Programming

Robot

Labs

File structure will be represented as follows:

“***Robots\Lab00.java***” – File ***Lab00.java*** located in folder ***Robots***.

“***Robots\worlds\***” – Folder ***worlds*** located in folder ***Robots***.

Args/Arguments are synonymous for parameters

‘No-arg constructor’ is synonymous for ‘default constructor’

The code which lies between two curly brackets is referred to as the body of the method/class

public static void main(String[] args)

{

//BODY CODE…

}

Reading through the ***Background***is highly recommended if you do not understand the topic or are interested in taking *AP Computer Science A* / *Advanced Computer Science AB* in the future.

In addition, these labs will only briefly cover the basics of String manipulation, so completing additional labs focused on Strings is recommended.

Abstract classes & Interfaces do not fit into the scope of this lab but are important concepts to understand.

**Lab00**

**Start Robot**

***Objective***

The student will gain a **brief** understanding of objects, classes, and method calls.

***Background***

A great text editor for beginners in both MacOS and Windows is JGrasp, available for free at [www.jgrasp.org](https://www.jgrasp.org/) (There’s a tiny download button on the left tab, 2nd button from the top).

For more seasoned programmers, Eclipse and IntelliJ (Community Edition) are also free at [www.eclipse.org](https://www.eclipse.org/) and [www.jetbrains.com](https://www.jetbrains.com/), though the rest of this guide will be specific to JGrasp.

***Specification***

Follow “**Setting Up.pdf**” before opening up file “***Robots\Lab00.java***”.

This program has been built for you already. The breakdown of the code is as follows:

// Found in Lab00.java

// Author, Name, Date

public class Lab00

{

public static void main(String[]arg)

{

// Setup the world

World.readWorld("begin1");

World.setSize(10, 10);

World.setSpeed(7);

// Create instance of Robot

Robot karel = new Robot();

// karel commands

karel.move();

karel.move();

karel.move();

karel.move();

karel.turnLeft();

karel.move();

karel.move();

karel.move();

karel.turnLeft();

karel.move();

karel.move();

karel.move();

karel.pickBeeper();

// Output on completion

System.out.println("Finished");

}

}

The class name **must** match the file name.

public class Lab00 **must be in** Lab00.java

Comments are preceded with double slashes.

//Comments are ignored when the program runs

All executable code statements must end with a semi-colon ;

The **main** method which the compiler looks for first. Code in this method is executed first. **MUST** be as follows:

public static void **main**(String[]arg){

// Code…

}

Methods are created through the following parameters:

public – access modifier denoting that this class can be accessed from anywhere (including other classes/files)

static – optional keyword denoting that this method is belongs to the class and rather than an object. This type of method is called without the instantiation of an object

void – keyword used to specify what the method will return. These can be primitives, objects, or void.

main – name of the method

(String[] arg) – The input parameter(s) and name of the parameter(s). There may be multiple parameters\*

All methods will end with a pair of parentheses. These contain the method parameters or are left empty ()

The . at the end of an object/class denotes that what follows belongs to that object/class. This can be a method or a constant of that object/class

The public static methods readWorld, setSize, & setSpeed belong to the class World. Open World.java to view these methods. Since these are static methods, there is no World object which needs to be instantiated.

The non-static public methods move, turnLeft, & pickBeeper belong to karel, which is a Robot

object. Open “***Robot.java***” to view these methods. Since these are not static methods, they belong to a Robot object rather than the Robot class. As such, we must create a Robot object. Keep in mind that this is because the methods we want to call do not belong to a class, but an object.

**Object Creation**: This is seen when we create a Robot object called karel through the following line of code:

Robot karel = new Robot();

Robot – declaration of the type of object that will be created

karel – name used to refer to the variable of the preceded object

new – tells Java to instantiate a new object by first allocating memory

Robot() – initializes an object by calling the constructor for the object specified

A constructor is a special method for objects. It is used to create an object and pass in parameters. In the example above, this constructor does not have any parameters. This is known as a **default** **constructor/no-argument constructor** (No Parameters).

Did you get why constructors are called what they are? This is because they are methods which pass along information (or no information) to construct an object.

Note: Constructors for objects will always have the same name as the object itself, followed by parentheses, which may or may not contain parameters inside them.

You may have noticed that your folder now contains numerous files ending in *.class.* These can be ignored – they are the product of the Java compiler. While unreadable to us, this contains bytecode the JVM reads to execute our code.

\*The parameters for the main method are: String[] arg because this allows the JVM (Java Virtual Machine) to pass parameters into the program when run. The usage of arg in the program is optional and typically is not used.

**Lab01**

**Robot Basics**

***Objective***

The student will gain an understanding of objects, classes, and method calls.

***Background***

Worlds are divided into streets and avenues. Streets run West to East (left to right). Avenues run South to North (down to up). Streets are numbered on the vertical axis starting with 1st Street, and avenues are on the horizontal axis starting with 1st Avenue. The corner at the bottom-left of the graphics window is (1st Avenue, 1st Street), or (1, 1).

Worlds create the context for solving robot problems. Pre-defined worlds are stored in the folder “***Robots\worlds\***”.

An *identifier* is the name of a class, an object, or a method. An identifier can be any unique sequence of numbers, letters, and the underscore character ( \_ ). An identifier **must** begin with a letter. An identifier **cannot** be a Java keyword, like class, new, or public. Identifiers are case-sensitive, so ‘lisa’ is not the same as ‘Lisa’. As a convention, only class names begin with an uppercase letter. Method names, objects, and other variables always begin with a lowercase letter. For multi-word situations, the lowerCamelCase naming convention is typically used. One exception is constants, like EAST, which are written in ALL CAPS. Constants are variables with final values.

Once the new operator creates a robot object, that object can perform robot methods using dot-notation. For instance, a robot named ‘pete’ will move forward with the command pete.move();. The identifier ‘pete’, before the dot, is the name of an object. The identifier ‘move’, after the dot, is the name of a method belonging to the robot ‘pete’.

***Specification***

Create “***Robots\Lab01a.java***”. This file must be located in your main project folder. For guidance on creating files, please revisit “***Setting Up.pdf***”

Create a class called “***Lab01a.java***”. Make sure that you have beginning and ending curly brackets to denote the body of the class – All defined methods, classes, and loops will require curly brackets.

public class example {

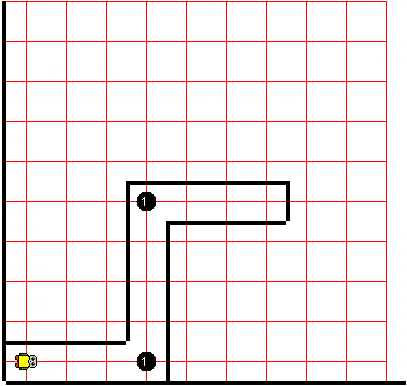
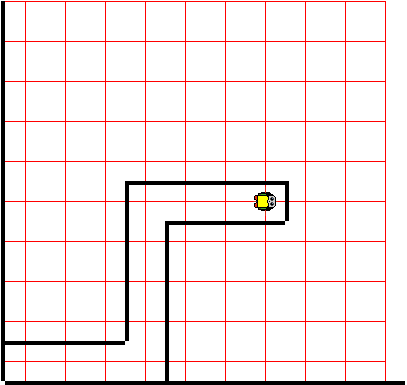
//code…

}

Create the main method and read in the “beginner” world with size 10x10, running at speed 7. Declare a Robot object called ‘karel’ using the default constructor. This places ‘karel’ at (0, 0) facing East with 0 beepers. Have ‘karel’ traverse the path (without bumping into the walls) and pick up all the beepers. Once it reaches the end, ‘karel’ should drop all of its beepers.

Use the previous lab for reference; Use method putBeeper() to place beepers on the road.

Start Finish



Now create file “***Lab01b.java***”. Copy and paste your code from the first part of this lab into the new file. Modify your declaration statement for the Robot by using the 4-arg constructor by following the following example below, as well as the provided guidelines.

Start the robot at (7,5), facing direction WEST with 0 beepers.

