## 5.4 Case Study: Solving a Maze

Here is another application that illustrates how stacks are used. We are given a maze, like this:

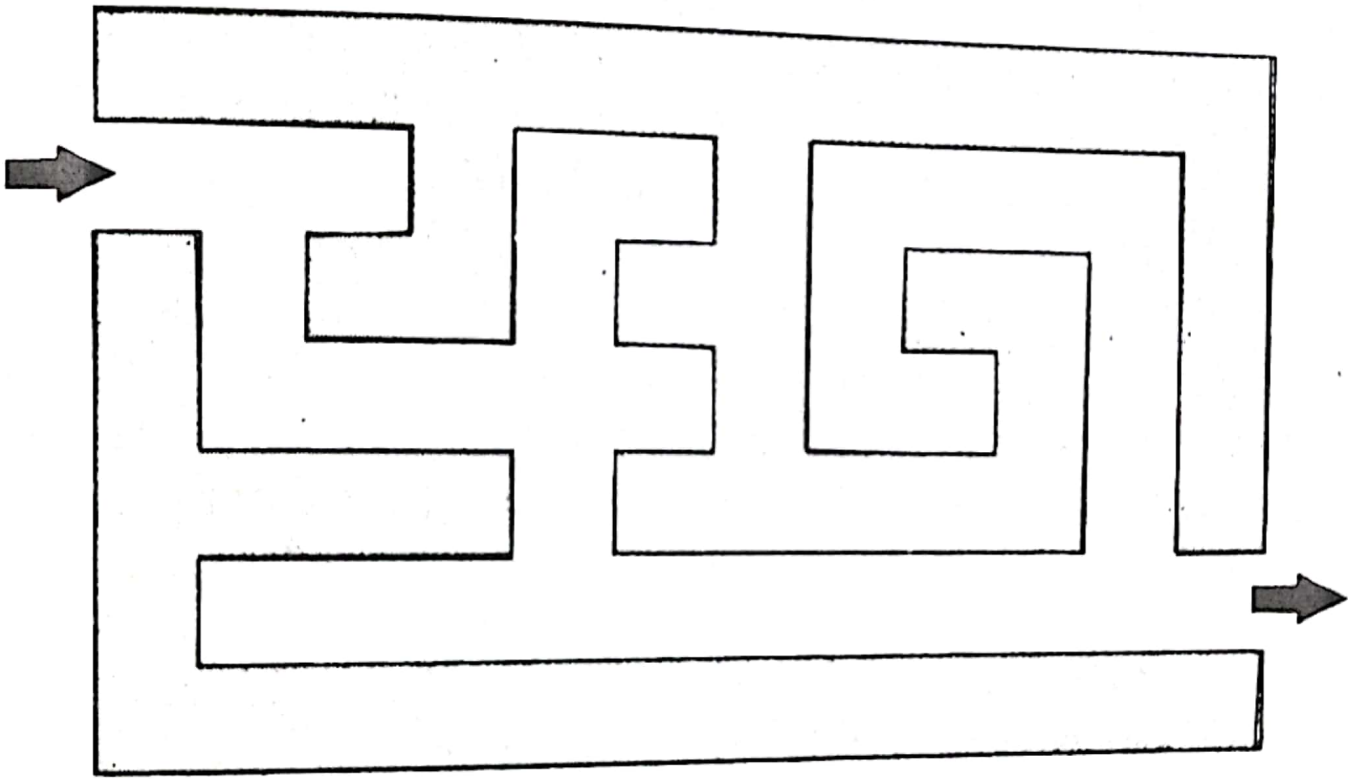

**FIGURE 5.6** A maze.

The problem is to find a path through the maze, from the entrance point at the upper left to the exit point on the lower right, as shown in Figure 5.6. Imagine a rat trying to find its way through this maze.

