20   { if (this.isBesideThing(Predicate.aFlasher))
21     { this.pickThing(Predicate.aFlasher);
22     }
23   }
24
25   // remainder of class omitted
```

F.4 Temporary Memory

The city's public works department has taken the walls from the paving project to another work site. The new situation is shown in Figure F-8.

Figure F-8: The paving project, minus the walls.



This presents a problem. `karel`, `jasmine`, and `pat` had been using the walls to determine when to stop collecting flashers. Without the walls, they will keep going east. At each intersection they will check for a flasher. Not finding one, they will go to the next intersection and check again – forever.

One possible solution is for each robot to count the number of moves it makes. Each robot should move four times, attempting to collect a flasher before each move. Then, collect the last flasher (if there is one) and turn around. A significant disadvantage of this plan is that `karel` and `pat` will have to travel farther than before. We will simply accept that limitation for now.

Variables store or remember information.

To make this plan work, each robot will need to remember how many moves it has made while it is collecting the flashers. Java provides *variables* to store or remember information. A variable is like a box with a name. Each variable stores a piece of information; in this case, a number. The number can be replaced by a new number at any time.

The following code creates a new variable named `numMoves` and stores a number, zero, in it.

```
int numMoves = 0;
```

A different number, in this case five, can be stored in `numMoves` like this:

```
numMoves = 5;
```

Notice that "`int`" is only used the first time, when the variable was created. "`int`" indicates that the variable will store an integer, a certain kind of number.

We can also use variables to calculate new values. For example,

```
int a = 0;
a = numMoves + 1;
```

will first create a new variable named “a”. In the next line, Java will first get the number stored in numMoves (5) and add 1 to it, obtaining 6. This new number is then put into the variable a, replacing the number that was there. The variable on the left side of the equals sign is forced to have the value calculated on the right side of the equals sign.

We can also use the same variable on both the left and the right side of the equals sign. For example,

```
numMoves = numMoves + 1;
```

Variables can be used to count.

gets the current number stored in numMoves (5) and adds 1 to it. This new number, 6, is then stored in the variable named on the left side of the equals sign. That is, numMoves is now one larger than it used to be. This is the fundamental step in counting – remembering a number one larger than the previous number.

Now, we can combine counting with a while loop to move a robot four times, a slight simplification of collecting flashers.

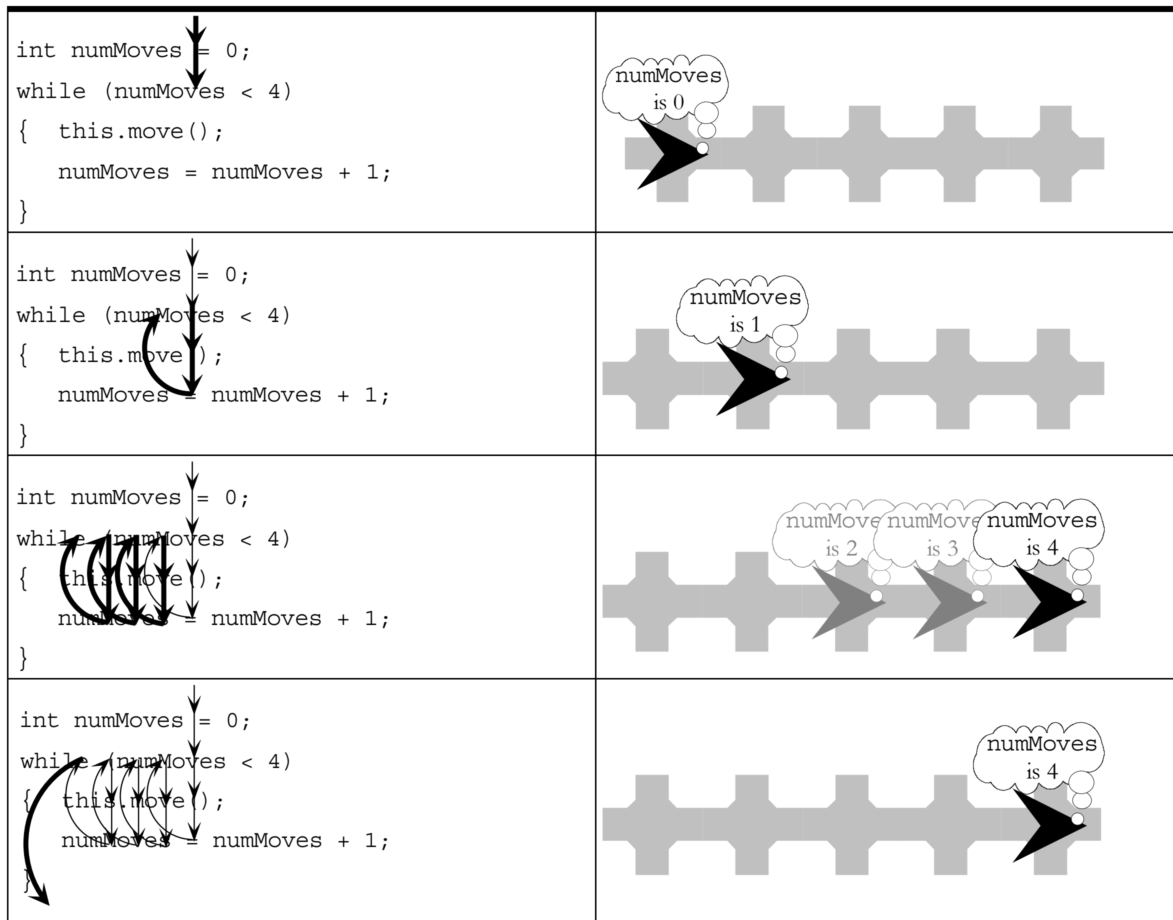
```
public void move4()
{ int numMoves = 0;
  while (numMoves < 4)
  { this.move();
    numMoves = numMoves + 1;
  }
}
```

Figure F-9 illustrates how this code is executed. It begins by creating a variable, numMoves, to store the number of moves the robot has made so far. The robot hasn’t moved yet, so the first value stored is zero. The test in the while loop, numMoves < 4, checks to see if the loop should continue. If the number of moves made so far, numMoves, is less than four, the two statements between the braces are executed. Otherwise, the loop ends. Inside the loop, the robot is told to move and the number of moves is incremented by 1. A while loop used in this way is sometimes called a “counted while loop.”

This particular while loop executes the move instruction four times, but it compares numMoves to 4 a total of five times. In Figure F-9, every time an arrow points to the line while (numMoves < 4) the comparison is made. The first four times it is true that numMoves is less than 4. The last time, however, numMoves has the value 4, and the loop ends.

To solve the flasher collection problem for `karel`, `jasmine`, and `pat`, the code shown in Figure F-9 must also pick up a flasher just before the robot moves, if one is present and again after the loop exits. This change is shown in Listing F-7. This version of `collectFlashers` always moves four times and checks five intersections for flashers. To understand why, look again at Figure F-9 – the robot moves four times but visits five intersections.

Figure F-9: Illustrating the execution of a counted while loop.



Listing F-7: A version of `collectFlashers` that always checks exactly five intersections.