Example

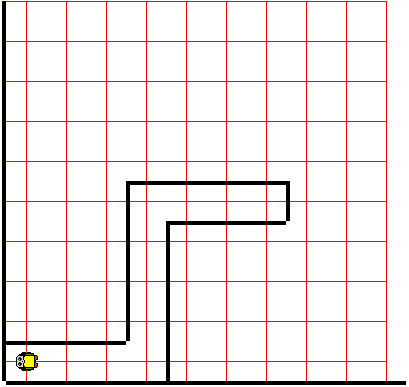
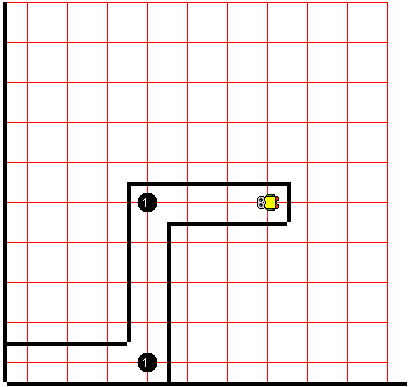
Creates a new Robot at (1, 1) facing EAST with 0 beepers.

Robot karel = new Robot(1,1,World.EAST,0)



Now run your program. This should perform the same actions as the first part of this lab, only in reverse.

Start Finish



**Lab02**

**Inheritance & Methods**

***Objective***

The student will gain an understanding of inheritance and methods.

***Background***

Escaping the maze shown below involves both left turns and right turns. Since our robots only know how to turn left we will define another type of object (athlete) in a separate class (Athlete) that knows how to turn right and 180 degrees.

Since we want our new object, Athlete, to inherit the methods contained in the Robot object (turnLeft(), pickBeeper(), putBeeper(), etc.) , we will need to use the term “**extends**” when creating our class so that Athlete is an extension of the class we want to inherit, Robot.

Since we are completely extending/inheriting a class, we will also need to invoke the keyword “super()” in our new Athlete class’s constructor. This calls the parent constructor, AKA the class we are extending (Robot). Simply said, in order for us to inherit the functionalities of a Robot object, we have to create an instance of it using the super() constructor in our Athlete’s constructor. This way, we have a Robot object AND whatever other methods we want, all in a single Athlete object. Hence, Athlete is an extension of Robot.

The guidelines used to define a method is also used when creating the main method. To create a method, use the following format:

<visibility> <return type> <name of method> (*params*)

Ex:

public void turnRight(){//Code…}

public int addNumbers(int a, int b){//Code…}

public class Athlete extends Robot

{

public Athlete()

{

super(1,1,World.EAST,0);

}

public void turnRight()

{

//complete this method

}

public void turnAround()

{

//complete this method

}

}

The Athlete class is inheriting the Robot class – Athlete contains all of Robot’s methods

In the Athlete constructor, we must call the super( ) constructor to create a Robot instance and inherit it.

This method is defined as follows:  
Visibility: public

Returns: void (nothing)

Name: “turnRight”

This method is defined as follows:

Visibility: public

Returns: void (nothing)

Name: “turnAround”

Take note that since Athlete extends Robot, we can refer to Robot in this instance as the parent class, and Athlete as a child class. This is because Athlete is a Robot, but Robot is not necessarily an Athlete.

A similar analogy would be with squares & rectangles: A square is a rectangle by definition, but a rectangle is not necessarily a square.

***Specification***

Create “***Robots\Athlete.java***”.

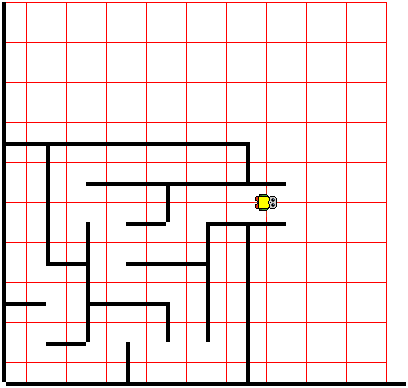
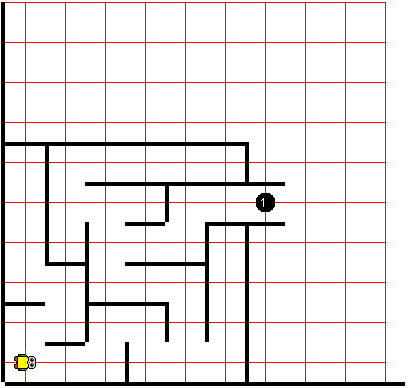
Using the above example code, finish creating the Athlete class, defining the methods turnRight() and turnaround()

Note: You can use methods in methods. When calling methods from the same class, it is not associated with an object, so you can call the method outright (It doesn’t need to be an object’s method) since it is available in that class. Also, remember that when extending a class, you inherit ALL of that class’s available methods.

Now, create “***Robots\Lab02.java***”.

Create the main method and read in the “maze1” world with size 10x10, running at speed 7. Declare an Athlete object called ‘karel’ using the default constructor. Have ‘karel’ travel to the end of the maze. Once ‘karel’ reaches the end of the maze, it should stop and pick up the beeper.

Start Finish



**Lab03**

**Parameters & For-Loops**

***Objective***

The student will gain an understanding of method parameters, method overloading, and for-loops.

***Background***

Inheritance and making methods makes our lives a lot easier by reducing the amount of code we have to write. Instead of having to keep writing code that performs the same function over and over again, we can use methods to represent these functionalities and simplify our code, making it more readable. However, this doesn’t give us the versatility we need.

Looking back at our Athlete class, it appears that we have hard-coded the robot’s starting location. If we want to create an Athlete object at (5,5), how could we do that? The only way this is possible is if we modify our super() statement. This makes it hard for us to use, so this is not the best solution.

You should already know what parameters are – you specify them when you’re loading in the world to use, the size, and the speed. These are all parameters. If we incorporate parameters into our methods, we can easily get around this hard-code limitation.

A good coding practice is to stray away from hard-coding elements into your program, which frequently means the solution is to use method parameters.

Loops are an essential part of programming – if you need to move 10 times consecutively, why tell ‘karel’ to move 10 individual times when you could instead loop a single move command 10 times? This significantly reduces the amount of code you have to write and makes it much more readable.

A for-loop has 3 parts to it, each separated by semicolons:

for(Initialization ; conditional ; incrementer){

//Code…

}

For the above for-loop:

* ‘Initialization’ – This is code that is typically used to create a variable which stores the number of times the loop has been executed by the program. This is executed ONCE when the program initially encounters the loop.
* ‘Conditional’ – This is the statement the program uses to determine if it should enter the body of the loop or not. This statement is checked before EACH loop. When true, the program enters the loop. When false, the program skips the entire loop.
* ‘Incrementer’ – This is code that is typically used to change/increment a counting variable. This is executed at the END of EACH loop, provided that the program entered the loop and finished executing everything inside. Afterwards, the program checks the conditional statement again, repeating the whole cycle.

Example for-loop which loops 10 times:

int index = 0 – Initialization

for(int index = 0 ; index < 10 ; index++)

{

//Code to loop…

}

index < 10 – Conditional statement

index++ - Increments the variable index by 1

Loops again until the conditional isn’t true

Typically, you will see people use ‘i’ instead of ‘index’ because it is much easier to type.

***Specification***

Open “***Robots\Athlete.java***” and add another constructor that is a 4-arg constructor. Then, invoke the super() constructor using these parameters. If you need a reminder of what the Robot 4-arg constructor looks like, open the “***Robots\Robot.java***” file and view its 4-arg constructor for it there.

Note 1: When looking at the “**Robot.java**” class, take a look at the no-arg constructor,

“public Robot(){//Code…}”

Take a closer look at the body statement of it. You should notice that when we don’t provide the constructor with any parameters, it has its own hard-coded default parameters.

Note 2: You can have more than 1 constructor in the same class, provided that each constructor has different parameters (Each constructor must be unique).

Ex: String vs int, 1-arg vs 2-arg, etc.

You should now have 2 constructors for Athlete, a no-arg constructor and a 4-arg constructor.

Now, let’s cut down on the code we have to write by extending Athlete with another class – a Runner.

Create class “***Robots\Runner.java***” that extends Athlete. Now create 2 constructors, a no-arg constructor and a 4-arg constructor. Arguments should be used to invoke the super() constructor.

Create a new method run(int x). This method has 1-arg, an integer ***x***. Define the method so when called, the method will move the Runner ***x*** number of times.

Now create another new method run(). As this method has no-args, there are no parameters given in. As such, we will need to define a default movement. When called with no parameter, move the Runner 1 time.

