There are several advantages to using pseudocode:

- Pseudocode helps us think more abstractly. As we discussed briefly in Section 1.1.1, abstractions allow us to "chunk" information together into higher level pieces so that we don't need to remember as much. In this case, pseudocode enables us to chunk together many lower-level steps into a single higher-level step, such as `pick all the things in one row`. Such higher-level thinking, however, comes at a cost: less precision. This lack of precision may allow us to accidentally slip in a "miracle" (see the cartoon in Figure 3-3), but overall, the benefits of using pseudocode outweigh the costs.
- Pseudocode allows us to simulate, or trace, our program very early in its development. We can trace the program after only scratching out a few lines on paper. If we find a bug, it is much easier to change and fix it than if we had invested all the time and energy into obeying the many details of the Java language.
- If we are working with other people, even nontechnical users, pseudocode can provide a common language. With it, we can describe the algorithm to others. They might see a special case we missed or a more efficient approach, or even help implement it in a programming language.
- Algorithms expressed with pseudocode can be converted into any computer programming language, not just Java.

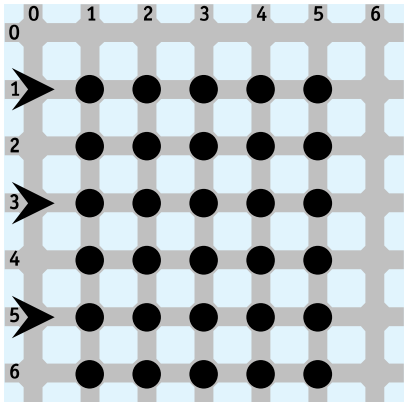
Pseudocode's usefulness increases as the complexity of the algorithm you are designing increases. In the next chapter, we will introduce Java constructs that allow us to choose whether to execute some statements. Other constructs allow us to repeat statements. These constructs are very powerful and vital to writing interesting programs—but they also add complexity, a complexity that pseudocode can help manage in the early stages of programming.

3.5 Variations on the Theme

Consider again the field harvesting task discussed in Section 3.2. There are many variations. Perhaps several robots are available to perform the task, or instead of harvesting things, the robot needs to plant things. This section explores these variations. In the process, we will see that early in the development of the `Harvester` class we made a key assumption that we should use a single robot. This assumption needlessly complicated the program; we should have explored more alternatives. We will also see how to make the computer appear to do several things at once, such as six robots all harvesting a row of things simultaneously.

3.5.1 Using Multiple Robots

One approach to solving the harvesting problem is to use several robots, which we briefly considered early in the process. In this approach, each robot harvests only a part of the field. For example, our `main` method could be modified to instantiate three robots, each of which harvests two rows. The initial situation is shown in Figure 3-8 and in the program in Listing 3-5. `mark` will harvest the first two rows; `lucy` the middle two rows; and `greg` the last two rows. Of course, the work does not need to be divided evenly. If there were only two robots, one could harvest two rows, and the other could harvest four rows.



(figure 3-8)

Harvesting a field with three robots each harvesting two rows

Listing 3-5: *The* `main` *method for harvesting a field with three robots*

```
1  import becker.robots.*;
2
3  /** Harvest a field of things using three robots.
4   *
5   * @author Byron Weber Becker */
6  public class HarvestTask extends Object
7  {
8    public static void main(String[] args)
9    {
10     City stLouis = new City("Field.txt");
11     Harvester mark = new Harvester(
12                              stLouis, 1, 0, Direction.EAST);
13     Harvester lucy = new Harvester(
14                              stLouis, 3, 0, Direction.EAST);
15     Harvester greg = new Harvester(
16                              stLouis, 5, 0, Direction.EAST);
```

FIND THE CODE

ch03/harvestWithThree/

Listing 3-5: *The* `main` *method for harvesting a field with three robots* (continued)

```
17
18      mark.move();
19      mark.harvestTwoRows();
20      mark.move();
21
22      lucy.move();
23      lucy.harvestTwoRows();
24      lucy.move();
25
26      greg.move();
27      greg.harvestTwoRows();
28      greg.move();
29    }
30  }
```

