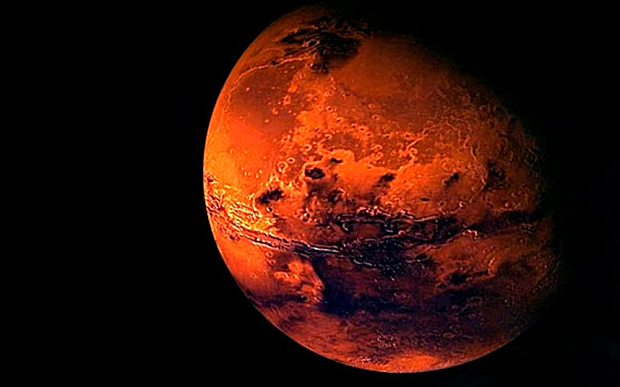
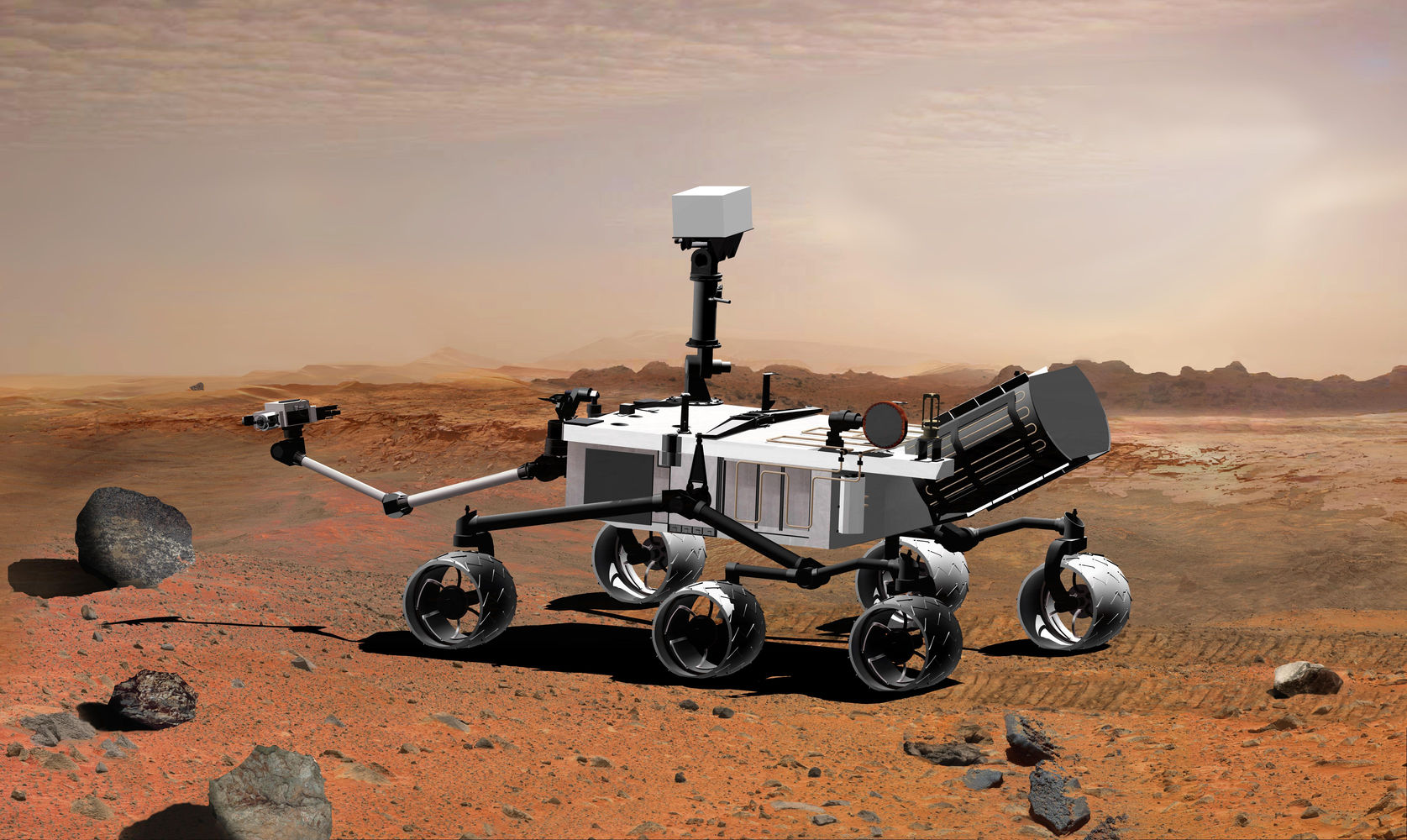
CS 46B Fall 2020