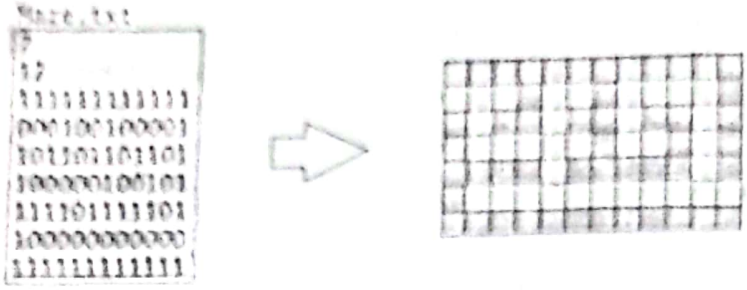

FIGURE 5.7 Loading the maze from a text file.

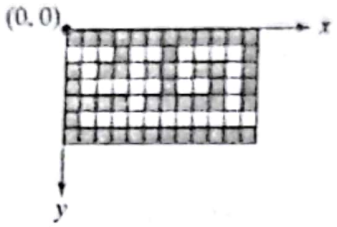


FIGURE 5.8 Coordinates.

We will store the maze in a two-dimensional array of `ints`, the value of each element being either 0 or 1. The value 0 represents part of a corridor, and the value 1 represents part of a wall. We will load the array from an ordinary text file like the one shown Figure 5.7. This `Maze.txt` file would load an array that would represent the maze.

The file first lists the number of rows ($m = 7$) and the number of columns ($n = 12$). Then, starting on the third line, it gives the complete $m$-by-$n$ array of 0s and 1s that define the maze.

To be consistent with the graphics context of the Java Abstract Window Toolkit (AWT), we use $(x, y)$ coordinates with the origin (0, 0) at the upper left corner, the $x$-axis running horizontally across the top to the right, and the $y$-axis running vertically along the left edge downward, as shown in Figure 5.8. So the entrance cell will always be at coordinates (0,1) and the exit cell will always be at coordinates $(n-1, m-2)$. In this example, the exit cell is (11, 5).

For an object-oriented solution to the problem, we first identify the primary objects: the maze itself, and a rat to move about in the maze. These require a `Maze` class and a `Rat` class. We'll use a separate `main()` class named `SolveMaze` to generate the solution, a path through the maze.

To generate the path, we have to move about, from one location to another in the maze. So we'll also define a `Location` class and a `Direction` class. A location is really just a pair of integer coordinates $(x, y)$, and a direction is really just one of four compass directions: north, south, east, or west. Creating separate classes for these entities is consistent with the spirit of object orientation, and it renders the result easier to understand and thus easier to generalize (*e.g.*, to three dimensions). The five classes, each in its own file, are depicted in Figure 5.10,

Listing 5.5 shows the `main()` class. It declares three objects: a `maze`, a `rat`, and a `stack`. The `main()` method simply instantiates the `main()` class, passing to the constructor the command

line argument `args[0]` which names the input file that contains the definition of the maze to be solved. We run the program with this command:

```
java SolveMaze Maze.txt
```

**LISTING 5.5: Solving a Maze**

```
1  public class SolveMaze {
2    Maze maze;
3    Rat rat;
4    Stack stack;
5
6    public static void main(String[] args) {
7      new SolveMaze(args[0]);
8    }
9
10   public SolveMaze(String file) {
11     maze = new Maze(file);
12     rat = new Rat(maze);
13     stack = new ArrayStack();
14     maze.print();
15     while (!rat.isOut()) {
16       Location currentLocation = rat.getLocation();
17       // see Programming Problem 5.23 on page 173
18     }
19   }
20 }
```

The output is

```
############
oo #  #    #
# ## ## ## #
#     #  # #
#### ##### #
#
############

############
oo?#??#????#
#o##?##?##?#
#oooo?#??#?#
####o#####?#
#   oooooooo
############
```

The output consists of the results from two calls to the `Maze.print()` method: one before, and one after the solution has been found. For this maze, the first seven lines of the output show the maze rendered with `'#'` and `' '` characters for the walls and corridors, respectively. The path of the rat is shown with `'o'` characters. So the "before" picture shows the rat just getting started in the upper left corner at the entrance to the maze. The "after" picture (the last seven lines here) shows the rat's path from entrance to exit. It also shows attempted dead ends, rendered with the `'?'` character.