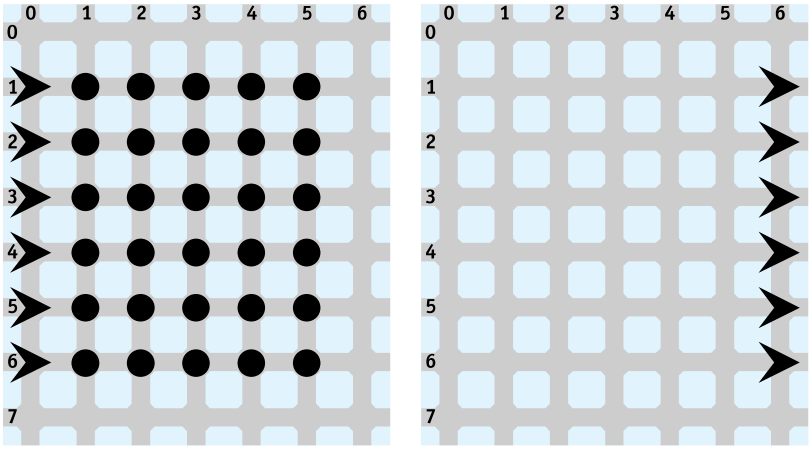
FIND THE CODE

ch03/harvestWithSix/

In fact, the original problem does not specify the number of robots to use, where they start, or where they finish. Perhaps the simplest solution is to have six robots each harvesting one row, and ending on the opposite side of the field. The initial and final situations are shown in Figure 3-9. If we had chosen this solution, the `Harvester` class would have consisted of only `harvestOneRow` and `harvestIntersection`—much simpler than what we actually implemented.

(figure 3-9)

Harvesting with six robots



3.5.2 Multiple Robots with Threads (advanced)

In the previous example, which uses six robots, one robot finishes its entire row before the next one begins to harvest its row. The entire task takes about six times as long as harvesting a single row, even though we have six robots.

If we were paying a group of people an hourly wage to perform this task, we would be pretty upset with this strategy. We would want them working simultaneously so that the entire job is done in about the same amount of time it takes one person to harvest one row.

In this section, we'll explore how to make the robots (appear to) do their work simultaneously. This material is normally considered advanced, but robots provide a clear introduction to these ideas, and it's a fun way to stimulate your thinking about other ways to do things. Check with your instructor to find out if he or she expects you to know this material.

Example: ThreadedRowHarvester

When you have several robots working simultaneously, each robot must be self-contained. The `main` method will start each robot, after which your robots will perform their tasks independently. This approach implies that each robot must be instantiated from a subclass of `Robot`, which "knows" what to do without further input from the program. We'll call this subclass `ThreadedRowHarvester`.

KEY IDEA

The `run` method contains the instructions the thread will execute.

The instructions each robot should execute after it's started are placed in a specially designated method named `run` in the `ThreadedRowHarvester`. The `run` method is free to call other methods to get the job done. In our case, we call the `HarvestOneRow` and `move` methods, as shown in the following method. The `run` method should be inserted in the `ThreadedRowHarvester` class. `harvestOneRow` is defined as in the `Harvester` class.

```
/** What the robot does after its thread is started. */
public void run()
{ this.move();
  this.harvestOneRow();
  this.move();
}
```

In the `main` method, we need to construct six `ThreadedRowHarvester` robots, one for each row. However, instead of instructing each robot to harvest a row, we start each robot's thread. The `run` method defined earlier then instructs the robot what to

Multiple Threads

do. A thread is started with two statements, one to create a `Thread` object and one to call its `start` method. For a robot named `karel`, use the following statements:

FIND THE CODE

ch03/harvestWithSixThreads/

```
ThreadedRowHarvester karel = new ThreadedRowHarvester(...);
...
Thread karelThread = new Thread(karel);
karelThread.start();
```

The `start` method in the last statement invokes the `run` method, which contains the instructions for the robot. For this strategy to work, the `Thread` class must be assured that the `ThreadedRowHarvester` class actually has a `run` method. You do so by adding `implements Runnable` to the line defining the class:

```
public class ThreadedRowHarvester extends Robot
                                  implements Runnable
```

This statement is your promise to the compiler that `ThreadedRowHarvester` will include all of the methods listed in the `Runnable` interface. The `run` method is the only method listed in the documentation for `Runnable`.