Lab 2





In this lab you will work with subclasses and superclasses, and will get experience with the Eclipse debugger. This week’s assignment is to build and play with a 4-level class hierarchy: DamagedRover, which extends MarsRover, which extends UnmannedVehicle, which extends Vehicle.

Things to note in your lab report will appear highlighted like this.

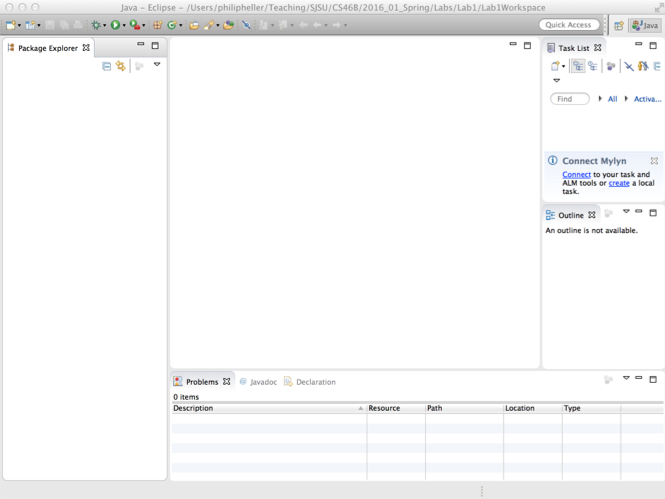
**STEP 1:** Create an Eclipse workspace directory anywhere you want on the driver’s computer. Call it “lab2workspace”.

**STEP 2:** Start Eclipse. If it prompts you to “Select a Workspace”, browse to lab1workspace. Otherwise, it might come up in the last workspace you were in; in this case select File -> Switch Workspace and browse to HW1Workspace.

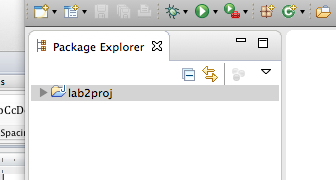
If you see the Welcome screen below, click the arrow circled in red.

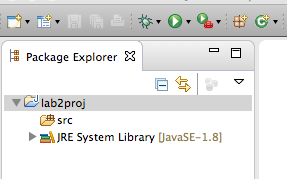


The Workbench screen (see below) is your main work area. The screenshot is from a Mac; appearance on other machines will be different. The workbench contains lots of panels that you don’t need. If you find that they just get in your way, you can get rid of everything except the Package Explorer on the left and the big empty space in the center that will contain source file editors.

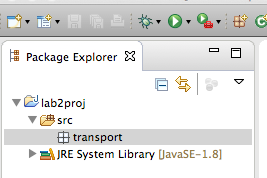


**STEP 3:** Create an Eclipse Java project called lab2proj. In the File menu, select New -> Java Project. A wizard will pop up. Type in the project name, leave all other settings as they are, and select Finish. The Package Explorer will display your new project:



Click the small triangle near the project name to display the project’s contents:

**STEP 4:** Create a package called transport. Right-click on the small icon next to “src” in the Package Explorer, and select New -> Package in the popup menu. In the wizard that appears, “Source folder” should be “lab2proj/src”. Type “transport” into the “Name” field and click Finish. The Package Explorer should now look like this:



**STEP 5:** Create class “Vehicle” in the transport package. Right-click the icon next to “transport” and select New -> class in the popup menu. In the wizard that appears, type in the class name, leave all other settings alone, and click “Finish”. A source code template will appear in the large central editor area of Eclipse.

What source code elements did Eclipse automatically create?

Add the following main method, which obviously won't compile:

public static void main(String[] args) { xxx }

Describe in words how Eclipse indicates that you have a compiler error.

