**Project 1 – The Maze**

This first project will allow you to practice what you have learned in the first few lessons of the AP CS; i.e. creating a new class with BlueJ, defining methods, and more.

Your first application will guide a small creature, called a MazeBot, through a predefined maze. In order to do this, your program will interact with two predefined classes; Maze and MazeBot.

To begin, copy the folder, “Project 1 – The Maze”, from the shared drive to your working directory. In that folder you will find a BlueJ project file, package.bluej. Open that file. BlueJ will start. You should see a BlueJ project window displaying an icon for a single predefined class, MyFirstProgram. We will look inside this class in a future lesson. For now, it is supplied so that you can focus on implementing your first class, MazeWalker. You will implement one method in MazeWalker, walkMaze. It will call methods in the MazeBot and Maze classes to guide the mazeBot through the maze.

**Activity 1 – Create your first class, MazeWalker**

To create the MazeWalker class:

1. In the project window, press the “New Class …” button.
2. In the Create New Class dialog, give your class the name, MazeWalker.

When BlueJ creates your MazeWalker class, it automatically includes some source code; the class declaration, a property (named x), a constructor and a method (named sampleMethod). You won’t actually use the automatically generated property or sampleMethod but MyFirstProgram will call the constructor so leave it as is. Everything you add to the class will go just before the right curly bracket at the very end of the file.

MazeWalker must have a method named walkMaze? Why? Because the code in MyFirstProgram calls that method! BlueJ will complain if you try to compile or run your program without a walkMaze method in the MazeWalker class.

In a recent lecture you learned that a method definition contains the following parts:

* The keyword, public. This indicates that other classes (e.g. MyFirstProgram) can call this method.
* The keyword, void. This indicates the method does not return a result. We will learn how to return results from methods in a subsequent lesson.
* The method name; in this case, walkMaze.
* A left parenthesis, followed by any parameters to the method, followed by a right parenthesis. walkMaze must take two parameters, a maze object and a mazeBot object. To declare a parameter you need to provide two pieces of information, the type or class name of the parameter and the name of the parameter.
* A left curly bracket, followed by the instructions that comprise the method and, finally, a right curly bracket to complete the method definition.

Put it all together and you end up with the following definition for walkMaze. Make sure that your definition matches this one:

*public void walkMaze(Maze maze, MazeBot mazeBot) {*

*// Your instructions go here*

*}*

Your program should now compile without errors. Try it. In the project window, press the Compile button. If all goes well, a message should appear at the bottom of the project window that says, “Compiling ... Done.”

Now, its time to run your program for the first time! Right click on the box labeled MyFirstProgram in the project window. A context menu will appear. On the the entries will say “void run()”. Select it with your cursor and your program will run! A new window will appear with a maze and an image that should remind you of a turtle. This is your mazeBot! In the next activity, you will guide the mazeBot through the maze.

**Activity 2 – Moving your mazeBot**

Its time to learn about the MazeBot class. It defines 3 methods that will allow you to move it through the maze; moveForward, turnLeft and turnRight. These are the simplest kind of method; they don’t require any parameters and they don’t return any results. To call a method that takes no parameters and returns no result, write a line of code that contains:

* The name of the object whose method you want to call.
* A period. This separates the object name from ...
* The name of the method
* An open parenthesis followed by a close parenthesis
* A semi-colon. This indicates the end of the method call statement.

You have now learned all you need to know to guide your mazeBot through the maze. Simply insert a sequence of instructions into your walkMaze method that moves the mazeBot and turns it as needed until it comes out the other end of the maze. I’ll give you a hint to get you started. Your mazeBot is pointed in the right direction, so your first instruction is:

*mazeBot.moveForward();*

Complete this activity by adding method call statements that move and turn the mazeBot as needed until it exits the maze.

By the way, unlike some programs, your first program does not exit when it is done. You should exit the program manually by going to the “BlueJ Virtual Machine” menu and selecting “Quit BlueJ Virtual Machine”. If you don’t, you will end up with a lot of old copies of your program cluttering your screen.

When you are done and after you hear the lecture on conditional statements, you may move on to Activity 3.

**Activity 3 – Boolean data, return values and conditional statements**

In activity 2, you guided your mazeBot through the maze. In this activity, you will instruct your mazeBot to celebrate if it successfully exits the maze and to be disappointed if it fails. mazeBot has three methods to help you accomplish this task, signalSuccess, signalError and didReachGoal. The method signatures for them are:

*public boolean didReachGoal()*

*public void signalSuccess()*

*public void signalError()*

The method, didReachGoal, returns true if the mazeBot successfully exited the maze and false, otherwise. Notice that, after the keyword public, the signature for didReachGoal contains the keyword boolean rather than the keyword void. That is how you know it returns a value. void implies no value is returned. If a value is returned, the type of the return value (in this case, boolean) is placed between public and the method name. Neither signalSuccess nor signalError return a value so their signature contains void after public and before the method name.

You learned that the syntax of a conditional statement is:

*if (<boolean test>) {*

*<success statement 1>;*

*…*

*<success statement n>;*

*}*

*else {*

*<failure statement 1>;*

*…*

*<failure statement n>;*

*}*

To complete activity 3 add, at the end of walkMaze, a conditional statement that checks to see if the mazeBot successfully exited the maze, call signalSuccess if it did and signalError if it did not.