```
1 public void collectFlashers()
2 { int numMoves = 0;
3   while (numMoves < 4)
4   { this.pickFlasherIfPresent();
5     this.move();
6     numMoves = numMoves + 1;
7   }
8   this.pickFlasherIfPresent();
9 }
```

When a variable is defined inside a method it is called a *temporary variable*. It may be used only within the method while the method is executing. When the method is finished, the temporary variable and the information it contains is discarded.

Temporary variables are discarded when the method containing them is finished executing.

A temporary variable may be used to control a `while` loop any time you know how many times a set of steps should be repeated. This is quite different from the way the `while` loop was used in Section F-2. There the steps were repeated over and over until the task was done. We didn't know, when we started out, how many times the loop would repeat before we reached a wall and stopped.

Use a counted loop when the number of repetitions is known in advance.

Examples that effectively combine a temporary variable with a `while` loop include:

- moving a robot around the four sides of a square.
- picking up exactly 100 flashers,
- running 10 laps around a track.
- counting the number the things on an intersection, replacing half of them.

In each example, a set of steps is repeated a known number of times.

The last example is interesting because it uses two loops, with the information gathered in the first used to control the second. Listing F-8 shows one way to solve the problem:

Listing F-8: Picking up half of the things on an intersection.

```
 1 public void pickHalf()
 2 { int numThingsFound = 0;
 3   while (this.isBesideThing())
 4   { this.pickThing();
 5     numThingsFound = numThingsFound + 1;
 6   }
 7
 8   int numPutBack = 0;
 9   while (numPutBack < numThingsFound/2)
10   { this.putThing();
11     numPutBack = numPutBack + 1;
12   }
13 }
```

The first loop, at lines 3-6, is not a counted loop. We don't know how many times it will execute. Rather, it repeats while there is still something on the intersection, counting the number of things it picks up. This count is kept in a temporary variable named `numThingsFound`.

The second loop, at lines 9-12, is a counted loop. It divides the number of things found by 2 to calculate how many times to execute. As long as `numPutBack` is less than this number, another thing is put down and the count of things put down is incremented by one.

Temporary variables have uses beyond controlling loops.

Remembering information temporarily in a variable is useful beyond controlling loops. In the last example, the robot remembered how many things were on the intersection so it could put back half of them. It might remember which avenue and street it was on before beginning a task so it can return there when it's done, or the direction it's facing so it can turn that way again.

F.5 More Flexible Methods

The manager of the construction company has become concerned about the solution just implemented. Recall that when the walls were in place, `karel`, `jasmine`, and `pat` each traveled only as far as needed to collect the flashers on their assigned street. Now, each robot checks five intersections even though `karel` is on a street where only four need to be checked, and `pat` really only needs to check 3 intersections. The manager is concerned that `karel` and `pat` will wear out faster than necessary. She would like a way to tell each robot how many intersections to check for flashers.

What the manager wants is a more flexible version of the service developed in the previous section. There, we developed a service that always moved four times, checking a total of five intersections for flashers to collect. Instead of always checking five intersections, the manager should be able to

tell each robot how many intersections to check. This would be done in the `main` method. In Listing F-4 we wrote