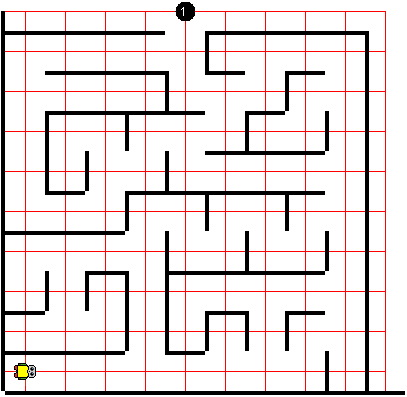
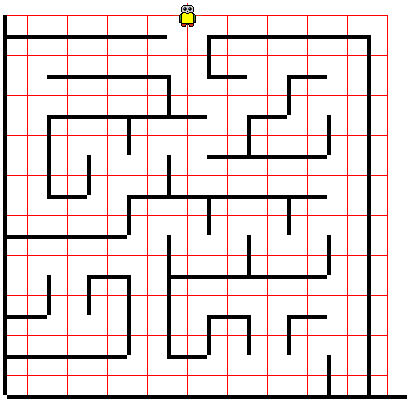
Note 3: You can have more than 1 method with the same name in the same class, provided that each method is unique – here we have methods with the same name, but one has no parameters and the other has an integer parameter. This is commonly referred to as “**Method** **Overloading**”

Hint: These methods should not return any information.

Create “***Robots\Lab03.java***”.

Create the main method and read in the “micromouse1” world with size 10x10. Declare a Runner object using the default constructor. Have the runner travel to the end of the maze. Once ‘karel’ reaches the end of the maze, it should stop and pick up the beeper. Remember to use the new method you just defined in Runner, using parameters to specify the distance you want the runner to move.

Start Finish



**Lab04**

**Overriding & While-Loops**

***Objective***

The student will gain an understanding of method overriding and while-loops.

***Background***

Although the Runner can move as many times we tell it to move when we give it a parameter, it is possible to automate this process and simply tell the Runner that we want to move forwards until we physically cannot due to a wall. This can be achieved using while-loops:

while(conditional){

//Code…

}

This loop above will continue to loop until the conditional is no longer true. At that point, the program doesn’t enter the loop, and instead skips it entirely.

Sometimes, it will be more intuitive to expand on an existing method that is inherited. This is a perfect example of that scenario – looking back at the previous lab when we created the Runner class, we made a method called run() with no parameters. It seems most fitting that we give the above functionality to this method run(). In this case, we will have to override an existing method and replace it with new code that fits our function better, commonly known as “**Method Overriding**”.

public class RoboRunner extends Runner

{

// Constructors…

@Override

public void run()

{

//complete this method

}

}

The RoboRunner class inherits all of Runner’s methods, which includes its run()method.

Since it’s more intuitive for us to change this method in RoboRunner to fit our functionality, we should override this method, changing what the method does in RoboRunner. However, the functionality of this method in Runner still stays the same.

***Specification***

Create “***Robots\RoboRunner.java***”.

Using the above example code as a template, finish creating the class by creating a no-arg and a 4-arg constructor, invoking the super()keyword.

Now, override the method run() and give it a new functionality – instead of moving 1 time when we call it, the method should cause the RoboRunner to keep moving until a wall has been detected. Only then will the RoboRunner stop moving.

Hints:

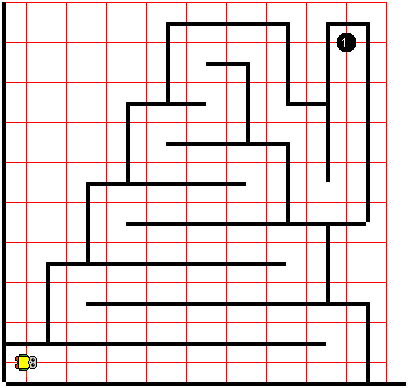
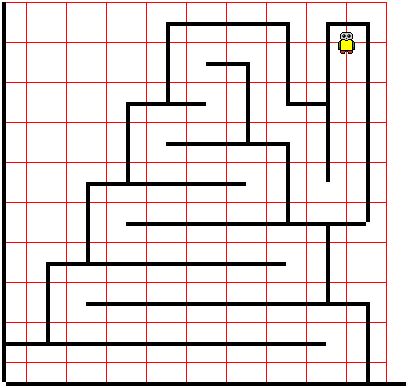
Visit the “***Robot.java***” class starting at line 87 to view methods defined for the Robot class which are also inherited by all of its child classes.

public boolean method(){//Code…} – Since the return type is “boolean”, this method will return only true/false, which is what a while-loop requires in its conditional.

Create “***Robots\Lab04.java***”.

Create the main method and read in the “tower1” world with size 10x10. Declare a RoboRunner object using the default constructor. Have the roborunner travel to the end of the maze. Once it reaches the end of the maze, it should stop and pick up the beeper.

Now that your run() method allows the RoboRunner to move forwards until it hits a wall, you don’t have to specify the number of times the RoboRunner has to move.

Start Finish

**Lab05**

**Thinking & Application**

***Objective***

The student will gain an understanding of a basic algorithm and will practice applying what they’ve learned

***Background***

Algorithms are a set of rules we can follow to solve problems in varying situations. When implemented, they become a set of procedures that the program follows in order to complete a task. In this lab, you’re going to consider what you’ve learned from the previous labs and try to figure our how to implement an algorithm on your own!

Now that our RoboRunner is able to move forwards without us telling it how it should move, the next step is to get it to turn without us telling it when and how to turn.

***Specification***

Open “***Robots\RoboRunner.java***”.

Create a new method called completeMaze(). This method should not take in any parameters and will not return any value. When called, this method should complete the maze and pick up the beeper at the end of it, then stop.

Using your knowledge of while-loops , method-calls, and conditional statements, implement an algorithm to get out of the maze and stop on the beeper.

Remember: If you need to see what methods Robot and its children have been given, view the file “***Robot.java***”.

If you stick to one side, you will always find the end of a maze!

1) If your right is clear, what should you do to get a wall on your right side?

2) Otherwise, if your front is clear (and assuming your right is not), what should you do?

3) Otherwise, we can now assume that our front and right is blocked, so what do you do?

4) Repeat steps 1-3 until the beeper is reached

Hints:

When taking the opposite of a boolean, use ‘!’ in front of the expression to change it.

For example, to change the result of a method to the opposite boolean value, you could use !isEven(). When this method returns true, the ! sign will turn it into false.

Feel free to use if-statements. They are similar to while-loops, except they do not loop.

Ex: if (isEven()){// Code…}

if-elseif statements are also intuitive to use:

if (isZero()){

// Code…

} else if (isNegative()){

// Code…

}

else{

// Code…

}

When using these if-elseif-else statements, there are certain guidelines you need to keep in mind

* An else-statement is only applicable if preceded with an if or elseif statement.
* The else-statement must be last
* Once an if-statement has been entered, the rest of the chain of statements will be skipped
* When an if-statement is not true, it will enter the next available if-statement, or the else-statement if it is last.

In a conditional statement, it’s possible to chain multiple conditions together with an AND, or an OR.

&& (AND) – Requires both conditions to be true

|| (OR) – Requires only 1 condition to be true

if (isPositive() && sqrtIsEven()){

// Code…

}

When you have multiple conditions chained together, they will execute in the order that they are written in. For the above example, if we input the number 9, since the number is positive, it passes the first condition and moves to the second. Since the square root of 9 is 3, and 3 is not even, it doesn’t pass the whole if statement.

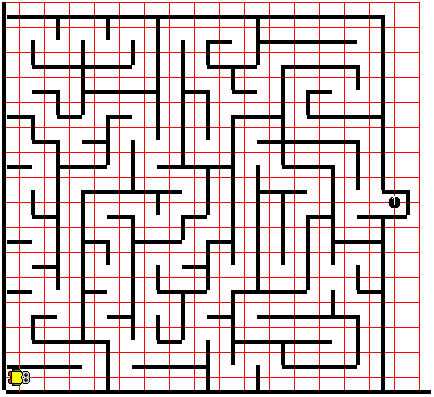
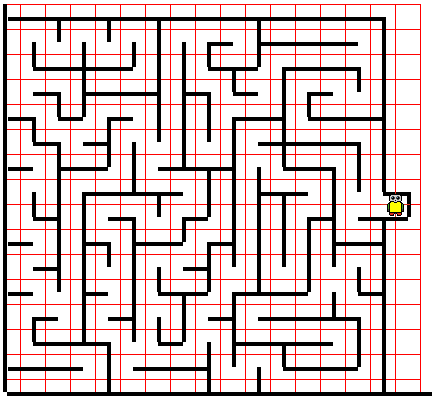
If we try passing in -16, this does not pass the first condition because -16 is a negative number. Because of this, -16 does not reach the second condition. Had -16 made it to the second condition, the program would throw an error because you cannot take the square root of -16.