**Activity 4 – Your first while loops**

The method, walkMaze, which you defined in activies 2 and 3 can be improved in a number of ways. It calls moveForward repeatedly – over a dozen times! It also only works for one maze. We can make it better in both of these respects. In activity 4 you will eliminate many of the calls to moveForward with the help of the while statement you learned about in a recent lecture. Then, in activity 5 you will simplify the method even further and generalize it to work with any maze. Recall that a while statement has the following syntax:

*while (<boolean test>) {*

*<repeating statement 1>;*

*…*

*<repeating statement n>;*

*}*

if the boolean test is true then your program executes all the repeating statements inside the curly brackets, just like and if statement. After the repeating statements are executed, though, the boolean test is checked again. If it is still true, the repeating statements are executed again. That test and repeat cycle keeps on going until the boolean test is false. Then, and only then, execution moves on to the next statement after the closing, right curly bracket.

In creating your first while statements, find every place in walkMaze where you have multiple calls to moveForward in a row and replace those calls with a while loop that executes as long as it is possible to move forward. mazeBot has a method, of course, to help you do just that. It is called canMoveForward. The method signature for canMoveForward is:

*public boolean canMoveForward ()*

The method, canMoveForward returns a boolean; true if the mazeBot can move forward and false if there is an obstacle in front of the mazeBot.

Now, rewrite walkMaze as described. Replace multiple, sequential calls to moveForward with a while loop that, while the mazeBot can move forward, calls the method, moveForward.

**Group Activity 5 – Designing a general maze walking method**

In activity 4, you practiced writing while statements but the resulting walkMaze method is not that much simpler and it still only works with one maze. You can write a walkMaze method that is both simpler and works with any maze. To do that, you need one more mazeBot method, didNotReachGoal. The method signature for *didNotReachGoal* is:

*public boolean didNotReachGoal ()*

*didNotReachGoal* returns a boolean, just like didReachGoal and canMoveForward. didNotReachGoal, returns a boolean value that is the opposite of didReachGoal. If didReachGoal returns true then didNotReachGoal returns false and vice versa. Shortly, you will learn about boolean expressions and you won’t really need this method but it will make your job a little simpler in this activity.

Using what you learned so far and practiced in activities 1 through 4, you can design a new walkMaze routine that is short, simple and works with any maze. There are at least two ways to do it, probably more. Break up into teams of 3 or 4 and discuss how to create such a walkMaze method. I am not going to tell you how but I will give you a few hints:

* The mazeBot you created in activity 3 has about 30 statements. This new mazeBot can have about half as many.
* Your solution will probably use one or two while statements and one or two if statements.
* My solution will used the following mazeBot methods you learned in the previous activities; moveForward, turnLeft, turnRight, canMoveForward, didNotReachGoal and signalSuccess. Your solution may use a slightly different set of methods but it will probably be quite similar in this respect.
* An if statement can have no success statements; for instance, if you only have failure statements in the else branch.
* Don’t forget to signalSuccess when your mazeBot successfully exits the maze.

Once your breakout group designs a solution, each of you should go to your computer and code it up in BlueJ.

Have fun!

**Activity 6 – Eliminating MyFirstProgram**

We provided you with the class, MyFirstProgram, at the beginning of this project so that you would not have to learn too much before you could start coding. In a recent lecture we read MyFirstProgram - all 3 statements in 1 method! Those 3 statements, though, along with the declaration of the method, run, require you to understand assignment statements, class constructors, the new operator and static methods.

To test your understanding of these concepts, add a new method, run, to the MazeWalker class. The method signature for run is:

*public static void run ()*

run contains the same 3 statements as the run method in MyFirstProgram. The first statement declares a variable, maze, to be an object whose class is Maze and assigns to it the result of calling the new operator with the Maze constructor. The second statement creates a mazeWalker object in a similar way. The third statement calls the walkMaze method of the mazeWalker object. The first parameter to walkMaze is simply the maze object you created in the first statement. The second parameter is the mazeBot, which you get by calling the getMazeBot method of the maze object.

Now you may run your program by right clicking on the box labeled MazeWalker in the project window. A context menu will appear. Select the entry labeled “void run()”. You no longer need MyFirstProgram. Feel free to delete it if you wish.

**Bonus Activity 7 – Testing you maze walker with different mazes**

The Maze class defines several constructors. The Maze constructor you used in activity 6 takes no parameters and creates the default maze object. There is another Maze constructor that takes one parameter, called the difficultyIndex. Here is the constructor declaration:

*public Maze(int difficultyIndex)*

The valid values for difficultyIndex are 0 thourgh 4. The higher the difficulty index the more difficult it is for your program to find a path through the maze. Test the general maze walking routine you created in activity 4 with all 5 of the mazes created with the different valid values for difficultyIndex.

WARNING: You can make a pretty smart mazeWalker just using what you have learned in class so far. I recommend considering this approach first. But to traverse the most difficult mazes you may need to use some things we haven’t learned, yet. Here is a hint about what those things are:

HINT: To traverse the most difficult mazes you may need to remember some things about the cells in the maze. The MazeBot class has a private instance variable, breadCrumb, that you may use for this purpose. Of course, since it is a private instance variable you must use the following getter and setter to read or write breadcrumb:

*public void setBreadCrumbObject (Object breadCrumbObject)*

setBreadCrumbObject takes the input parameter, breadCrumbObject, and associates it with the cell in the maze that currently contains the mazeBot. The parameter can be an object of any type so, for example, you can define a new class, create objects from that class and pass those objects to setBreadCrumbObject.

*public Object getBreadCrumbObject ()*

getBreadCrumbObject returns the breadCrumbObject previously associated with the cell in the maze that currently contains the mazeBot.