In summary, three things need to be completed to start a thread:

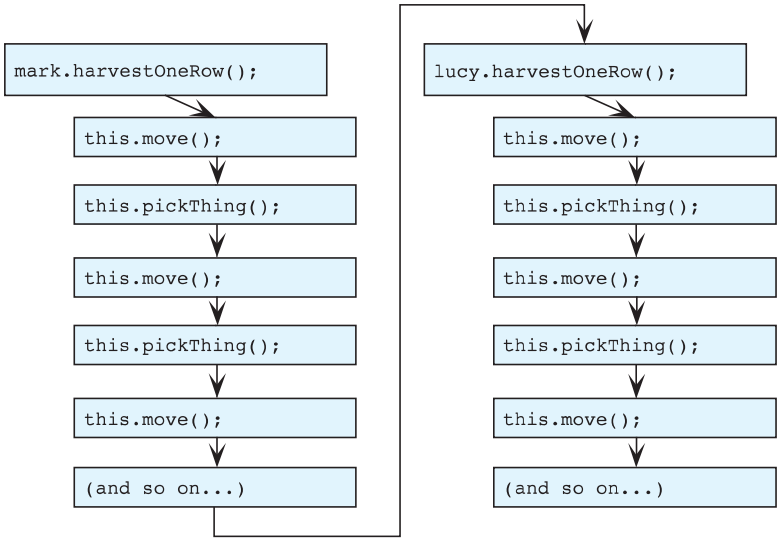
- Include the instructions for the robot in a specially designated method called `run`.
- Implement the interface `Runnable` to tell Java that your class is set up to run in a thread.
- Start the thread.

In this example, each thread performs identical tasks, which need not be the case. We could, for instance, set up two threads with robots harvesting two rows each, and two more threads with robots harvesting one row each.

About Threads

A thread starts a new flow of control. We learned in the Sequential Execution pattern that each flow of control is a sequence of statements, one after the other, where each statement finishes before the next one begins.

The `main` method begins execution in its own thread. As long as we don't start any new threads, execution proceeds one statement after another, as shown in Figure 3-10. This figure supposes that we have two robots named `mark` and `lucy`. The `main` method first calls `mark.harvestOneRow();` and then `lucy.harvestOneRow();`. Between these calls, many other statements are executed, one after the other.



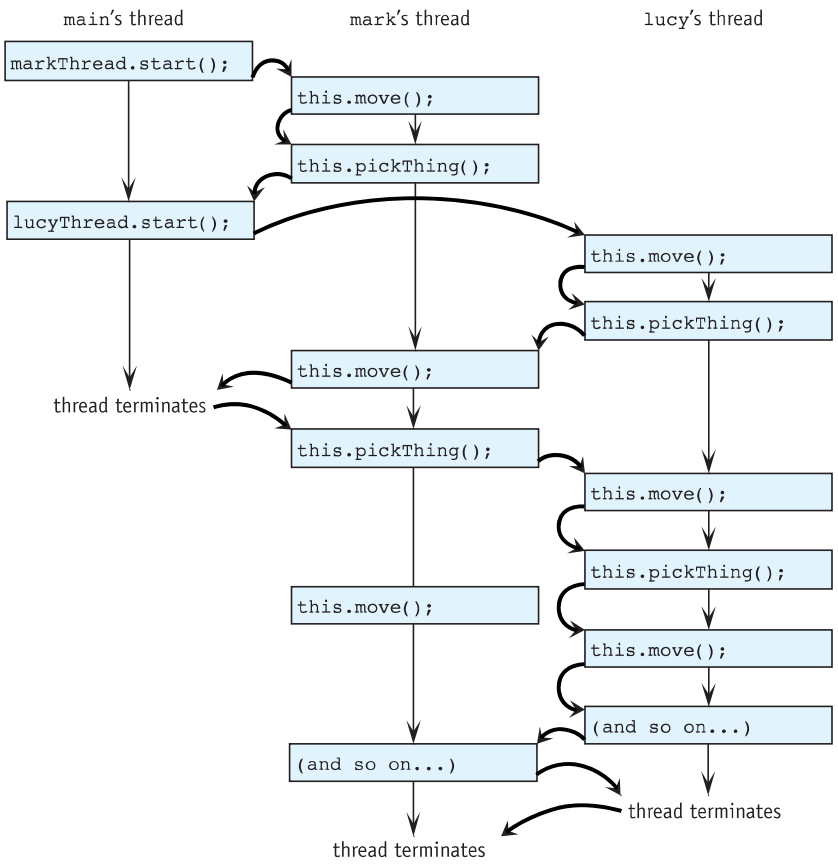
(figure 3-10)