Knowing that the order matters in conditional statements is important – this helps us structure our code so that we avoid throwing errors.

Create “***Robots\Lab05.java***”.

Create the main method and read in the “tower1” world with size 10x10. Declare a RoboRunner object using the default constructor. Call the method completeMaze(). When run, this should cause the RoboRunner to complete the maze.

If this works, change the map read in the “bigmaze1” world with size 17x16. Your program should still be able to complete the maze. Feel free to increase the speed if this takes too long.

 Start Finish

**Lab06**

**Static Keyword**

***Objective***

The student will gain an understanding of the differences between static and non-static methods & variables.

***Background***

An instance method is invoked on an object. This type of method MUST be associated with an object. For instance, pickBeeper() is an instance method belonging to the Robot class. This type of method is considered to be non-static because it can only be called with a Robot object.

A traditional static method is an example of a class method. This type of method belongs only to that class and is not associated with any objects. As such, these types of methods can be called without creating any objects. An example of a static method is the main method, public static void main(String[] args){//Code…}.

public class test{

public static void main(String[] args)

{

// Code…

}

public static int addNumbers(int a, int b)

{

// Code…

}

}

Since the main method is not associated with any objects, it is a static method

Since the method addNumbers(int a, int b) is not associated with any objects, it belongs to the class test, so this is also a static method. This method can be called without creating any objects at all.

Static variables also serve another purpose – when there are multiple objects of the same type created in a single program, regular variables are always limited to the scope of that object, and nothing else. Static variables, however, serve to unify the multiple objects in a single program. These types of variables are not unique to each different object; instead, they are present for all of the instances of that object. Each object of the same type has access to this variable, and the value is publicly shared between all the objects.

For instance, if we have a program designed to assign each student a unique ID number, every time we add a new Student object to the program it will immediately give that Student object its own ID. How could we keep track of the previous student’s ID number? We could use static variables here – if we simply assign the new Student an ID number that is 1 above the previous student’s ID number, we could create a static variable that is present in all of the Student objects and increment the variable by 1 after the student is added.

public class Robot{

private static int numberOfRobots = 0;

…

private int identifier;

public Robot(…)

{

…

ID = numberOfRobots++;

}

}

If we were trying to keep track of the ID’s of each robot, we’d define a static variable in the Robot class and increment that variable each time we created a new Robot object.

Did you know that you can increment a variable while using it on the end of a copying operation? In ID = numberOfRobots++, the operation is executed in the order of the code. In this instance, ID sets itself equal to the current value of numberOfRobots before it increments numberOfRobots by 1.

***Specification***

Create “***Robots\Lab06.java***”.

Create the main method and read in the “cleaner1” world with size 10x10. Create 10 Runner objects at (1,1), (1,2), (1,3), (1,4), etc.. Each runner should face EAST with 0 beepers.

Now, create an additional static method in the file called clean(). This method should have 1 parameter, an Runner object. Once it takes in the Runner object, it should loop to tell each Runner to move to the right side of the screen, picking up all beepers it encounters. Once it reaches the end, the Runner should turn around and return to its starting position. Once it reaches its original position, drop all beepers.

Call this static method on every Runner.

Hint:

There is an easier way to instantiate Runner objects than by creating each object one by one. This can be achieved by creating a Runner object that isn’t associated with a variable name.

Here, we are passing an anonymous Runner object into the action(Runner) method. This allows us to call this method and create many Runner objects by calling the method multiple times in a loop.

public class test{

public static void main(String[] args)

{

action(new Runner(1,1,World.EAST,0);

}

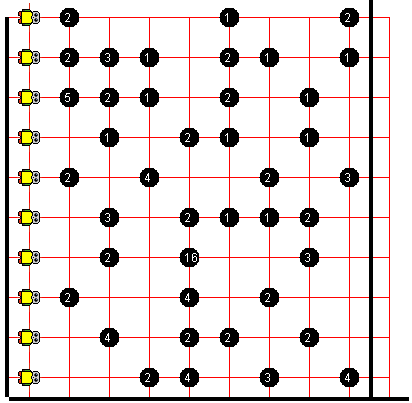
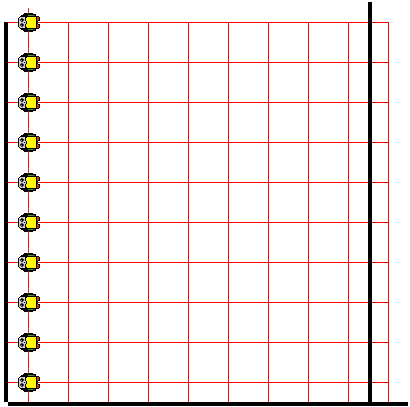
public static void action(Runner runner){

// Code…

}

}

Start Finish



**Lab07**

**User Input**

***Objective***

The student will gain an understanding of imports and how to use the Scanner class.

***Background***

Most programs which users use do not run on their own – rather, they depend on user input so that the program understands what the user wants it to do. This gives the program much more functionality since there’s a multitude of different inputs the user can give the program.

The most common way to receive user input in the console is through the usage of a Scanner object. The Scanner class, however, is not accessible without importing it first. Many different libraries and functions are stowed away in Java’s many libraries, but to save space, they are not loaded into the program unless the programmer chooses to import them.

Imports are always placed at the very beginning of the program – the easiest way to import these libraries is to import java.util.\* and java.io.\* . The star indicates that EVERYTHING under java’s ‘IO’ and ‘Util’ libraries will be loaded in.

When creating the Scanner , the constructor requires a single parameter specifying where the Scanner is receiving its information from. In our case, since we want to receive user inputs from the console, we will use System.in.

In addition to receiving user input, it’s also important to be able to output to the console too – this informs the user about any information the program will need, or just sends useful information to the user.

To send information out to the user, we will have to use the standard output library. The 2 most common commands to output to the console are as follows:

System.out.println(“TEXT”); - This prints your text onto its own line

System.out.print(“TEXT”); - This prints out text without putting it onto its own line

import java.util.\*;

import java.io.\*;

public class test{

public static void main(String[] args)

{

System.out.println(“Enter a number”);

Scanner input = new Scanner(System.in);

int number = input.nextInt();

// Code…

}

}

Remember – imports are always at the very start of a file, and \* indicates that you want to import everything from that library into the program.

When you want to get information from the user, it’s important to inform the user what info you want, hence the usage of System.out.println(INFO) here

\*Keep in mind that when you are outputting text through a String, you must surround your text with quotations marks “TEXT” to indicate that you are outputting a string and not referring to a variable

input.nextInt() takes the next integer input into the console

For a more extensive list of Scanner methods, view them [here](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html) in the Java API (Scroll down to “Method Summary”).

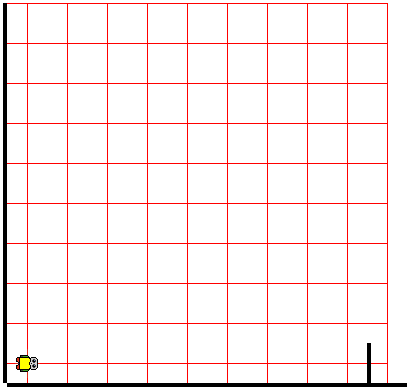
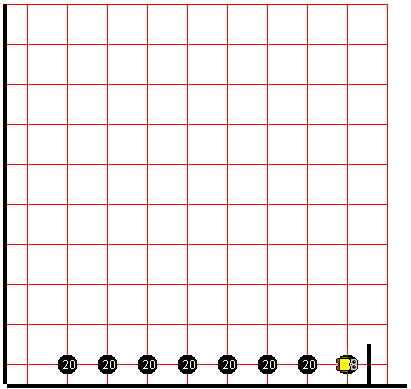
***Specification***

Create “***Robots\Lab07.java***”.

Create the main method and read in the “empty1” world with size 10x10. Create an Athlete object at (1,1) facing EAST with infinite beepers. This can be indicated using World.INFINITY .