Hover your mouse cursor over the error indication and wait for popup information. What pops up? (Don’t type in the exact words; just briefly explain what the words tell you.)

Delete the main method.

There is no constructor code in Vehicle.java. If you added the following to Vehicle, would you get a compiler error? Why or why not? (Don’t actually type in the code, just think about it).

public static void main(String[] args) { Vehicle v = new Vehicle(); }

Now type in the main method above. Were you right about getting a compiler error? If you were right, enter “I/we were right about the compiler error.”

If you were wrong, enter “I/we were wrong about the compiler error.”

**STEP 6:** Delete your main method and replace it with this:

public static void main(String[] args) {

int i = 10;

System.out.println(i);

i += 100;

System.out.println(i);

i += 1000;

System.out.println(i);

}

To execute this, click on the little Run button at the top of Eclipse. It’s a white triangle in a green circle. Notice where the output appears.

**STEP 7:** For the rest of this lab, and for all other labs, whenever you type anything, save your work. This forces the compiler to run. To save, type CTRL-S in Windows, CMD-S in MacOS.

If Eclipse isn’t displaying source code line numbers, right-click in the margin to the left of the code and select “Show Line Numbers”.

Create a breakpoint at the line “i += 100;”. Right click on the line number for that line, and select “Toggle breakpoint”. Describe what you see, or insert a screenshot into your report.

Execute up to the breakpoint. Instead of clicking the Run button, click the Debug button, which looks like a green bug and is near the Run button. You will probably get a “Confirm Perspective Switch” dialog. Select “Remember my decision” and then click “Yes”. Your Eclipse will rearrange itself. The JVM has executed your code up to the breakpoint that you set. The current line is marked in the main source code window. Hover your mouse cursor over any occurrence of the variable i in that window. What happens?

Click the single-step button: sstep.tiff Again, hover your mouse cursor over any occurrence of the variable i in the source window. Now what is i?

Run the app to completion by clicking this button: rundbg.tiff

Change from the Debug Perspective to the Java Perspective by clicking this button, which is probably in the upper-right corner: javapersp.tiff Delete the main method.

You now have basic Eclipse survival skills. During the semester, it is your responsibility to get familiar with more features of Eclipse.

In your report, write “I understand that it is my responsibility to get familiar with more features of Eclipse.”

**STEP 8:** Create 2 more classes in the transport package. UnmannedVehicle is a subclass of Vehicle. MarsRover is a subclass of UnmannedVehicle. To specify the superclasses, you can either type “extends …” in the source code, or you can configure the “New Java Class” wizard. I find that typing “extends …” is easier. In your report, write “Created 2 more classes”.

Add a private int nWheels to Vehicle. Add a Vehicle constructor that takes one int arg called nWheels. The constructor should store its arg in the instance variable nWheels.

Look at the code for UnmannedVehicle. It now has an error. (If it doesn’t, remember to save the files you’ve edited. This forces the compiler to run.) Hover the mouse cursor over the error. Write “UnmannedVehicle error = “ and the error message.

If you comment out the Vehicle constructor, what happens to the error message in UnmannedVehicle? Do you understand what causes the error? Write a simple explanation.

**STEP 9:** With the Vehicle constructor in place (and not commented out), add a no-args constructor to UnmannedVehicle that explicitly calls the Vehicle constructor, with an argument of 4 to specify 4 wheels. Do you have any compiler errors now? Write “Explicit UnmannedVehicle constructor => compiler errors” or “Explicit UnmannedVehicle constructor => no compiler errors”. Also explain why you saw what you saw.

Add a no-args constructor to MarsRover that prints out “MarsRover ctor”. (“Ctor” is an abbreviation of “Constructor”). Add a line to the UnmannedVehicle ctor that prints out “UnmannedVehicle ctor”. Can that line be the first line of the UnmannedVehicle ctor? If yes, write “UnmannedVehicle ctor: 1st line ok”; if no, write “UnmannedVehicle ctor: 1st line not ok”. Add a line to the Vehicle ctor that prints out “Vehicle ctor”.

Now every ctor prints out an identifying line. Before you execute think about this: if a MarsRover is constructed, in what order will the lines be printed out? Write “The order will be:” and the three lines in expected order.