```
16      karel.collectFlashers();
17      karel.turnAround();
18
19      jasmine.collectFlashers();
20      jasmine.turnAround();
21
22      pat.collectFlashers();
23      pat.turnAround();
```

We would like to replace these lines with statements that specify the number of intersections the robot should check. For example,

```
16      karel.collectFlashers(4);
17      karel.turnAround();
18
19      jasmine.collectFlashers(5);
20      jasmine.turnAround();
21
22      pat.collectFlashers(3);
23      pat.turnAround();
```

With this change, the `main` method tells `karel` to check four intersections. Similarly, `jasmine` and `pat` are told to check five and three intersections, respectively.

Implementing this ability requires communicating the 3, 4 or 5 (or any other number) from the place where `collectFlashers` is called to the place where the number is used – the definition of `collectFlashers`. Adding a *parameter* to `collectFlashers` facilitates this communication.

Parameters are used to give additional information required to complete a job.

We need to make some minor modifications to the definition of `collectFlashers` to receive the parameter given in `main`. The new version is shown in Listing F-9. The changes from the previous version, described on page 485, are in bold.

Listing F-9: A more flexible version of `collectFlashers` that uses a parameter.

```
 1 public void collectFlashers(int numIntersections)
 2 { int numMoves = 0;
 3   // move one fewer times than there are intersections
 4   while (numMoves < numIntersections - 1)
 5   { this.pickFlasherIfPresent();
 6     this.move();
 7     numMoves = numMoves + 1;
 8   }
 9   this.pickFlasherIfPresent();
10 }
```

numIntersections is like a temporary variable that is automatically assigned a value just before collectFlashers begins to execute. The value it is assigned is the value given between the parentheses when collectFlashers is called. When we write karel.collectFlashers(4), numIntersections will be given the value four. When we write jasmine.collectFlashers(5), then numIntersections will be given the value five.

To check five intersections, the robot needs to move four times – one less than the number of intersections. This observation implies that the loop should execute numIntersections - 1 times – one fewer intersections than specified by the manager. The while loop's test at line 3 takes this into account.

A complete listing of the class containing main is given in Listing F-10. It uses a new kind of City, shown in Listing F-11. This class uses the same technique used to extend a Robot to add a useful service to a City: placing the flashers. Finally, a complete listing of the Collector robot is given in Listing F-12.

The parameter added in these listings gives collectFlashers a tremendous amount of flexibility. This simple change shifts the service from always checking five intersections to being able to check any number of intersections. You can probably imagine how parameters could be used to make a service that can move a robot any distance or a service that can pick a specified number of things. Eventually, you will be able to write a goto service that directs a robot to go to a particular intersection, no matter where in the city it is (provided nothing blocks it).

Listing F-10: A `main` method calling services with parameters.

Find The Code:
layofland/parameters

```
1  import becker.robots.*;
2
3  public class Main extends Object
4  { public static void main(String[] args)
5    { ConSite site = new ConSite();
6      site.placeFlashers();
7
8      Collector karel =
9               new Collector(site, 1, 1, Directions.EAST);
10     Collector jasmine =
11              new Collector(site, 1, 2, Directions.EAST);
12     Collector pat =
13              new Collector(site, 1, 3, Directions.EAST);
14
15     CityFrame frame = new CityFrame(site, 7, 4);
16
17     karel.collectFlashers(4);
18     karel.turnAround();
19
20     jasmine.collectFlashers(5);
21     jasmine.turnAround();
22
23     pat.collectFlashers(3);
24     pat.turnAround();
25   }
26 }
```

Listing F-11: A new kind of city with a service to place flashers on the construction site.

Find The Code:
layofland/parameters

```
1  import becker.robots.*;
2
3  public class ConSite extends City
4  { public ConSite()
5    { super();
6    }
7
8    public void placeFlashers()
9    { Flasher flash01 = new Flasher(this, 2, 1, true);
10     Flasher flash03 = new Flasher(this, 4, 1, true);
11     Flasher flash04 = new Flasher(this, 1, 2, true);
12     Flasher flash06 = new Flasher(this, 3, 2, true);
13     Flasher flash07 = new Flasher(this, 5, 2, true);
14     Flasher flash08 = new Flasher(this, 1, 3, true);
15     Flasher flash10 = new Flasher(this, 3, 3, true);
16   }
17 }
```

Listing F-12: A new kind of robot that can collect flashers from a specified number of intersections.

Find The Code:
layofland/parameters