Create your Scanner object and ask the user for the number of beepers that they would like to drop. Once you receive the input, move the athlete to the right until it reaches the wall. After each move, drop the number of beepers that the user initially input.

Start Finish



**Lab08**

**Strings & Equality**

***Objective***

The student will gain an understanding of concatenation and content equality versus reference equality

***Background***

Strings – an object that stores a sequence of various characters. This is the most common medium used for storing phrases, outputs, and texts.

Example of String objects:

String output = “Hello World!”;

String question = “Can you spare $20?”;

You can combine different variables with different values together to **concatenate** them by using a generic addition operator between them.

String first = “This is an ”;

String last = “example of”;

String combined = first + last + “ concatenation!”;

System.out.println(combined);

Output below:

This is an example of concatenation!

If you want to add a blank line, for example, the character literal for this in a String is ‘n’. In order for us to properly call a line break, we must use the following: “\n”.

If we want to call a string literal (A character in a string with special meaning), we must precede the character literal with a ‘\’. This symbol acts as the **escape sequence**, indicating that we want to invoke the string literal’s function. The reason this is called an “escape sequence” is because this tells the Java Compiler we wish to deviate from the standard function of the character sequence that follows. This can be thought of as the shortcut keys on a keyboard – if we want to perform an alternative function, we could press ctrl + character sequence to execute a command.

String out = “Joe: Hey, how are you?\nJames: Great! Yourself?\nJoe: Amazing!”

System.out.println(out);

Output below:

Joe: Hey, how are you?

James: Great! Yourself?

Joe: Amazing!

[A more extensive list of string literals](https://www.tutorialspoint.com/escape-sequences-in-java).

In addition to string literals, the escape sequence can be used to indicate that we wish to use some reserved characters as text. For instance, if we wish to put quotation marks in a String, we would do so as follows:

System.out.println(“Joe said, \”I don’t care about your elbow!\””);

Output below:

Joe said, “I don’t care about your elbow!”

As you might know, checking for equality between primitives (Integer, Double, etc.) all use the operator ‘==’. However, since String is an object, checking for equality is different here. When checking to see if an object has the same content as one another, we must use the *Object*.equals(*Object*) method. This is available on every object.

String text = “This is joe”;

System.out.println(text.equals(“This is joe”)); // True

In this scenario, the equals() method is being called on 2 strings to check if both their values are equal.

Why can’t we use the regular ‘==’ to check for equality between objects? This is an advanced topic, but the brief rundown is as follows: While the method equals() checks for literal equality between two objects, the ‘==’ operator checks for **reference** equality. Reference is the object that a variable points to. If we have two completely different objects, then they would not have the same reference equality.

Imagine we have a Car object.

public static void main(String[] args)

{

Car Mom = new Car(“VW”);

Car James = new Car(“VW”);

System.out.println(Mom.equals(James)); //True

System.out.println(BMW == VW); //False

}

Here we are creating 2 objects Mom and James. Their content is the same because they both create a VW Car object. As such, both Mom & James have the same type of car, hence the cars’ contents are equal.

With the reference equality check, we are checking if both objects are referring to the same object. However, since both of these objects are instantiated separately, each variable is pointing to their own car.

Confused? Here’s an example using a reference

Here we are creating 2 objects Mom and James, though we are only instantiating Mom’s car. Instead of creating a new Car object for James, we simply set it equal to Mom’s car. This means that the content of both cars will be equal, and the reference for both cars will also be equal. This is because Mom’s car is also James’s car. Therefore, both variables point to the same car.

main

{

Car Mom = new Car();

Car James = Mom;

System.out.println(James == Mom); //False

}

Note: Since primitives are NOT objects, when set to another variable’s value, it copies that other variable’s entire value. Also keep in mind that primitives don’t have methods, such as equals().

public static void main(String[] args)

{

int x = 10;

int y = x;

System.out.println(y); //10

}

When setting y = x, the entire variable x’s value is copied. No reference is given.

[A more extensive list of String methods](https://www.geeksforgeeks.org/string-class-in-java/).

***Specification***

Create “***Robots\Lab08.java***”.

Ask the user for the world they would like to input, and then load in that world. Use the standard world size of 10x10 and set the speed to 10 as well.

Create a new Athlete object at the standard location and orientation with infinite beepers.

Now, create a loop that continuously takes in user input until the user inputs the word “stop”. When this happens, exit the program, closing it.

If the user does not input the termination phrase, take their input and perform the method that corresponds to that input. For example: If the user inputs “turn left”, then the Athlete object should turn left. The user should be able to turn, move, and pick/put beepers. If for whatever reason the user’s input does not correspond to any method, then tell the user that their input was invalid and they should try again.

This program should essentially allow the user to command the Athlete to move from the console.

Now try to solve maze1 & maze2 by manually typing in the commands for the robot through the console. You should be able to reach the end of the maze and terminate the program when you’re done!

Hint: To terminate the program, call System.exit(0);

If you wish to remove the case sensitivity, then call the String’s method .toLowerCase() on the user’s input.

You can also disable logging the Robot’s current status by calling World.disableLogging(). This will reduce the clutter in the console.

No images will be provided as this should be self-explanatory.

**Lab09**

**Getters & Setters**

***Objective***

The student will gain an understanding of common programming practices; specifically, accessor and mutator methods for private variables in objects. They will also practice using return statements.

***Background***

You may know that one of the many reasons why people use Java is because it is an ‘Object-Oriented Programming’ language, or OOP. This makes data manipulation easy, but sometimes there will be conflicts between scopes of variables and what a programmer may want to access. To prevent this issue, variables that are specific to an object should be declared as private variables.

As such, it is regular programming practice to use Getter and Setter methods in an object to receive and manipulate variables. This ensures that the programmer will never accidentally modify variables they should not be modifying and streamlines the process of mutating and getting information. This is because experienced programmers will know that if they want to get/set information, they can just look at the getter/setter methods to see what they can do with the class.

public class Student{

private int ID;

private int name;

public Student(String n, int id){

ID = id;

name = n;

}

public int getID(){

return ID;

}

public int getName(){

return name;

}

public void setID(int id){

ID = id;

}

public void setName(String n){

name = n;

}

}

Variables specific to an object should always be declared at the top of a program before the constructor(s). Most variables should be kept private if they are specific to each instance of the object (Student IDs, credit card numbers).

Then, in the constructor, the variables are instantiated with values.

Getter/Setter methods should be named intuitively so that anyone can tell what the method does.

Getter methods will always return something

Setter methods will always modify a variable, but might not return something.

Since the setter methods don’t return anything, their return type is specified as ‘void’, so these don’t need a return statement.

Note: When you specify that a method will return something that isn’t ‘void’, you MUST return that ‘something’ in the method, or else the program will not compile and run.

Remember – imports are always at the very start of a file, and \* indicates that you want to import everything from that library into the program.

***Specification***

Create “***Robots\Tracker.java***”.

Tracker should extend Runner. This should use a 4-arg constructor. The class should also have variables to keep track of the beeper count and its position. Override its move method so that when it’s on a pile of beeper, it counts the number of beepers and adds it to its overall beeper count. In addition, it should output the number of beepers and its position, if beepers are present.

Don’t forget to create the appropriate getter/setter methods for each private variable.

Create “***Robots\Lab09.java***”.

Ask for the world and use size 10x10. Create 10 Tracker objects at (1,1), (1,2), (1,3), (1,4), etc.. Each Tracker should face EAST with 0 beepers. Disable logging to the console.

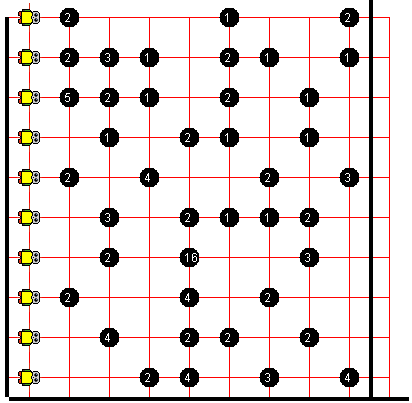
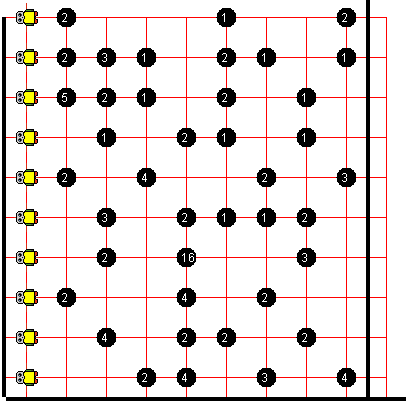
Now, each Tracker should move from the left side to the right and return to its origin position. At the end, output the total number of beepers present on the screen.