**STEP 10:** Add a main method to MarsRover with just this line:

MarsRover mr = new MarsRover();

Execute this main. Paste the output into your report. If it’s not what you expected, figure out why.

Before moving on, pause and reflect on what you have done in Steps 7-9. In class, you learned the theory of how Java constructs objects. That’s an invisible process. you made it more visible by using println statements in constructors. That’s a powerful tool. Whenever your code does something that you don’t understand, it’s important to get understanding before you change the code.

**STEP 11:** Congratulations! You got a job with NASA, writing Java software to simulate a rover in the hostile environment of the Martian surface.

On your first day at work, you learn that a huge Martian dust storm has blown the rover to the top of a mountain and damaged its motor and steering mechanism. The rover is safe for now, but if it travels more than 1000 meters in any direction it will fall of a cliff and be destroyed. Google “cost of a Mars rover.” Is a Mars Rover expensive? Should you try to save it? The damaged rover can only travel forward or backward, in little steps of exactly 1, 2, 3, or 4 meters. The distance and direction (forward or back) of each step is random. The rover’s battery has enough energy for 10 km (10,000 meters) of travel.

It seems like it’s just a matter of time before poor little Mars Rover falls off a cliff. NASA wants to know how far the rover will travel back and forth before that happens. You can't know this precisely, because the rover is controlled by random processes, but you can simulate those processes, run the simulation lots of times, and compute the average travel distance.

In package “transport” create a class called “DamagedRover” which extends MarsRover. At the beginning of the class, define 2 constants:

private final static int MAX\_TRAVEL\_METERS\_BEFORE\_EMPTY\_BATTERY = ????;

private final static int METERS\_FROM\_START\_TO\_CLIFF = ????;

Set the values of these ints. Also add the following line:

private final static int N\_SIMULATIONS = 500;

Follow the Java convention for constants by putting these lines right after the open-curly that starts the class definition. Then paste in these variable definitions:

private double position; // Distance from start. Range is

// -1000 to +1000. If Rover travels

// beyond this range, it falls off

// a cliff.

private double metersTraveled; // Total meters traveled

// back and forth.

private boolean fell; // If true, an expensive loss.

Pasting might mess up the alignment of the comments. Type tabs and spaces to make the comments look clean as shown above. It’s harder to get a job if you write messy code, even if the code works.

Paste and complete the following method. Read it and understand it.

//

// Simulates travel under damage conditions. In each turn, travels forward or

// backward either 1, 2, 3, or 4 meters. Continues until there's no more power

// in the battery, or we fall off a cliff. Cliffs are at position = 1000 or

// position = -1000.

//

public void simulateStormDamageTravel()

{

// Reset instance variables here

while (metersTraveled < MAX\_TRAVEL\_METERS\_BEFORE\_EMPTY\_BATTERY)

{

// Random int: 1, 2, 3, or 4. Represents the

// travel distance (maybe forward, maybe back) this turn

double distanceNextTurn = (int)(1 + 4\*Math.random());

// Random boolean for direction of travel this turn.

boolean forwardNotBack = (Math.random() > 0.5);

// Adjust position and metersTraveled.

if (forwardNotBack)

position = ?????

else

position = ?????

metersTraveled = ?????

// Check for falling off cliff. If Rover fell, set fell to true and

// terminate (break out of) the loop.

if (??????)

{

fell = true;

TERMINATE THE LOOP

}

}

}

Add accessor (getter) methods public double getMetersTraveled() and public boolean getFell().

Add a main() method that constructs an instance of DamagedRover, and calls simulateStormDamageTravel(). If the rover fell off the cliff, print “Fell” and the number of meters the rover traveled before falling. If the rover ran out of power without falling, print “No more power”.

Run your main() method. Paste the output. Run it a few more times. Surprisingly, Rover doesn’t fall off the cliff. (If it does, check your code for a bug).

**STEP 12:** Now rewrite the main() method so that it runs the simulation 500 times. Remember that you defined a constant N\_SIMULATIONS, so use that constant rather that the literal number “500”. Count the number of times the rover falls off the cliff, and print it out. Paste the output. Is Rover likely to fall off a cliff before it runs out of power? Are you surprised by this result?