The `main()` class constructor invokes the `Maze`, `Rat`, and `ArrayStack` constructors at lines 11–13 to initialize the `maze`, `rat`, and `stack` objects. These are illustrated in Figure 5.9. A `Location` object is a pair of integers, such as (10, 2). A `Maze` object is a two-dimensional array of integers, along with its dimensions `m` and `n`. The `Rat` object stores its current `location` and a reference to the `Maze`. Besides the `Rat` object and the `Maze` object, the `main()` class also has an `ArrayStack` object.

After these objects have been instantiated, the program prints the maze and starts the main `while` loop at line 15. On each iteration of the `while` loop, the rat makes one move on the grid, from one cell to a neighboring cell. The loop is controlled by an `isOut()` method in the `Rat` class that can determine when the rat gets out of the maze. We also assume the `Rat` class has a `getLocation()` method that returns the current `location` of the rat in the maze.

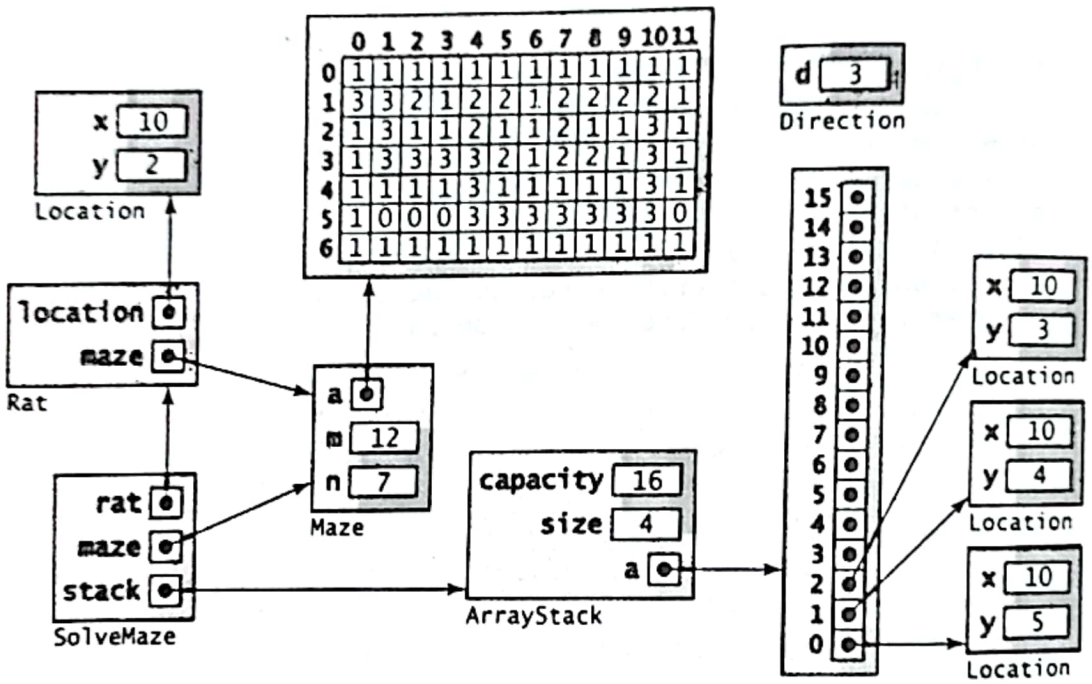


**FIGURE 5.9** The data structures for the Maze solution.

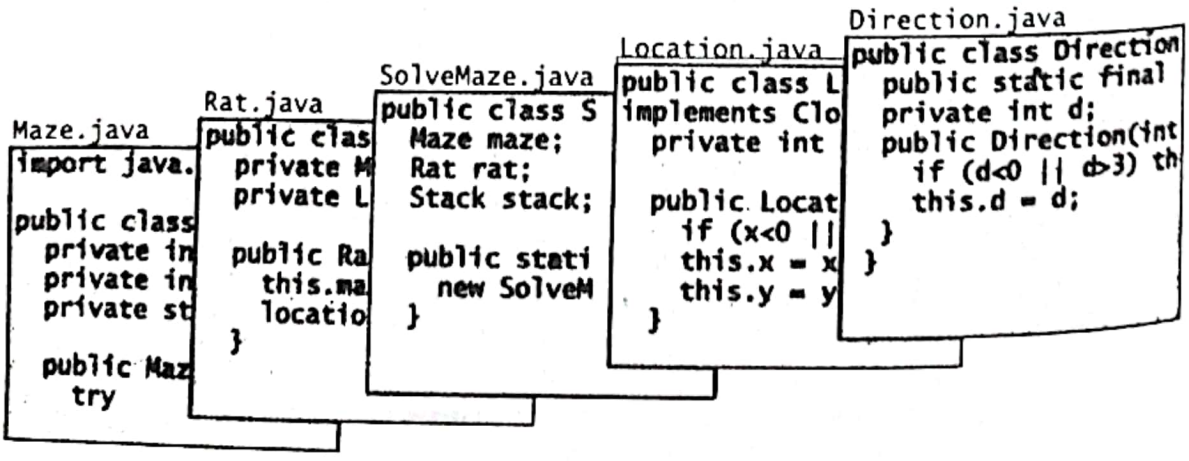


**FIGURE 5.10** The classes for the Maze solution.

The states of the data structures shown in Figure 5.9 represent the maze on the iteration where the rat has moved all the way into the spiral and then back-tracked to location (10,2). The back-tracking from the center of that spiral that the rat has already done is marked by the "2" digits at positions the (8,3), (7,3), (7,2), (7,1), (8,1), (9,1), and (10,1). The back-tracking that the rat still has to do is marked by the "3" digits at positions the (10,2), (10,3), (10,4), and (10,5). After that, it will find the exit at (11,5).