```
 1 import becker.robots.*;
 2 
 3 public class Collector extends Robot
 4 {
 5   public Collector(City city, int ave, int str, int dir)
 6   { super(city, ave, str, dir, numThings);
 7   }
 8 
 9   public void collectFlashers(int numIntersections)
10   { int numMoves = 0;
11     // move one fewer times than there are intersections
12     while (numMoves < numIntersections - 1)
13     { this.pickFlasherIfPresent();
14       this.move();
15       numMoves = numMoves + 1;
16     }
17     this.pickFlasherIfPresent();
18   }
19 
20   // pick up a flasher, if one is present on the current intersection
21   public void pickFlasherIfPresent()
22   { if (this.isBesideThing(Predicate.aFlasher))
23     { this.pickThing(Predicate.aFlasher);
24     }
25   }
26 
27   public void turnAround()
28   { this.turnLeft();
29     this.turnLeft();
30   }
31 }
```

F.6 Asking the User

The manager of the construction company has expressed appreciation for the flexibility gained by adding the parameter to `collectFlashers`, but she is not quite satisfied. As it stands, the `main` method always tells `karel` to collect flashers from four intersections, `jasmine` is always told to collect from five intersections, and `pat` is always told to collect from three intersections. The manager would like to vary the instructions each time the program is run. Sometimes `karel` should collect from two intersections and sometimes from forty – depending on the manager's whim.

We can solve this problem by getting *input* – a number – from the human who runs the program, storing that number in a temporary variable, and then

passing the number to `collectFlashers` via a parameter. These changes all take place within the `main` method.

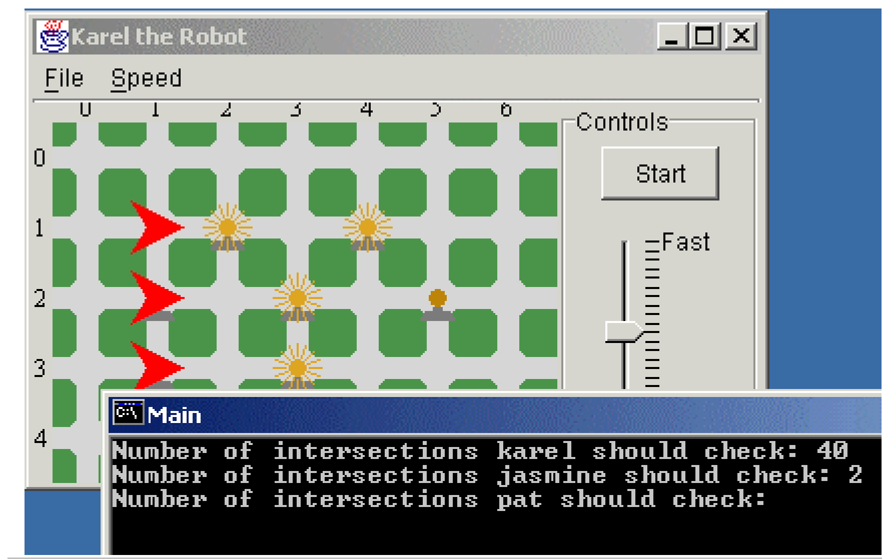
For each piece of information required, we must to do three things:

tell the user what information is being requested
get the information from the user, placing it in a temporary variable
finish getting the current line of information, preparing for the next line

We will repeat these three steps three times, once for each robot. After we have all the information stored in the temporary variables, we can use the variables to tell each of the robots how many intersections to visit.

When the program actually runs, the user will be asked for the information in a separate window, called the *console*, shown in Figure . The user will likely need to click on the console to bring it to the front before it will accept input from the keyboard. After each number is typed, the user should press the "Enter" key. If something other than a number without a decimal point is typed, the program will give an error message and stop.

Figure F-10: Asking the user for information.



Consider the situation shown in Figure F-10 and suppose the user enters 3 for the last number. When the "Start" button is clicked, the top-most robot will collect flashers from 40 intersections, proceeding off the right edge of the display in the process. The middle robot will collect the flasher from (1,2), proceed to (2,2), and turn around. The bottom robot will collect the flashers from three intersections, and then turn around.

So what does the code to do this look like? For each of the three steps listed earlier we would write instructions like this:

```
System.out.print("Number of intersections karel should check: ");
int kNum = in.readInt();
in.readLine();
```

The first line uses a special object, `System.out`, to print a message on the console window. The object's `print` service simply prints the characters that appear between the double quotes in its parameter.

The second line does two things. First, it creates a new temporary variable, `kNum`, the number of intersections `karel` should check for flashers. Then it uses an object named `in` to get a number from the user, putting the number it gets into `kNum`. In Figure F-10 the user has entered the number 40; the `readInt` service gets this number and places it in `kNum`.

In the third line, the `readLine` service is used to process the rest of the line where the user entered the number 40, preparing for the next cycle of asking for information and getting it.

Key Idea: `System.out` is used to show the user textual information.

Key Idea: Create an instance of `TextInput` to obtain textual information from the user.

These three lines of code use two objects, `System.out` and `in`. `System.out` is a special object that is automatically constructed when the program starts. It's primary service is `print`. The other object, `in`, must be constructed by the programmer before it is used. Its construction is similar to the construction of `Robot` or `City` objects except that the name of the class is `TextInput`:

```
TextInput in = new TextInput();
```

The `TextInput` class must be imported before it can be used. Include

```
import becker.io.TextInput;
```

at the beginning of the file using the `TextInput` class.

Finally, the number stored in the temporary variable `kNum` can be given to the `collectFlashers` method with the statement

```
karel.collectFlashers(kNum);
```

This directs `karel` to check 40 intersections for flashers, assuming the interaction shown in Figure F-10.

Listing F-13 shows how to integrate this new code into the `main` method.

Listing F-13: A program which asks the user how many intersections each robot should check.