Flow of control with only one thread

When we have two or more flows of control, execution switches among them. The statements *within* each flow of control still execute in order with respect to each other, but statements from a different thread might execute between them. This concept is illustrated in Figure 3-11.

The `main` method's flow of control starts a thread for `mark` and then for `lucy`, represented by the light arrow between the two left-most boxes.

But now that we have three threads of control (one for `main`, one for `mark`, and one for `lucy`), the execution switches between all three threads, as represented by the heavier arrows. Execution switches among the threads so quickly that it *appears* that all the robots are moving simultaneously, though they are not (unless you are fortunate enough to have a computer with at least as many processors as the program has threads). The computer's operating system ensures that each thread runs at least a little bit before stopping it and starting another thread. It also ensures that every thread is eventually run.



(figure 3-11)

One possible flow of control with three threads

Notice that although execution switches among the threads, the statements within each thread are still executed in the same order as before. The only difference is that statements from another thread might be executed between the statements.

Complexities

This simple example glosses over some complexities. For instance, each robot's task in these examples is independent of the tasks performed by the other robots. If a seventh robot collected all the things harvested by the first six robots, it would need a way to wait for those robots to finish their task before starting.

In the next chapter, we will explore ways that programs can make decisions. For example, a robot can check if a `Thing` is present on the intersection. Suppose `mark` is programmed to check for a `Thing` on the current intersection. If there is one, `mark` picks it up; otherwise, `mark` goes on to the next intersection. But the check is in one program statement and the call to `pickThing` is in another. `lucy`, running in another thread,

might come along and snatch the thing between those two statements. So the thing `mark` thought was there disappears, and `mark` breaks when it executes `pickThing`.

In spite of these and other complexities, threads are a useful tool in many applications. For example, animations run in their own threads. Many word processors figure out page breaks in a separate thread so that the user can continue typing at the same time. Printing usually has a separate thread so that the user can do other work instead of waiting for a slow printer. Graphical user interfaces usually run in one or more threads so that they can continue to respond to the user even while the program is carrying out a time-consuming command.

LOOKING AHEAD

In Section 10.7, we'll learn how to use a thread to perform animation in a user interface.

3.5.3 Factoring Out Differences

Suppose that instead of picking one thing from each intersection in the field, we want to plant a thing at each intersection. Other alternatives include picking two things or counting the total number of things in the field.

Each of these programs is similar to the harvesting task. In particular, the part that controls the movement of the robot over the field is the same for all of these problems; it is only the task at each intersection that differs. The original task of harvesting things is only one example of a much more general problem: traversing a rectangular area and performing a task at each intersection.

KEY IDEA

Think about variations of the problem early in the design.

If we started with this view of the problem, we might design the program differently. Instead of solving the harvesting problem directly, we could design a `TraverseAreaRobot` that traverses a rectangular area. At each intersection, it calls a method named `visitIntersection` that is defined to do nothing, as follows:

```
public void visitIntersection()
{
}
```

By overriding this method in different subclasses, we can create robots that harvest each intersection or plant each intersection, and so on. A class diagram illustrating this approach is shown in Figure 3-12.

It may seem strange to include a method like `visitIntersection` that does nothing. However, this method must be present in `TraverseAreaRobot` because other methods in that class call it. On the other hand, we don't know what to put in the method because we don't know if the task is harvesting or planting the field, and so we simply leave it empty, ready to be overridden to perform the appropriate action.