**LISTING 5.6: The Maze Class**

```
1  import java.io.*;
2
3  public class Maze {
4    private int m, n;
5    private int[][] a;
6    private static final int OPEN = 0, WALL = 1,  TRIED = 2, PATH = 3;
7
8    public Maze(String file) {
9      // see Programming Problem 5.26 on page 173
10   }
11
12   public boolean isOpen(Location location) {
13     return (a[location.getY()][location.getX()] == OPEN);
14   }
15
16   public void markMoved(Location location) {
17     a[location.getY()][location.getX()] = PATH;
18   }
19
20   public void markTried(Location location) {
21     a[location.getY()][location.getX()] = TRIED;
22   }
23
24   public int getWidth() {
25     return n;
26   }
27
28   public int getHeight() {
29     return m;
30   }
31
32   public void print() {
33     char[] chars = {' ', '+', '?', 'o'};
34     for (int i = 0; i < m; i++) {
35       for (int j = 0; j < n; j++)
36         System.out.print( chars[ a[i][j] ] );
37       System.out.println();
38     }
39   }
40 }
```

We implement the following backtracking algorithm to solve the maze problem:

1. If the rat can move in any one of the four directions (north, east south, west), then push its current location on the stack and move it to the neighboring location in that direction;
2. Otherwise, if the stack is empty, report that there is no solution, and exit;
3. Otherwise, mark the current location in the maze as "tried," pop the last location from the stack, and move the rat back to that previous location.