```
 1 import becker.robots.*;
 2 import becker.io.*;
 3
 4 public class Main extends Object
 5 { public static void main(String[] args)
 6   {
       // This part is the same as lines 6-13 of Listing F-10
16
17     TextInput in = new TextInput();
18
19     System.out.print("Number of intersections karel should check: ");
20     int kNum = in.readInt();
21     in.readLine();
22
23     System.out.print("Number of intersections jasmine should check: ");
24     int jNum = in.readInt();
25     in.readLine();
26
27     System.out.print("Number of intersections pat should check: ");
28     int pNum = in.readInt();
29     in.readLine();
30
31     karel.collectFlashers(kNum);
32     karel.turnAround();
33
34     jasmine.collectFlashers(jNum);
35     jasmine.turnAround();
36
37     pat.collectFlashers(pNum);
38     pat.turnAround();
39   }
40 }
```

Using `System.out` and a `TextInput` object provides many opportunities for interactions between programs and users. A robot program might ask the user where a robot should be placed or how many things to collect. A banking program might ask the user how much money to transfer between accounts or an e-mail program might ask for the address where a message should be sent. In each of these cases, the program tells the user what information is needed, the user types it in, and then the program reads the information, using the services of an object such as `TextInput`.

This style of interaction is simple and easy to implement, but has been superceded in many programs by windows, dialog boxes, buttons, mice, and so on. We will, in time, learn how to interact with users via a graphical user interface as well.

F.7 Objects That Remember

karel was accidentally moved into a wall and has broken beyond repair. Rather than replacing karel with a new robot, the construction company's management has decided that jasmine will collect the flashers from both streets. However, because jasmine and pat are now doing significantly different amounts of work, they must be on different maintenance schedules. Management wants each robot to keep track of how many flashers it has collected, printing out the number at the end of each job.

The main method shown in Listing F-13 must be modified to remove lines 19-21 and 31-32, which refer to karel, and replace lines 34-38 with the following. These statements direct jasmine to

- collect the flashers from one street,
- go to karel's former street and collect flashers there, and,
- print out the number of flashers collected.

pat then collects flashers and also prints out the total number it collected.

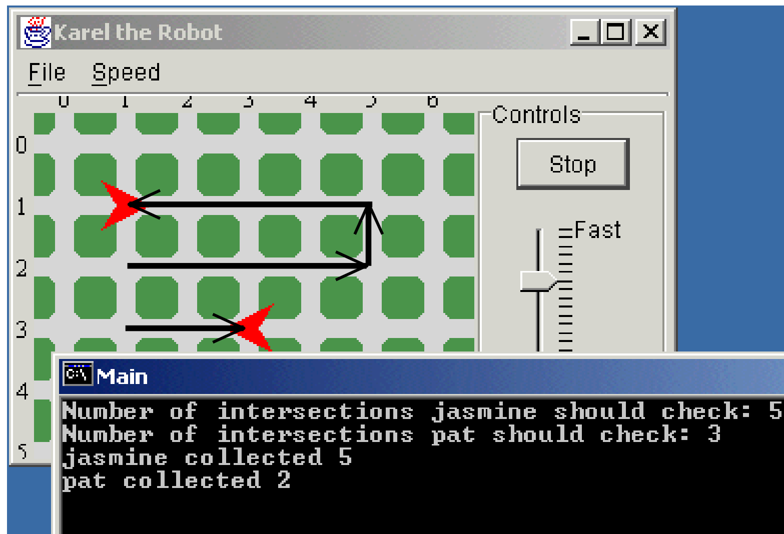
```
jasmine.collectFlashers(jNum);
jasmine.turnLeft();
jasmine.move();
jasmine.turnLeft();
jasmine.collectFlashers(jNum);
jasmine.turnAround();
System.out.print("jasmine collected ");
System.out.println(jasmine.numFlashersCollected());