Hint: You can keep track of direction by representing each direction (Left/Right) as a number (1, 2).

Since you know that keyword super calls the parent class, you could call the previous class’s methods by using super.method()

Test this using world “cleaner1”.

Start Finish



**Lab10**

**Practical Methods**

***Objective***

The student will practice applying their knowledge to solve a line maze.

***Background***

Now that you’ve learned the fundamentals of objects, we’re going to try a unique problem. You will be given no detailed instructions for this lab.

In this lab, you will need to trace a line of beepers from start to finish by creating new methods.

***Specification***

Create “***Robots\Detector.java***”.

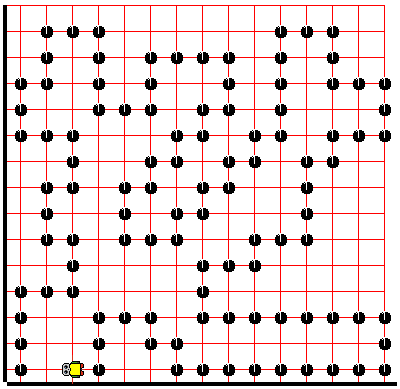
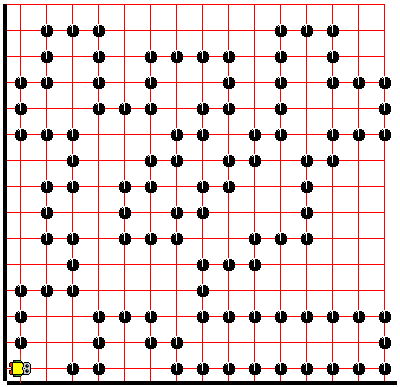
Runner should be the parent class of Detector. Create methods that check the front and sides of the Detector’s current position for beepers. Override methods as needed.

Create “***Robots\Lab10.java***”.

Ask for the world and use size 15x15. Create a Detector with the default constructor. Go to the end of the beeper path. Upon completion, print out the total number of beepers in the path. Use world “beeperpath1”.

Hint: Don’t bump into walls.

Start Finish



**Lab11**

**Arrays**

***Objective***

The student will learn the basics of arrays and how to retrieve/set data.

***Background***

You’ve learned several basic ways to store data – primitives & objects are just the basics of many larger data structures. One of the basic and most common data structures used anywhere are arrays. Arrays can be thought of as a list of elements. This list has a fixed size, and each element has its own index to go along with it. The index is the location in the array that the element is at, which is how we would access and mutate it.

When declaring an array, you must denote this by using []. This is similar to how you would create a regular variable –

int example = …;

However, to create an array, you must tack on the brackets [] to specify that you are declaring and integer array.

int[] array = …;

What goes on the right side? To initialize an array, you must specify the type of array you want to create, and put the size inside the brackets:

int[] array = new int[10]; - This creates an integer array of size 10.

Basic syntax to create an array is as follows:

Type[] name = new Type[size];

Type name[] = new Type[size];

It’s also possible to create an array by field initialization (at declaration):

Type[] name = {element1, element2, element3,…};

Type[] name = new Type[]{element1, element2, element3,…};

It’s recommended that for the above 2 scenarios you use the first example because it is simpler and more intuitive to use.

It is also possible to declare an array without actually initializing it:

Type[] name; // Declaration

Type = new Type[size]; // Initialization

Note that when you declare an object without initializing it as well, its value is set to null. This is useful when creating a private array in an object and you want to finish initializing it in the constructor.

Arrays are not limited to primitives, however – they are also applicable to objects as well.

Example:

Car[] – Car Array

Car[] cars = new Car[6];

cars – name of the car array

new Car[6] – new car array of size 6

To better understand this example, let’s read it in words:

“We are declaring a car array called cars and initializing it to a new car array of size 6.”

As you can see, creating arrays doesn’t have to be hard – it’s actually very intuitive.

When you create an array, you now have an array that is empty – but is it truly empty? Not exactly. When creating a primitive array, each index is set to a default value. This value varies from primitive to primitive, but generally this will be 0 and false. So if we created an integer array, all indices of the array will be set to 0 automatically, and for a boolean array every index would be set to false.

However, when creating an object array, such as a String[], since String is an object, the value of each index in the array will be automatically set to null. Any other objects will have a default value of null as well.

In addition, arrays start at indices of 0. What does this mean? If we had an array of size 10, we would have elements in indices ranging from 0 to 9.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Index

Array

Now that you know how to create an array, it’s important that you understand how to use an array.

Each element in an array is manipulated just as a regular variable would be. As a matter of fact, manipulating an element in an array is almost identical to that of a regular variable. The only difference is the way you would call the element. Instead of identifying each element by a specific variable name, we can identify each element by the overall name of the array and that element’s specific index in the array.

If we had an integer in an int[] called array at index 6, we could access this element by calling: **array[6]**

That’s all you need to know about manipulating elements in an array – now that you know how to call an element in an array, you can change the value by setting it equal to another value, or get the value simply by calling the element and using it like a regular variable.

Now, let’s say you wanted to print out every element in an array – you would have to iterate through each element in the array and print them out individually. To get the size of an array, access the length property of the array by calling

**array.length**

Note that this is not a method – this is actually a variable in the array that you’re accessing. Since the array’s size is fixed, this value will never change

To iterate through the entire array, use a loop to go through each element in the array:

for(int i = 0; i < array.length ; i++){

System.out.println(array[i]);

}

Example:

Let’s say I wanted to store a list of Car objects – I could store these using an array structure. In addition, I could declare each Car object inside of the array so that each Car would be identified not by a variable name, but instead by the array name and index.

Car[] cars = new Car[6];

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] | [5] |
| null | null | null | null | null | null |

We now have an array of size 6, and each element’s value is null

Since each element is set to null, I have to iterate through each element and instantiate each Car object.

for(int i = 0; i < cars.length ; i++){

cars[i] = new Car();

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] | [5] |
| KIA | KIA | KIA | KIA | KIA | KIA |

We now have an array of size 6, and each element’s value is null

Now that I have each car in an array, I can upgrade and downgrade each car by calling the array element.

cars[1].upgrade();

cars[5].downgrade();

cars[0].upgrade().upgrade();

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] | [5] |
| BMW | HONDA | KIA | KIA | KIA | RENAULT |

We now have an array of size 6, and each element’s value is null

***Specification***

Modify RoboRunner to keep track of the number of moves it makes. Then create the appropriate getter method to retrieve this value.

Create “***Robots\Lab11.java***”.

Ask the user for the world name, size, number of RoboRunners to create, and the position of the RoboRunner. Each robot should have 0 beepers and face EAST. Once all RoboRunners have been created, run the completeMaze() method on each RoboRunner. Then, output the number of moves each RoboRunner made.

Use the input to test your program:

bigmaze1

17

16

4

1

1

1

15

10

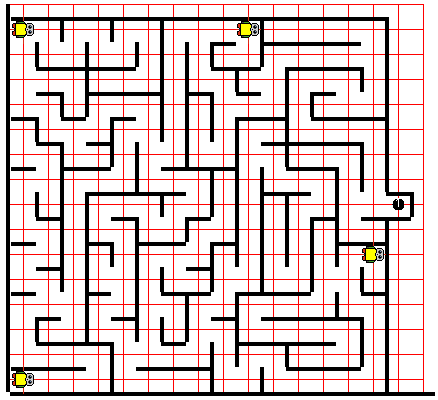
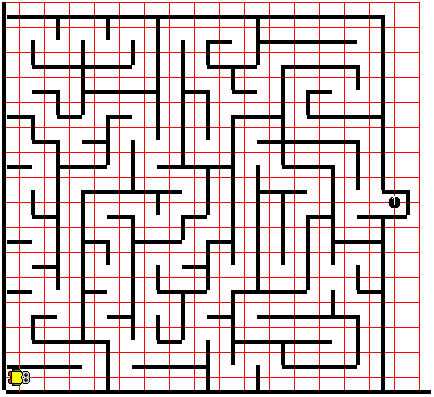
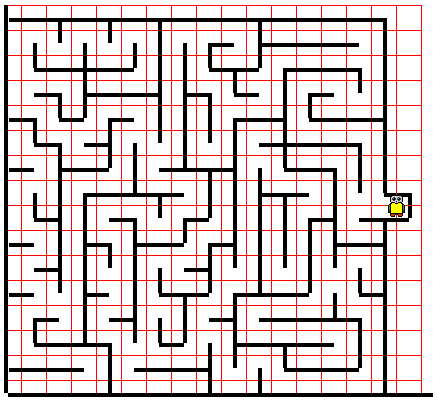
15

15

6

Feel free to try to create robots at different positions.