The `Maze` class (see Listing 5.6) uses four `static` named constants: `OPEN`, `WALL`, `TRIED`, and `PATH`. Each element of the array `a[][]` stores one of these four values. Initially, they all are either `OPEN` or `WALL`, as read from the file. Each iteration of the main loop changes one cell to either `TRIED` or `PATH`. When the loop is terminated, the `PATH` cells show the solution to the maze. For example, the array `a[][]` for the maze shown in Figure 5.6 will progress as shown Figure 5.11 on page 165. The `WALL` cells (value 1) are slightly shaded. The cell that changed at each iteration is shown with a darker square boundary. The three dots stand for 35 missing snapshots of the grid before it reaches the final two states shown. The `TRIED` cells (value 2) indicate *backtracking*.

**LISTING 5.7: The `Rat` Class**

```
1  public class Rat {
2    private Maze maze;
3    private Location location;
4
5    public Rat(Maze maze) {
6      this.maze = maze;
7      location = new Location(1,1);
8    }
9
10   public Location getLocation() {
11     return (Location)location.clone();
12   }
13
14   public void setLocation(Location location) {
15     this.location = location;
16   }
17
18   public boolean canMove(int direction) {
19     Location neighbor = location.adjacent(direction);
20     return maze.isOpen(neighbor);
21   }
22
23   public void move(int direction) {
24     location.move(direction);
25     maze.markMoved(location);
26   }
27
28   public boolean isOut() {
29     // See Programming Problem 5.27 on page 174
30   }
31 }
```

The Rat class is shown in Listing 5.7. Note how messages are sent from one objec another. The main() class instance "asks" the rat if it can move in the direction north statement:

```
if (rat.canMove(Direction.NORTH)) ...
```

To respond, the rat "asks" its location object to report who its northern neighbor is:

```
Location neighbor = location.adjacent(direction);
```

Then it "asks" the maze object whether that neighbor is open:

```
return maze.isOpen(neighbor);
```

The rat then sends that message back to the main() class object. This kind of *message passi* central to object-oriented programming.

**LISTING 5.8: The Location Class**

```
1  public class Location implements Cloneable {
2    private int x, y;
3
4    public Location(int x, int y)  // See Programming Problem 5.28
5
6    public Object clone()  // See Programming Problem 5.28 on page 17
7
8    public int getX()  // See Programming Problem 5.28
9
10   public int getY()  // See Programming Problem 5.28
11
12   public void move(int direction) {
13     switch (direction) {
14       case Direction.NORTH: --y;  break;
15       case Direction.EAST : ++x;  break;
16       case Direction.SOUTH: ++y;  break;
17       case Direction.WEST : --x;  break;
18     }
19   }
20
21   public Location adjacent(int direction) {
22     switch (direction) {
23       case Direction.NORTH: return new Location(x, y-1);
24       case Direction.EAST : return new Location(x+1, y);
25       case Direction.SOUTH: return new Location(x, y+1);
26       case Direction.WEST : return new Location(x-1, y);
27     }
28     return null;
29   }
30
31   public String toString()  // See Programming Problem 5.28 on pag
32 }
```

The `Location` class is shown Listing 5.8. Note the use of the `break` statement in the `switch` statement at lines 14–17 to avoid a "fall through."

Finally, the `Direction` class is shown in Listing 5.9. It is really little more than an enumeration type.

**LISTING 5.9: The `Direction` Class**

```
1  public class Direction {
2    public static final int NORTH = 0, EAST = 1, SOUTH = 2, WEST = 3;
3    private int direction;
4
5    public Direction(int d) {
6      if (d < 0 || d > 3) throw new IllegalArgumentException();
7      direction = d;
8    }
9
10   public int getDirection() {
11     return direction;
12   }
13 }
```

The use of a stack to manage backtracking is a common strategy for game playing and for more general modeling of rational thought. No one knows yet quite how the brain works, but its use of general trial-and-error strategies is clear. Simulating that with a computer usually calls for the stack data structure.