pat.collectFlashers(pNum);
pat.turnAround();
System.out.print("pat collected ");
System.out.println(pat.numFlashersCollected());
```

Figure F-11 shows the path the robots take when this program is run. It also shows the console window displaying the number of flashers each robot collected.

This program requires two changes to the Collector class. First, it must be modified to remember the number of flashers collected. Second, a query, numFlashersCollected, must be added to retrieve the number so it can be printed.

At first, we may think that all we need is a temporary variable to remember the number of flashers a robot has collected. This will not work, however, because a temporary variable exists only as long as the service containing it executes – then the variable and the value it contained are gone. jasmine, however, will execute collectFlashers twice before we ask for the number of flashers collected. A temporary variable inside collectFlashers could not keep a running total across both uses of the service.

Figure F-11: The result of running the modified program. Arrows show the path each robot took.



An *instance variable*, however, will do what we need. In many ways, an instance variable is like a temporary variable. Both remember information such as a number. Both kinds of variables can have the information they store changed, perhaps by adding one to the number already stored there. The information stored in both kinds of variables can be printed on the console or passed as a parameter to a service.

There are, however, two crucial differences. First, a temporary variable is temporary. It only lasts as long as the method containing it; sometimes even less. An instance variable, on the other hand, lasts as long as the object lasts. An instance variable in `jasmine` will be created when the object is created and will continue to remember information until `jasmine` is no longer needed.

An instance variable remembers information for an object.

Second, a temporary variable can only be used in the method where it is created. An instance variable may be used anywhere in the class containing it.

A temporary variable remembers information while a method executes.

Each object created from the class will have its own instance variable. `jasmine` will have one instance variable and `pat` will have another. This duplication enables `jasmine` and `pat` to each count the flashers they have picked up, each completely independent from the other.

These three properties are crucial to solving the problem of remembering how many flashers have been collected. First, because instance variables last as long as the object, each time `collectFlashers` is called the count can continue from where it left off the previous time. Second, because instance

variables can be used in any method in the class, we can write a separate method that returns the current value of the instance variable. Third, because each object has its own instance variable, `jasmine` and `pat` can each keep track of their own work.

Listing F-14 shows how to use an instance variable named `flasherCount` to remember how many flashers have been collected. The only differences between Listing F-14 and the previous version in Listing F-12 are shown in bold.

Listing F-14: Modifications to `Collector` to remember the number of flashers collected.

```
 1 import becker.robots.*;
 2
 3 public class Collector extends Robot
 4 { private int flasherCount = 0;
 5
 6   public Collector(City city, int ave, int str, int dir)
 7   { super(city, ave, str, dir);
 8   }
 9
10   public void collectFlashers(int numIntersections)
11   { int numMoves = 0;
12     while (numMoves < numIntersections - 1)
13     { this.pickFlasherIfPresent();
14       this.move();
15       numMoves = numMoves + 1;
16     }
17     this.pickFlasherIfPresent();
18   }
19
20   public void pickFlasherIfPresent()
21   { if (this.isBesideThing(Predicate.aFlasher))
22     { this.pickThing(Predicate.aFlasher);
23       this.flasherCount = this.flasherCount + 1;
24     }
25   }
26
27   public void turnAround()
28   { this.turnLeft();
29     this.turnLeft();
30   }
31
32   public int numFlashersCollected()
33   { return this.flasherCount;
34   }
35 }
```

The instance variable `flasherCount` is declared at line 4, inside the class, but outside of all the methods. It is declared just like a temporary variable except for the keyword `private` at the beginning of the line.

flasherCount starts out with a value of 0 when the robot object is created. However, just after a flasher is picked up at line 22 the instance variable is incremented. Incrementing flasherCount is very similar to incrementing the temporary variable at line 15 – except that we use the keyword this to emphasize that flasherCount is something that belongs to the object, much like move, turnLeft, and collectFlashers belong to the object.

The code at lines 32-34 provide a query answering the question of how many flashers the robot has collected so far. It is like other methods except that it says what kind of answer it returns – an integer, abbreviated int. It also includes an instruction to return the answer – the value contained in flasherCount – to the client that called the query.

There are many times when an object may want to remember information for a long time. A robot may want to remember how far it has traveled or the location of a Thing representing a pot of gold. An object representing a bank account will need to remember the balance for as long as it exists. An Employee object should remember the employee's starting date and annual salary, no matter which of many possible services is being executed – or even if no service is being executed at the moment.

On the other hand, if the information is only needed in a single method, an instance variable is more power than is needed. A temporary variable is likely a better choice.

F.8 The Same, But Different

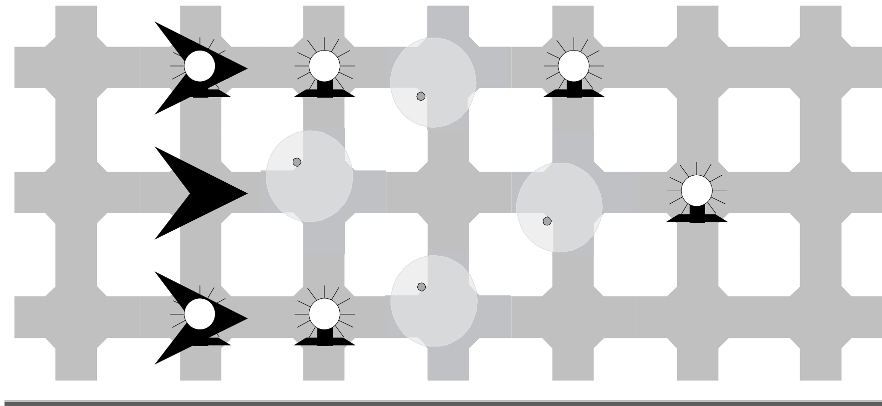
The paving project is nearly complete. Workers have put up streetlights on a number of the intersections; flashers remain on some of the rest (see Figure F-12). jasmine and pat have been moved to a different site and three new robots (sam, migel, and laura) are being used to turn the lights off each morning – both the flashers and the streetlights. sam, migel, and laura are instances of Extinguisher robots that include a service named turnLightsOff. Like collectFlashers, turnLightsOff has a parameter that says how many intersections the robot should visit.

To turn off a flasher, turnLightsOff could have code like this:

```
Flasher f = (Flasher)this.examineThing(
                                Predicate.aFlasher);
f.turnOff();
```

this.examineThing instructs this robot to examine the intersection it occupies for a Thing. In particular, the parameter, Predicate.aFlasher, tells it to look for a flasher. If the robot finds a flasher on the intersection, it assigns it to the temporary variable, f.

Figure F-12: Streets with two different kinds of lights, flashers and streetlights.



Similar code could be used to turn off a streetlight, except that we have to specify that we are interested in `Streetlight` objects.

```
Streetlight s =
    (Streetlight)this.examineThing(
                         Predicate.aStreetlight);
s.turnOff();
```

There is one restriction on this code. If the intersection does not actually have a streetlight, the program will produce a run-time error when it tries to turnoff the non-existent streetlight. One way to fix this problem is to use an `if` statement to only execute this code if the robot is beside a streetlight.

These ideas can be combined to create a `turnLightsOff` method. It, together with a helper method, are shown in Listing F-15.

The `turnLightsOff` method is very, very similar to the `collectFlashers` method. The difference is what happens on each intersection. Instead of calling `pickFlasherIfPresent`, `turnLightsOff` calls the method `turnLightsOffHere`.

When the `turnLightsOffHere` method executes, it first checks if the intersection has a flasher. If so, the robot gets the flasher and calls its `turnOff` method. Then the robot checks for a streetlight. If there is a streetlight, the robot gets it and calls its `turnOff` method.

Listing F-15: A first attempt at the `turnLightsOff` method.

```
 1 public void turnLightsOff(int howFar)
 2 { int numMoves = 0;
 3   while(numMoves < howFar - 1)
 4   { this.turnLightsOffHere();
 5     this.move();
 6     numMoves = numMoves + 1;
 7   }
 8   this.turnLightsOffHere();
 9 }
10
11 public void turnLightsOffHere()
12 { if (this.isBesideThing(Predicate.aFlasher))
13   { Flasher f = (Flasher)this.examineThing(
14                                  Predicate.aFlasher);
15     f.turnOff();
16   }
17   if (this.isBesideThing(Predicate.aStreetlight))
18   { Streetlight s = (Streetlight)this.examineThing(
19                                  Predicate.aStreetlight);
20     s.turnOff();
21   }
22 }
```

It is no coincidence that flashers and streetlights are both turned off with a method named `turnOff`. Recall from Section G.1 that we extended the `Robot` class to create a new kind of robot, a `Collector`. A `Collector` robot had all of the methods a regular `Robot` has: `move`, `pickThing`, `putThing`, and so on. It was also customized to include a new method, `collectFlashers`.

`Flasher` and `Streetlight` both extend the class `Light`. The `Light` class contains the methods `turnOn` and `turnOff`. The `Flasher` and `Streetlight` classes both inherit these methods. Just as a `Collector` robot can move, thanks to the `move` method inherited from `Robot`, a `Flasher` and a `Streetlight` can be turned on or off, thanks to the `turnOn` and `turnOff` methods inherited from `Light`.

Looking at these classes another way, `Flasher` and `Streetlight` are both a kind of `Light`. Therefore, they *must* be able to be turned on and off.

This yields an interesting idea. Perhaps we can instruct the robot to examine the intersection for a light. Not a flasher, in particular, nor a streetlight in particular, but just a light.

```
Light lite = (Light)this.examineThing(Predicate.aLight);
lite.turnOff();
```

This, in fact, *does* work. The eight lines of code in the `turnLightsOffHere` method, shown in Listing F-15, can be cut to just four lines, as shown in Listing F-16.

Listing F-16: Turning off lights instead of turning off flashers and streetlights.

```
1 public void turnLightsOffHere()
2 { if (this.isBesideThing(Predicate.aLight))
3   { Light lite = (Light)this.examineThing(Predicate.aLight);
4     lite.turnOff();
5   }
6 }
```

In some ways, it is surprising that this code works. After all, there are differences between streetlights and flashers. When a `Streetlight` is on, it just shines gently. When a flasher is on, it flashes insistently. It seems reasonable that these differences in behavior would result in differences in the `turnOff` methods – a streetlight would turn itself off differently than a flasher would turn itself off.

This is the case. The definition of `turnOff` for a `Streetlight` is

```
public void turnOff()
{ this.setIcon(this.offIcon);
}
```

while the definition of `turnOff` for a `Flasher` is

```
public void turnOff()
{ FlasherIcon fi = (FlasherIcon)this.getIcon();
  fi.stop();
  this.on = false;
}
```

Java allows a class to *override* methods in the class it extends. `Flasher` can replace the `turnOff` method inherited from `Light` with its own version, as can `Streetlight`. Then, when the statement