Start Finish



**Lab12**

**2D Arrays**

***Objective***

The student will learn about creating 2D Arrays and practice using nested loops.

***Background***

You know that 1D arrays, or 1 dimensional arrays, are just a fixed list of elements. However, we can take this to another dimension. 2D arrays are very similar to 1D arrays; these are just arrays of arrays.

The basic syntax to crate a 2D array is as follows:

Type[][] name = new Type[dimension1][dimension2];

Based on this concept, you can now see that it’s possible for us to create arrays of many dimensions; for right now, we’ll stick with 2D.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| [0] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [1] | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 |
| [2] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [3] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [4] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [5] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [6] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [7] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [8] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [9] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

int[][] array = new int[10][10];

As you can see, a 2D array gives us much more versatility and space for storage. This also enables us to store information in a grid-like fashion.

Calling an element from a 2D array is similar to that of a 1D array. The only difference is that you have another parameter, the column.

array[1][5] = 10;

Iterating through a 2D array is also slightly more complex – If you iterated through with only 1 loop, each element of the first array would be another array. To get the individual integer elements, you would need to use a **nested loop**. This is just a loop inside of a loop.

Remember to make the incrementer for the 2 loops different - I used i & j.

for (int i = 0 ; i < array.length ; i++){

for (int j = 0 ; j < array[i].length ; j++){

System.out.print(array[i][j].length);

}

System.out.println();

}

print() and println() are used separately to allow multiple elements to be printed on the same line.

***Specification***

Create “***Robots\Lab12.java***”.

Ask for the world and use size 15x15. Create a Detector with the default constructor. Disable logging. Go to the end of the beeper path. Using your knowledge of 2D arrays, keep track of the beeper path and print out a character representation of the path from start to finish. Use “ O ” for a beeper and “ . ” for no beeper. Use the world “beeperpath1”.

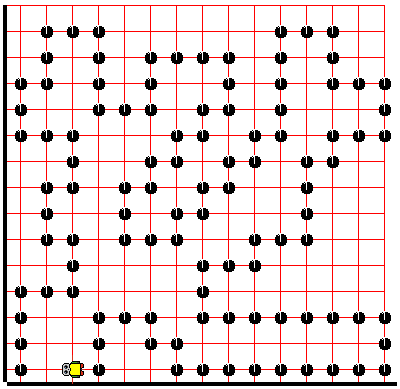
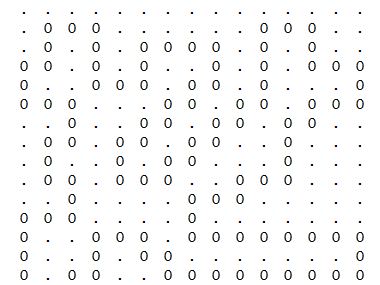
Hint: In order to get the console to print the map in a fashion that looks similar to that of the map, you may have to reverse the nested loop you’re using to output. Instead of starting from 0, start from the maximum value and decrement it to 0.

for(int i = array.length-1; i >= 0 ; i--){

System.out.print(array[i]);

}

Console Path



**Lab13**

**ArrayLists**

***Objective***

The student will learn to use an ArrayList and understand what a Wrapper class is.

***Background***

You know what an array is – a list with a fixed size. In this lab, we’re going to cover *ArrayLists*, another data structure similar to arrays; however, *ArrayLists* use methods to access and mutate elements and are dynamically sized, meaning that they don’t have a fixed size.

When declaring an *ArrayList*, note that an *ArrayList* is a class in java that must be imported via java.util.\* . As such, this has a fairly normal declaration statement:

ArrayList<Integer> arrayList = …

Similar to how we would create any other object, we must use the following syntax:

As you can see, we must specify the type of objects we will be putting into an *ArrayList*. This will be done in angle brackets <>.

ArrayList<Type> name = …

To finish instantiating the ArrayList, use the following syntax:

ArrayList<Integer> arrayList = new ArrayList<Integer>();

We aren’t required to specify the size of the *ArrayList*. That is because it is dynamically sized.

Basic ArrayList creation guide:

ArrayList<Type> name = new ArrayList<type>();

What if we want to get an element from the *ArrayList*? In that case, you would have to call its getter method

arrayList.get(index);

Similar to an array, the index of an *ArrayList* starts at 0.

What about if we wanted to add an element?

There are 2 different ways to add an element to an *ArrayList*: You can add the element to the back of the *ArrayList*, or you can insert it at a specific index. In that case, it will shift the elements currently at and right of the index to the right.

arrayList.add(element);

arrayList.add(index, element);

What if we wanted to remove an element?

The first method will remove the element at that index and shift elements leftward to fill the gap, if there is one.

arrayList.remove(index);

arrayList.remove(element);

The second method will remove the first instance of that element.

What if we wanted to change an element?

This will swap replace the current element at that index with the element you are passing to the method.

arrayList.set(index, element);

Iterating through an *ArrayList* isn’t any more complex than iterating through an array. A key difference, however, is the size() method for an *ArrayList*. As an *ArrayList* is dynamically sized, you must call size() to retrieve the current size of the *ArrayList*.

for(int i = 0; i < arrayList.size() ; i++){

System.out.println(array.get(i));

}

When using ArrayLists, there is a downside to it – we can’t create an ArrayList using primitives. Because of this limitation, Java has **Wrapper Classes**. These are classes which contain a primitive. For instance, an Integer wrapper contains a primitive int, a Double wrapper contains a primitive double, etc. Luckily for us, when we add an integer to the *ArrayList*, Java will automatically convert it to its Wrapper Class equivalent. As such, the only major difference we need to take note of is when we create the ArrayList, we must use objects.

[A more extensive list of ArrayList methods](https://www.tutorialspoint.com/java/util/java_util_arraylist.htm)

***Specification***

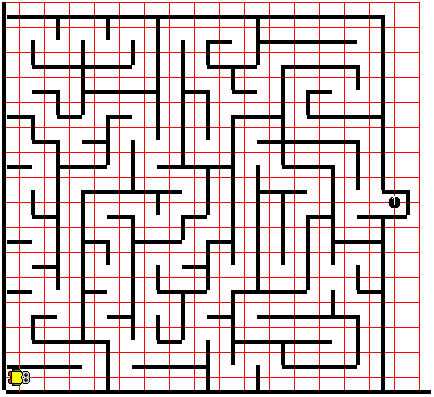
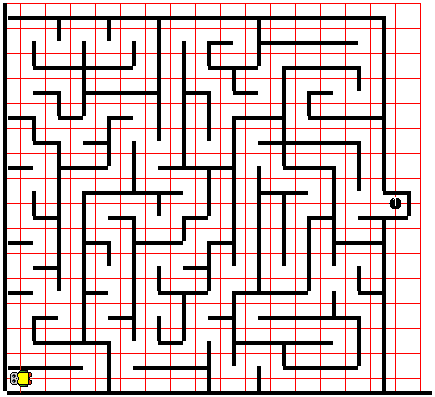
Create “***Robots\BackTracker.java***” extends RoboRunner. BackTracker should override the method completeMaze() to keep track of its moves and turns. It should have its own method backtrack() which causes the robot to turn around and go back the way it came, traveling all the way back to the origin (1, 1).

Create “***Robots\Lab13.java***”.

Ask for the world and use size 17x16. Create a RoboRunner with the default constructor. Go to the end of the beeper path, then backtrack and go back to the origin. Use world “bigmaze1”.

Hint: When I approach a corner and turn right, if I turn around and want to go back the direction I came from, which way do I turn?

Start Finish



**Lab14**

**Error Checking**

***Objective***

The student will learn about try-catches and practice implementing them.

***Background***

Have you ever considered that when you ask the user for input into the program, the user may not always respond with the expected input? In order to account for this and prevent the program from breaking, programmers typically catch the errors.

