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.

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.
import becker.robots.*;

public class Main extends Object {
    public static void main(String[] args) {
        City site = new City();

        // create the walls at the end of each street's construction zone
        Wall detour0 = new Wall(site, 0, 1, Directions.EAST);
        Wall detour1 = new Wall(site, 0, 2, Directions.EAST);
        Wall detour2 = new Wall(site, 0, 3, Directions.EAST);
        Wall detour3 = new Wall(site, 5, 1, Directions.WEST);
        Wall detour4 = new Wall(site, 5, 2, Directions.WEST);
        Wall detour5 = new Wall(site, 5, 3, Directions.WEST);

        // create 12 flashers, four on each street
        Flasher flash00 = new Flasher(site, 1, 1, true);
        Flasher flash01 = new Flasher(site, 2, 1, true);
        Flasher flash02 = new Flasher(site, 3, 1, true);
        Flasher flash03 = new Flasher(site, 4, 1, true);
        Flasher flash04 = new Flasher(site, 1, 2, true);
        // seven similar lines omitted

        // create the three robot workers
        Robot karel = new Robot(site, 1, 1, Directions.EAST);
        Robot jasmine = new Robot(site, 1, 2, Directions.EAST);
        Robot pat = new Robot(site, 1, 3, Directions.EAST);

        // display the construction site
        CityFrame frame = new CityFrame(site, 6, 5);

        // tell each robot what to do
        karel.pickThing();
        karel.move();
        karel.pickThing();
        karel.move();
        karel.pickThing();
        karel.move();
        karel.pickThing();
        karel.move();
        karel.turnLeft();
        karel.turnLeft();

        jasmine.pickThing();
        jasmine.move();
        // Seven instructions to jasmine, similar to lines 42-48, omitted.

        pat.pickThing();
        pat.move();
        // Nine instructions to pat omitted.
    }
}
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:

```java
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`.

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 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

```java
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.

```java
import becker.robots.*;

public class Collector extends Robot {

    public Collector(City city, int ave, int str, int dir) {
        super(city, ave, str, dir);
    }

    // Collect a row of four flashers
    public void collectFlashers() {
        this.pickThing();
        this.move();
        this.pickThing();
        this.move();
        this.pickThing();
        this.move();
    }

    // turn around and face the opposite direction
    public void turnAround() {
        this.turnLeft();
        this.turnLeft();
    }
}
```

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.

```java
import becker.robots.*;

public class <className> extends Robot
{
    public <className>(City city, int ave, int str, int dir)
    {
        super(city, ave, str, dir);
    }

    <newService>
}
```

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.

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

```java
import becker.robots.*;

public class NewMain extends Object
{
    public static void main(String[] args)
    {
        ConSite site = new ConSite();
        site.placeBarriers();
        site.placeFlashers();

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

        CityFrame frame = new CityFrame(site, 6, 5);
        continued...
```
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.
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.

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

```java
import becker.robots.*;

public class Collector extends Robot {
    public Collector(City city, int ave, int str, int dir) {
        super(city, ave, str, dir);
    }

    // Collect flashers as long as the front of the robot is not blocked.
    public void collectFlashers() {
        while (this.frontIsClear()) {
            this.pickThing();
            this.move();
        }
        this.pickThing();
    }

    // turn around and face the opposite direction
    public void turnAround() {
        this.turnLeft();
        this.turnLeft();
    }
}
```
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.

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.

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.](image)

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.

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:
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

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

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.

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     {
7         super(city, ave, str, dir);
8     }
9
10     // Collect flashers as long as the front of the robot is not blocked.
11     public void collectFlashers()
12     {
13         while (this.frontIsClear())
14             {
15                 this.pickFlasherIfPresent();
16                 this.move();
17             }
18
19     // pick up a flasher, if one is present on the current intersection
20     public void pickFlasherIfPresent()
21     {
22         if (this.isBesideThing(Predicate.aFlasher))
23             {
24                 this.pickThing(Predicate.aFlasher);
25             }
26         }
27
28     // 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.

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.

To make this plan work, each robot will need to remember how many moves it has made while it is collecting 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.

\[ \text{int numMoves} = 0; \]

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

\[ \text{numMoves} = 5; \]

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,

```java
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,

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

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.

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

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

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.

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.

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:

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

Use a counted loop when the number of repetitions is known in advance.
Listing F-8: Picking up half of the things on an intersection.

```java
public void pickHalf()
{
    int numThingsFound = 0;
    while (this.isBesideThing())
    {
        this.pickThing();
        numThingsFound = numThingsFound + 1;
    }

    int numPutBack = 0;
    while (numPutBack < numThingsFound/2)
    {
        this.putThing();
        numPutBack = numPutBack + 1;
    }
}
```

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.

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