```
lite.turnOff();
```

is executed, `lite` might be a `Flasher` object or `lite` might be a `Streetlight` object. But it doesn't matter. Each object will use its own `turnOff` method. A `Flasher` will turn off one way; a `Streetlight` will turn off another way. They are the same – they both turn off – but they are

also different – they turn off in different ways. This concept of having the same service behave differently, depending on the class, is called *polymorphism*.

Polymorphism is useful when you have different kinds of objects that need to perform variations of the same basic action. For example, you might have two different kinds of dancing robots. A left-dancing robot moves to the left, forward, and then back to the right when sent the `move` message. A right-dancing robot moves to the right, forward, and then to the left when it moves. Both respond to the `move` message, but move differently.

As another example, consider an `Employee` class that is extended in two different ways: `HourlyEmployee` and `SalariedEmployee`. Every `Employee` should have a `calcWages` method, but `HourlyEmployee` and `SalariedEmployee` calculate the answer differently. Fortunately, the code

```
Employee e = (Employee)this.getNextEmployee();
e.calcWage();
```

will use the correct calculation no matter if `e` is an `HourlyEmployee` or a `SalariedEmployee`.

Or, consider programs such as *Windows Media Player* or *RealPlayer* that can play downloaded music or videos. If the user selects a music file, the `play` method in the `Music` class is executed. If the user selects a video file, the `play` method in the `Video` class is executed. This could be implemented by having `Music` and `Video` both extend a class named `Media`, which has a `play` method. Then, no matter what kind of media the user has selected, the lines

```
Media selected = (Media)this.getUsersSelection();
selected.play();
```

will cause the right `play` method to be executed. Thanks to polymorphism, each object can execute a method in an appropriate manner without the client even needing to know what kind of object it is. Polymorphism allows the code calling a method to treat objects in the same way, even if the objects belong to different classes.

F.9 Collections

The original paving job is finished and the construction company has landed another contract. This time the contract is much larger – a huge subdivision consisting of fifty streets. As before, robots are required to collect the flashers from the intersections each morning.

Working with fifty robots on fifty streets raises two issues. First is the tedium of coming up with the names for fifty robots. Second is the large amount of code that is exactly the same except for the name of the robot involved.

Using only the techniques we have learned so far, the `main` program would have to be written as shown in Listing F-17.

Listing F-17: A naïve program directing 50 robots to collect flashers on 50 streets.

```
 1 import becker.robots.*;
 2
 3 public class Main
 4 {
 5   public static void main(String[] args)
 6   { ConSite site = new ConSite();
 7
 8     Collector worker_0 =
 9              new Collector(site, 1, 1, Directions.EAST);
10     Collector worker_1 =
11              new Collector(site, 1, 2, Directions.EAST);
12     Collector worker_2 =
13              new Collector(site, 1, 3, Directions.EAST);
14     ...    // many workers omitted
15     Collector worker_48 =
16              new Collector(site, 1, 49, Directions.EAST);
17     Collector worker_49 =
18              new Collector(site, 1, 50, Directions.EAST);
19
20     CityFrame frame = new CityFrame(site);
21
22     worker_0.collectFlashers();
23     worker_0.turnAround();
24     worker_1.collectFlashers();
25     worker_1.turnAround();
26     worker_2.collectFlashers();
27     worker_2.turnAround();
28     ...    // many workers omitted
29     worker_48.collectFlashers();
30     worker_48.turnAround();
31     worker_49.collectFlashers();
32     worker_49.turnAround();
33   }
34 }
```

Using numbers in the names of the robots makes the similarity of many lines obvious. One might wonder if we can make use of all that similarity.

If fact, we can. There are various ways to collect many objects, such as robots, together. The result is called a *collection*. Collections are used by giving the name of the collection together with a number. For example, suppose the collection is named `workers` and already contains the fifty robots. Then the fifth robot could be told to collect flashers and turn around with the following code:

```
Collector worker = (Collector)workers.get(5);
worker.collectFlashers();
worker.turnAround();
```

This doesn't seem to be useful until we realize that the `5` can be replaced with a variable. If we put these three lines inside a loop that counts from 0 to 49, then 50 robots will collect flashers and then turnaround! All with just the following 7 lines of code (instead of the 100 required in Listing F-17).

```
int workerNum = 0;
while (workerNum < 50)
{ Collector worker = (Collector)workers.get(workerNum);
  worker.collectFlashers();
  worker.turnAround();
  workerNum = workerNum + 1;
}
```

The beauty of this approach is that 1,000 robots could be told to collect flashers with the same seven lines of code. Only the `50` in the second line would need to change.

Collections are used when lots of similar information must be managed.

Collections are often used in programs. Each robot uses a collection to implement its "backpack" and the `City` class uses a collection to manage all the things it contains – robots, walls, flashers, and so on. Programs used at a bank use collections to keep track of all the different `Account` objects and payroll programs use collections of `Employee` objects. Word processors use collections to store many `Paragraph` objects and all the words in the spelling checker's dictionary.

Any time a program must use many similar objects or similar pieces of information, a collection is probably being used.