

Maze Escaping Robots!!!

1 Objectives

1. Use inheritance
2. Use exceptions
3. Design and execute a testplan
4. Use version control
5. Use Javadoc
6. Write documentation
7. Revise your work
8. Comment your code

2 Maze Escaping Robots!!!

This is an elaboration of project 10.1 on page 495 of Big Java. Read that problem before you go on.

It is the distant future, the year 2000, we are maze escaping robots!!! Robots have taken over the world and put criminal robots into maze-like prisons. Your task is to program wrongfully accused robots with varying behaviors that will enable them to escape their maze-y jails, such as the following:

```
*X*****
*      * *
* ***** *
* * * * *
* * *** *
* ^* *
*** * * *
*E      * *
*****X*
```

The ^ indicates the current location of the robot and the direction that it is facing, E indicates the maze entrance, and the Xs are exit locations. Provide a common superclass `Robot` that captures qualities that are common to all robots. Provide subclasses `RandomRobot`, `RightHandRuleRobot`, and `MemoryRobot`. Each of these robots has a different strategy for escaping. The `RandomRobot` simply makes random moves. The `RightHandRuleRobot` moves around the maze so that its right hand always touches a wall. The `MemoryRobot` remembers all positions that it has previously occupied and never goes back to a position that it knows to be a dead end.

The `MemoryRobot` is Extra Credit and should be saved for last.

3 Details

In this project you will practice using inheritance by implementing several different kinds of virtual robots that navigate around a maze according to different sets of rules. We will use the ideas of inheritance to minimize the amount of code duplication — the enemy of any good implementation. This project also requires some design on your part, so start thinking about the design early and don't start coding right away.

You will need at least 5 classes: `Maze`, `Robot`, `RandomRobot`, `RightHandRuleRobot`, `MemoryRobot`, and `RobotEscapeException`. As the book indicates, `Robot` is the superclass of the robots. **You must name your classes as seen here.**

3.1 Maze Details

Your book doesn't say much about the functionality of the `Maze` class. However, they will need to keep track of an entrance location and exit locations. An entrance location cannot also be an exit location. Here are some other implementation suggestions. Feel free to use other ideas if you do not like these suggestions.

1. You will need an instance variable to store the maze. What will you use to represent it?
2. You will need one or more `Maze` constructors. For simplicity you only have to construct regular, rectangular mazes where every row has the same number of columns. The interior of the maze may have any configuration.
3. Override the `toString` method so that it returns a string representation of the maze. You can use the format from the book, asterisks (*) and blanks, or come up with your own. `toString` *does not* print anything! It just returns a string. This should be one of the first methods that you write.

Explain and justify your choices in your technical specification before you write any code.

3.2 Robot Superclass

Each robot should keep track of its location in the maze. Since all robots need this, it should be part of the superclass. Before a robot can move in a maze, it must enter the maze. As a result, unlike in the book, you should not need to pass the maze as a parameter to `move`. Instead, all robots will need to implement an `enter` method that takes the maze as its parameter and sets the robot's location to the entrance point. When a robot enters a maze, it should never be pointing towards a wall. As a result, make sure it has the right orientation (when there are more than one possible starting orientations, use the following order of orientation precedence: up, right, down, left).

The `Robot` class should have a `toString` method that overrides the `Object` version of that method. `toString` takes no parameters and returns a concise description of the robot as a string. Specifically, it should indicate the robot's location.

To make good usage of inheritance, ask yourself what actions could all robots take? Make these methods part of the `Robot` class. You cannot have an instance of a `Robot` since the "brains" portion that knows how to escape will be in the subclasses. This means you can take advantage of abstract methods. Polymorphism may also come in handy here.

One of the common robot actions will be movement in the maze. These actions should be as simple as possible and do only one thing. For example, to take a step forward is simple. To turn left (in place) is simple. To step forward and then turn left is not simple.

3.3 Robot Subclasses

Don't forget to create constructors for each subclass. If a subclass has additional instance variables, initialize them in the constructor for that class *after* calling the superclass constructor. You may also need to override the `toString` method.

The `RandomRobot` will use random numbers to choose each step. It may take quite a while to escape using this strategy (or the escape may never happen at all!). If the number of moves required to exit the maze reaches ten thousand, you should assume that the robot will never escape and have it quit trying.

The `RightHandRuleRobot` will try to escape by walking with its right hand on the wall at all times. For this class, begin by assuming that the robot is always placed in the maze with a wall on its right. After you get that working relax that restriction. An easy way to do this is to have the robot move forward until it encounters a wall. After that, it should keep a wall on its right.

The `MemoryRobot` must remember where it has been and not go back to any dead ends it encounters. You can use anything you want to simulate its memory (but justify your decision). I might suggest a 2D array.

