Java Methods

Methods

Java classes can implement methods to represent the behavior of the class. These methods can take parameters which are used by the class to perform the behavior. We will discuss the different kinds of parameters and the underlying memory.

1. Create a new Eclipse project called “JavaMethods”.
2. Add the Bicycle class from the Java Classes lab.
   You can do this by adding a new class to this project and copying the code or by making a copy of Bicycle.java from the “JavaClasses” src folder (*from the previous lab*) and add the copy to the “JavaMethods” src folder. This can be done either in your system’s file explorer or in Eclipse.
   After adding the file to the src folder, right-click on the project in Eclipse and choose “Refresh”. It should automatically add the class to the project.

We will continue with our example of a Bicycle class. We present the Biker class, which has fields for the Biker's name and their bicycle.

3. Add the Biker class (Figure 1) to a project with the Bicycle class.

Consider
What behaviors does the Biker class have? How do these behaviors affect the state of the Biker's Bicycle?

Method parameters

Information is passed to methods using parameters. These parameters act as variables inside the method. When creating a method, consider what information the method needs before it can begin performing its task. Java parameters are passed to methods in one of two ways, by reference or by value.

The primitive types are passed by value. The values of these parameters are local to the scope of the method. Any changes made to these parameters are not reflected in the calling scope.

```
public class Biker{
    Bicycle bike;
    String name;

    public Biker(String name, Bicycle bike){
        this.name = name;
        this.bike = bike;
    }

    public void shiftUp(){
        bike.gear = bike.gear + 1;
    }

    public void shiftDown(){
        if (bike.gear != 0)
            bike.gear = bike.gear - 1;
    }

    public void slowDown(int amount){
        bike.applyBreak(amount);
    }

    public void speedUp(int amount){
        bike.speedUp(amount);
    }
}
```

Figure 1 Biker class

Objects are passed by reference. When a method returns, the passed-in reference will still reference the same object as before. If the fields of this object were changed, these changes would be reflected in the calling scope. If the parameter is reassigned an object in the method, that object will be lost when the method returns. We try to illustrate this in the following code snippet with a metaphor.

4. Add this class (Figure 2) to the Eclipse project.

```
class LousyBikeMechanic{
    void repairBike(Bicycle brokenBike){
        brokenBike.gear = -100;
        brokenBike = new Bicycle(100, 100, 100);
    }
}
```

Figure 2 LousyBikeMechanic class

Metaphor:
A LousyBikeMechanic is asked to repair a brokenBike. Being lousy, the mechanic damages the bike more (`brokenBike.gear = -100`). To hide their mistake, the mechanic produces a

new Bicycle to give back to the customer. The customer remembers their bike, however, and the damage persists while the new Bicycle is lost.

Arrays as parameters

Arrays are objects in Java, so, when passed as parameters, arrays are passed by reference.

Arrays can be passed as parameters by adding square brackets to the end of a type as shown in Figure 3.

```
void varArgs(int a, String[] b){}
```

Figure 3 String array parameter

The three dots, or ellipses (…), can be written as the last parameter in the parameter list. This is special syntax in Java for passing a variable number of parameters of a type including none. When a method with varargs is called, it can be passed either an array or a sequence of arguments. Either way, the method will treat the parameter as an array.

```
void varArgs(int a, String... b){}
```

Figure 4 Valid syntax for varargs

Note: There are some rules for using varargs. First, the varargs parameter must be the last parameter in the parameter list. Second, there can only be one variable argument in the method. These two examples would cause a compile time error.

```
void twoVarArgs(int... a, String... b){}
void varArgsFirst(int... a, String b){}
```

Figure 5 Invalid syntax for varargs

Graded Submission Task 1
Write a `printBikes()` method in the LousyBikeMechanic class which takes a variable amount (varargs) of Bicycle objects as a parameter. It should print each Bicycle object's gear and each object's index in the array. If no Bicycles are passed as arguments, print the message "Nothing to check".

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 1. You will submit this copy at the end of this lab.

Stack and heap memory

Important Note: This portion of the lab will use the == operator between two objects. This is not the proper way to determine equality of objects. Applying == to two objects compares their memory addresses. This will be useful to explore how Java handles memory in the background, but again it is not appropriate for testing equality.

Graded Submission Task 2

Make the following changes to the LousyBikeMechanic class:

- Add a field for a Bicycle object
- Add a constructor which takes a Bicycle object as a parameter and assigns it to the field
- In the `repairBike()` method, use == to compare the memory addresses of the Bicycle field and brokenBike and print whether they are equal or not
- In the `repairBike()` method, use == to compare the memory addresses of the Bicycle field and the method brokenBike after the `new` operator is used and print whether they are equal or not.