An example of this would be when the program asks the user to input their favorite even number, and the user types in a word instead. If the program was expecting an integer, and instead received a word, the program would be unable to parse this so it would crash, terminating the program.

By using a try-catch statement, we are able to catch Exceptions because we have created a scenario for when the exceptions occur.

For example:

int in;

boolean validInput = false;

while(!validInput){

try{

in = input.nextInt();

validInput = true;

} catch(InputMismatchException e){

System.out.println(“Invalid input! Try again!”);

Input.nextLine();

}

}

Here, a boolean validInput is used to keep track of whether or not we have a valid input.

If we don’t receive a valid integer, the try-catch statement catches the exception and reenters the while-loop. If a valid input is entered, then validInput is switched to true, and this exits the while loop.

Note: We use input.nextLine() at the end of the catch-statement to ensure that the program doesn’t have an endless loop.

This kind of try-catch loop ensures that the program will only continue if there is a usable input for the program to use.

***Specification***

Open “***Robots\Lab11.java***”.

Modify the inputs so that when the program does not receive a valid integer input, the program will catch the exception and ask the user to try again.

Hint: If you aren’t sure what type of error your program is throwing, try to enter in an invalid input and see what error it throws. Errors typically end with ‘Exception’, such as ‘InputMismatchException’.

Copy your code to new file “***Robots\Lab14.java***”.

**Extra Lab #1**

**Threads & Runnable**

***Objective***

The student will learn about Threads and implement Runnable to multithread.

***Background***

Have you noticed that Java typically only runs 1 command at a time? What this means is that only one task can be executed at a time. However, it is possible to run multiple tasks via additional Threads in Java. What this means is your application will be given multiple platforms to execute code on. Because of this, multiple functions and commands can run at the same time.

A simple way to create a Thread is to pass along an object that implements the Runnable interface. What’s an interface? An interface is a collection of methods that have not yet been defined yet. We won’t go into depth on interfaces.

By implementing the Runnable interface, we are specifying what to do when a Thread tries to run the tasks specified in the class implementing the Runnable interface.

To implement an interface, specify it in the header:

public class Car extends … implements Runnable{

…

public Car(){

…

}

public void run(){

// Code to execute in thread…

}

}

Since we’re implementing the Runnable interface, we also have to override and define its methods. Runnable has 1 method, run(). This method is what specifies to the Thread the tasks that must be executed.

public class test{

…

public static void main(String[] args){

Thread thread = new Thread(new Car());

thread.start();

}

}

We can pass Car into the constructor of Thread because Car implements the Runnable interface.

When we call start() on a Thread object, the thread executes the tasks specified in the object’s run()method. In this case, it executes the code in Car’s run() method.

***Specification***

Create “***Robots\AutoRunner.java***”. Extend RoboRunner and implement Runnable. Give it a 5-arg constructor, receiving an additional parameter, roboNumber, an integer denoting the number of the robot. Save this value into a private variable. Then, override the run() method. In this method, call the method to complete the maze and then output the robot # and the number of moves the robot has taken.

Create “***Robots\Extra1.java***”. Copy over the code from **Lab11**. Modify the lab to create AutoRunners. Instead of having an array of RoboRunners, change it to an array of Threads. When instantiating the Threads, pass a new AutoRunner object as its parameter. Now, remove loops which called methods on the original RoboRunners. Create a single for-loop at the end of the main method to call the start() method on each Thread stored in the array. Now run the program – All 4 AutoRunner objects should run at the same time.

Lab00 (Setting Up) **Setup**: Provide a brief introduction of the labs, showing the students sample code and explaining the functions of every line of code. This should serve as a good crash course (The student does not need to digest all of this information right now, it will slowly be explained in greater detail over time). Most of this information should be review for the student seeing as they have prior experience dealing with constructors and methods.

Lab01 (Robot Basics) **Novice Start**: Give the student an opportunity to play around and practice creating a class, writing the main method, instantiating Robot objects, and then calling methods on them. They will also understand the differences between a no-arg constructor and a 4-arg constructor, and how to call them.

Lab02 (Inheritance & Methods) **Inheritance and Methods**: The student will learn to create classes and understand the core principle behind “Object Oriented Programming”. They will practice creating objects and will see how methods are defined. They will also practice using inheritance by extending the Robot class. They will then learn to invoke the super() constructor in the “*Athlete’s*” constructor and will define methods turnRight(), turnaround(). Finally, using the methods just defined, they will solve the maze with these new methods.

Lab03 (Parameters & For-Loops) **Micromouse**: The student will again extend their “*Athlete”* class to create class “*Runner”*. They’ll also practice using parameters, learning that they can have multiple constructors in a single class. They will then create a method with a 1-arg parameter, taking in an integer which denotes the number of times the Runner needs to move. This will be defined using a for-loop. In addition, another method will be defined with the same name, but instead will have no parameters. This will show that you can have multiple methods and multiple constructors with the same name as long as their parameters are unique.

Lab04(Overriding & While-Loops) **Tower Escape**: The student will extend their “*Runner*” class to create a “*RoboRunner*” class. This class is special – the run() method will be overridden to allow the “*RoboRunner”* to instead continue moving forwards until there is a wall in front of it. The student should understand method overriding and while-loops through this exercise.

Lab05 (Thinking & Application) **Escape the Maze**: Allow the student to develop an algorithm using their prior knowledge of while-loops and conditional statements, as well as the methods given to them by the Robot class to solve the maze. Once it is out of the maze, it should pick up the beeper.

Lab06 (Static Methods) **Shuttle Clean**: The student will create a lineup of “*Runner*s” along the axis X=1. These runners will all move to the right until it touches the wall, picking up all beepers along the way. Then, it will travel back it’s original location, dropping all of its beepers once it reaches its origin. The student should create a static method to streamline this process, calling the method on each runner to pick up the beepers and drop them at their origin.

Labe07 (User input) **User Input**: The student will create a single “*Athlete*”, then takes the user input to receive the number of beepers that the athlete should drop. This will teach them about parsing & using the Scanner class.

Lab08 (Strings & Equality) **Manual Runner**: The student will create a Scanner to allow the user to specify which world they would like to use. After taking this input and creating the world, they should create an “*Athlete*” object and continue to take user input until they type “stop”. With all other inputs, if it corresponds to a certain command, then the athlete should execute that. The student will learn about concatenation, escape sequences, and content equality vs. reference equality.

Lab09 (Getters & Setters) **Tracker** The student will create 10 “*Tracker*” object that keeps track of the number of beepers it has and the position of the tracker object. When the tracker encounters a pile of beepers, the tracker should count the number of beepers on that spot and place them back down. The program should then output the number of beepers and the position on the map that they’re at. Once all trackers have executed, the trackers should all return to their origin, and the program should output the total number of beepers present on the entire screen. This will teach the student about proper programming practices in Java, specifically getting/setting variables through methods instead of accessing the variable directly.

Lab10 (Practical Methods) **Detector**: The student will create a new class “*Detector*” which will have methods that check for beepers in all 4 directions. The student will then follow a path made by the beepers to the end and output the total distance of the path.

Lab11 (Arrays) **Full Control**: The student will practice using arrays by creating *RoboRunner* objects in an array and calling methods on each *RoboRunner* to solve the maze and receive the number of moves it took. This will teach the student how to use arrays and the use cases for anonymous objects.

Lab12 (2D-Arrays & Nested Loops) **Explorer**: The student will practice using 2D arrays by revisiting the beeper-tracer lab. The student will create a map representing the path of the beepers in a 2D array, then use nested loops to output the map into the console.

Lab13 (ArrayLists) **Backtracker**: The student will learn to use ArrayLists and practice backtracking by using the ArrayList to keep track of the robot’s steps. Then, by using the ArrayList as a Stack-like data structure, the student will go through the ArrayList to have the robot backtrack its steps.

Lab14(Error Checking) **Try Again**: The student will learn about error checking to understand that it’s important to realize no user will follow all directions. As such, the student will realize the importance of catching errors before they happen (Exceptions).

Extra1 (Threads & Runnable) **Multitask**: The student will alter their **Lab11** to implement the *Runnable* interface. They should complete the abstract methods and implement the multiple threads to allow all *AutoRunners* to run at the same time.

Good luck! That is the end of this lab series. I understand that not all Java topics are covered here, but this is just meant to serve as an introduction to Java programming. Working on personal projects or completing problems at <https://www.hackerrank.com/> are both great ways to practice and further your knowledge of Java.