To keep the problem manageable, limit the moves that a robot can make to turn (rotate) left, turn (rotate) right, and go forward. For `RightHandRuleRobot` robot, you may need to add a fourth move: go around right corner. Without this, the

robot can find itself without a wall on it's right. Normally, this would be three separate moves, go forward, turn right, go forward. For the `RightHandRuleRobot`, this is an atomic (all done in one step) move.

Your technical specification should include the algorithm for each robot's movement. These algorithms should be detailed and are often best explained in pseudo code rather than wordy English.

3.4 Strategy and Hints

I recommend that you implement your robot classes in the following order.

1. `RobotEscapeException`
2. `Robot`
3. `Maze`
4. `RandomRobot`
5. `RightHandRuleRobot`
6. `MemoryRobot` (Extra Credit)

`RightHandRuleRobot` is a little harder than `RandomRobot`. Before moving on to a new segment of the project, make sure you have a well tested, well commented and well designed program.

The `toString` method will be your friend. Implement it for each class in such a way that it provides useful information.

How will you convey the progress of the robot? Should you provide different options on how to display the robot? Remember that you will need to convince me that it is working. When the robot reaches a maze exit (or quits trying), it should throw a `RobotEscapeException` that contains the number of moves required to reach its current location. Catch the exception in your main method, print a victory (or failure) message, and quit the program.

3.5 Testing

How do you know your robots are obeying their rules and not just hopping over the wall to freedom? Develop a testplan to verify that each robot behaves correctly. Use the testplan handout for help. You will need to develop the testplan **in the design phase**. You will need to execute your testplan for each type of robot **after** you are done.

3.6 Sample Mazes

I have provided some sample mazes which are linked from the website.

Provide a mechanism for trying your code with different mazes. If you provide a main method for each robot, it will be easy to test the various robots.

3.7 Read Me

Include a `readme.txt` file that explains where you had difficulty with this project. Does each robot work for all 6 mazes? The last 3 mazes are large and it takes a long time for a random robot to escape. Have your robots report how many moves it took to escape. Prepare a table showing how many moves each robot takes for each of the mazes.

3.8 Important Deadlines

You may use any existing classes and objects that you worked with in the labs or in the previous projects. You will need to produce Javadoc for this assignment. As always you need appropriate comments. You must use Subversion for this project.

3.9 Phase 0 - The day it is assigned

You must work in a teams of two or three. You may not work with someone who has already been your partner for an assignment. Make sure your name and your partners name(s) are included (typed) with all submitted documents.

Give me one clearly written piece of paper (or email) with the usernames of all people in your team and your team name. Don't make me choose a team name for you or you will get something ridiculous like Boss Atlantic.

3.10 Phase 1

Handin via Subversion

- Problem description as a Javadoc style comment at the top of the Robot class.
- CRC cards as html/ascii drawings embedded in Javadoc.
- UML in Violet.
- Algorithms as a Javadoc style comment at the top of the methods they pertain to.
- Testplans in English or ascii drawings in a text file.
- An overall technical specification in Javadoc using the private flag so your problem description, member data, methods, and algorithms can all be seen in a Javadoc page like the API. (Note: the API only shows public things and is a User Manual, you are showing private things to make a Technical Specification).

I expect that your Javadoc will show all member data and methods for Maze, Robot, Random Robot, and Right Hand Robot. These methods do not have to have code in them yet but if you instead write internal comments, you will spend less time coding later.

Do not wait for my input before beginning the next phase.

3.11 Phase 2

You must get the Random and Right Hand robots working. The Memory robot is EC and due by the last Monday of classes in the semester.

Your code should follow the Style Guide posted on the CSCI 204 web site.

Handin via Subversion

- User manual (text file is fine)
- CRC, UML, Testplans, Technical Specification (Javadoc) updated to match final program.
- Test runs copied into a file (use your test plan) to show your Robot is working.
- Code for all classes.

You also need to fill in the following chart

Number of moves required for each maze. Enter the number of moves for three different runs in the random case.

	maze1	maze2	maze3	maze4	maze5	maze6
Random						
RightHandRule						
Memory						

It is possible that some of your answers will be "infinite".

3.12 Phase 3

If you worked in a team, you must individually email me an assessment of your and your partner's contribution to the assignment. The assessment should be a percentage of how the work was split. It should add to 100. In a perfect team, you would both score a 50. If you are not writing me 50, Bob 50, I'd like a sentence or two saying why. If I decide the work split was very unfair, the grades will be adjusted accordingly. Your email must have the subject 204 project: RobotMaze. This email must arrive within 1 class day of the phase 3 deadline.

If you send me an email with a subject other than the one seen here, I may choose not to count it (and I may not find it in the many incoming emails until after projects are graded). If you do not email me, I will assume you agree with your partner(s) evaluation of you. If nobody on your team emails me, I will assume everything went fine.

4 Acknowledgements

Thanks to Todd Neller and Scott Russell for their help in defining this project.

5 Save the robots!

Help them, CSCI204 coders! You're their only hope!