In the driver, instantiate a Bicycle object. Call the LousyBikeMechanic method `repairBike()` and pass Bicycle object you created the argument using the following code:

```
new LousyBikeMechanic(bike).repairBike(bike);
```

Your output should tell you that at the beginning brokenBike and the field have the same memory address. Then, after `new`, the two no longer refer to the same object.

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 2. You will submit this copy at the end of this lab.

Let's go through exactly what's going on. Starting from the main method:

1. On execution, the JVM allocates memory for the main class, the main method and Bicycle variable within.
2. A Bicycle object is instantiated. The Bicycle class is loaded to the stack. Memory for the object is allocated in the heap, and the bike variable in the stack references the heap location.
3. LousyBikeMechanic object is created in heap, referencing the heap location of bike for the field. LousyBikeMechanic class is loaded to the stack.
4. `repairBike()` is called with a reference to the same bike
5. `new` operator creates a new Bicycle object in the heap, and the brokenBike variable now refers to it.
6. On return, the new Bicycle object in the heap is marked for garbage collection.

The Drone Application

We continue with developing the drone simulation. If you would like to keep these iterations separate, create a new Java project in Eclipse for this lab task. Otherwise, you can work in the original "DroneSimulation" project.

Drone controller (think air traffic control)

A Drone controller is the central unit of the simulation. They are built at a position (x,y) in 2D-space and are immobile. Drone controllers have knowledge of all existing entities which are currently deployed including drones and depots. They can communicate with the drones, but only within an effective signal range. For now, assume transmission time is instantaneous i.e., there is no delay between the controller sending a command and the target receiving it. Communication with depots is assumed to be instantaneous and can occur at any distance.

The drone controller must be able to:

- Hold a collection of drones and drone depots. (Assume the maximum number of drones is static, and defined when the drone controller is instantiated)
- Hold a collection of drones within signal range.
- Remove a drone from the collection of drones within signal range if that drone moves out of range.
- Request a drone be deployed, up to the allowed number.

Distance

Distance for the purposes of communication is computed using the following formula:

$$\text{Distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Graded Submission Task 3 specifics

Create a DroneController class with:

Fields

- Array of Drones, all drones in system
- Array of Drones, all drones in range
- Maximum number of drones
- Current number of active drones
- x and y position
- Effective signal range

Constructor

- Takes maximum number of drones, x,y position, and effective signal range as arguments

Methods

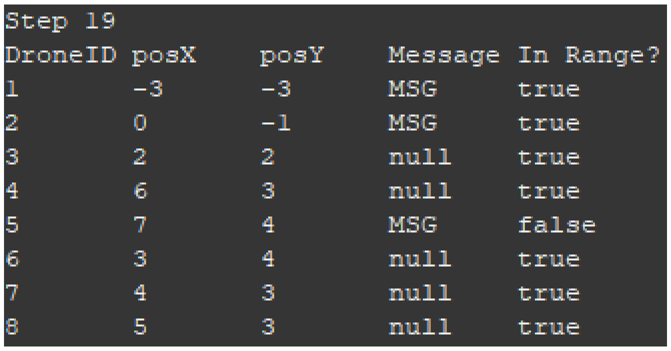
- **`signalDrone(String, int)`**
  Send a message to a drone with a certain ID in range.

- **`signalDeploy(DroneDepot)`**
  Signal DroneDepot to deploy a Drone and add it to system, only if the current maximum has not been reached.
- **`scanRange()`**
  Determine which drones are in range.

Use the provided Driver class to test your implementation. You will need to include your previous implementation of Drone and DroneDepot classes (from Lab 2) to run the driver.

The driver runs a simple simulation for 20 time steps:

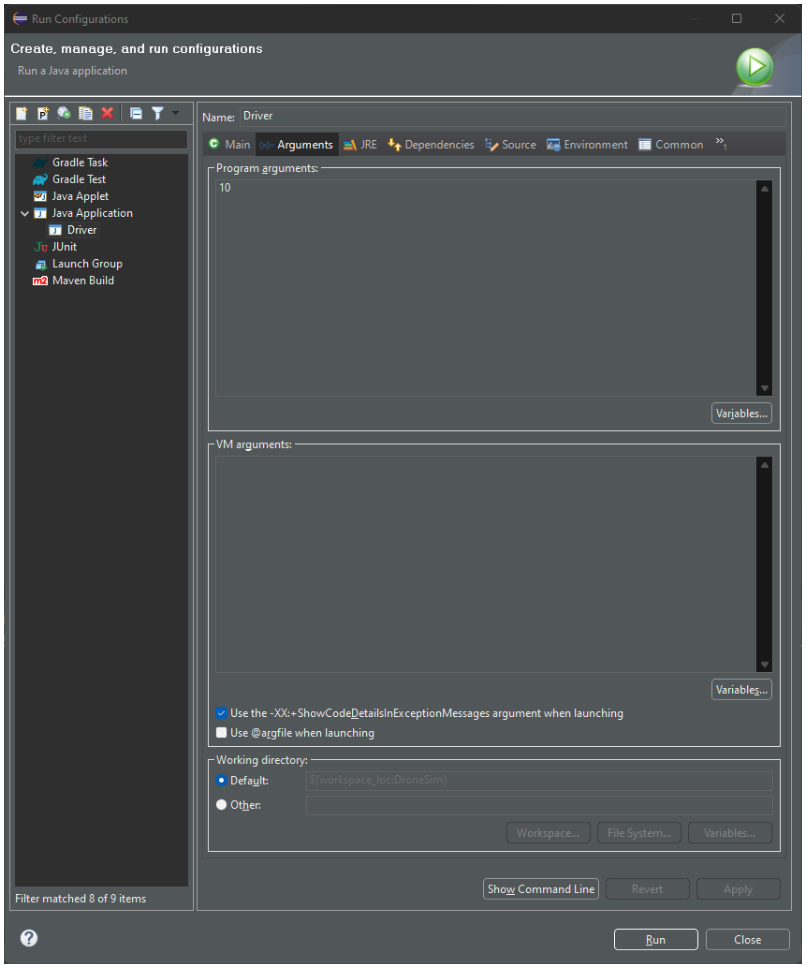
1. Creates a DroneController object with a maximum number of drones at position (0,0) with a signal range of 10
2. Creates a DroneDepot object at position (5,5)
3. Each Drone in the system is ordered to move randomly along the X or Y axis
4. The DroneController scans for drones which are still within signal range
5. The DroneController randomly signals the DroneDepot to deploy another drone OR signals a random Drone in range with the message "MSG"
6. The state of the system is printed to the console i.e.. Each drone's ID, (x,y) position, message, and whether it is in range of the DroneController



```
Step 19
DroneID posX    posY    Message In Range?
1       -3      -3      MSG     true
2       0       -1      MSG     true
3       2       2       null    true
4       6       3       null    true
5       7       4       MSG     false
6       3       4       null    true
7       4       3       null    true
8       5       3       null    true
```

Sample final timestep

This program takes input as an argument passed to the main method. You can set the value that is given at the beginning of the program by navigating to Run -> Run Configurations and entering a number in the program arguments box in the Arguments tab.



Name: Driver

Program arguments:

```
10
```

VM arguments:

[✓] Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching

[ ] Use @argfile when launching

Working directory:

(●) Default: ${workspace_loc:DroneSim}

( ) Other:

Submit your code, including the provided driver, as a zipped Eclipse project

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 3. You will submit this copy at the end of this lab.

Lab wrap-up

In this lab we discussed methods in Java. Methods represent the behaviors of a Java class. Methods can be passed data, as arguments, to use in the execution of the method. Arguments passed to methods can be passed by reference or by value, the choice of which is made automatically. The primitive data types are passed by value. Objects are passed by reference. The values of primitive data types passed to methods are local to the scope of the method. Any changes made to the value are not reflected when the method returns. The reference to an object passed to methods will always refer to the original object. The fields of the object can be modified, and these changes will reflect when the method returns. If the parameter bound to the reference is used to create an object with the `new` operator, the newly created object will be lost when the method returns.

Arrays in Java are objects and are passed by reference. When the last parameter in the parameter list has an ellipses (…), zero or more arguments can be passed for that parameter. These arguments are passed as an array. The varargs argument must be the last parameter in the parameter list, and there cannot be more than one varargs argument.

We also discussed what happens in stack and heap memory. Objects are created in the heap. When a method that has an object as an argument is called, the class of the object is loaded in the stack and the location of the object in the heap is referenced by the variable in the stack. If the `new` operator is called on an argument, the newly created object is created in the heap, but marked for garbage collection when the method returns (if its reference is no longer used).

If you are finished with all graded tasks, you can zip them together into a zip file and submit them at this time. You can find more information about these submissions below.

Submission notes for graded tasks:

- Submit a zipped folder containing the Eclipse projects for each graded task.
- For each project, ensure you have included the src folder containing the .java file, the .classpath file, and the .project file.
- Ensure that each project is named appropriately to correspond with each task.
- Zip the projects together and name the zipped folder “Lab3_firstName_lastName”.
- Submit the zipped file to the respective Canvas link.

Grading Rubrics		
Task 1 (points allotted) - Compilation (2) - Successfully use varargs (4) - Condition to check amount of arguments (4)	Task 2 (points allotted) - Compilation (2) - Bicycle field (2) - Constructor (2) - Expected output for memory comparisons (4)	Task 3 (points allotted) - Compilation (2) - DroneController with required fields (2) - DroneController with required methods (2) - Expected driver output (4)