

Laboratory 1: Eclipse and Karel the Robot

Your first laboratory task is to use the Eclipse IDE framework (“integrated development environment”, and the “d” also stands for “design” or “debugging”) to write and run Karel programs.

Log in. Begin by logging in to an ETC laboratory computer. This involves entering your account name and password. If you don’t know these, some combination of the Math 121 staff and the CUS (Computer User Services) staff will help you to figure them out.

Connect to your home server. Use the mouse to hover over the various items in the dock (which should be centered at the bottom of the screen) until you find “Connect to Home Server.” Click on this, and a window should come up offering you only one volume to mount, with your account name. Click on “OK”.

Create a folder for your work in this course. A window should appear, showing your account directory. Click on this window to activate it, select “File” from the menu bar, and then “New Folder” from the “File” menu to create a new folder named “untitled folder. It will appear at the top of your directory window. Rename the folder `math121`. Your home directory now contains a folder for your work in this course. You may want to move it somewhere deeper in your home directory later.

Download the starter project. Although it is not that hard to create new Eclipse projects from scratch, this course will provide you with starter projects, at least for the first assignments. That way, you can focus on the problem-solving aspects of the work. To download the starter project for today’s Karel assignment, find the Safari icon in the dock and click to open it. Use the Safari address bar to enter the URL

<http://www.reed.edu/~jerry/121/hw.html>

and when this page comes up, click on “Programming assignment #1”. Download the assignment even though the downloads manager is worried that it contains an application. Safari should download the starter folder, which is a ZIP archive, and then unzip it as well, assuming that the machine that you are using has the appropriate software. The unzipped content of the ZIP file is a directory named **Assignment1** that contains the project. You should be able to see this on the desktop. Just to be tidy, quit Safari since we’re done with it. This should reveal the window with your home directory again.

Put the downloaded starter project into your folder. Drag the **Assignment1** folder from the desktop into your home **math121** folder to place a copy of it there. *The desktop copy of **Assignment1** will remain, and it is the copy of **Assignment1** that you will modify during this work session. So at the end of the session, be certain to put it in your home **Math121** folder again. Putting it there now was only a practice run. The desktop version*

of **Assignment1** will disappear from the ETC laboratory computer soon after you log out. Beware that in general, work that you leave on the laboratory computer is most likely to be lost.

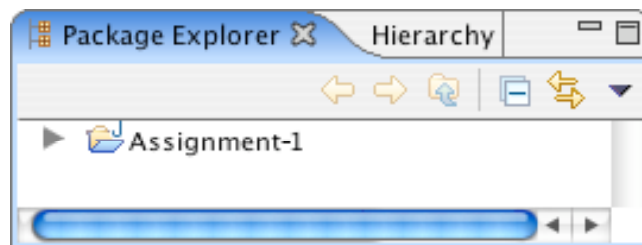
Open Eclipse. Find the Applications folder in the dock (not the Mac OS 9 Applications folder). Open it. Find the Eclipse folder inside the Applications folder, e.g., by starting to type `eclipse` on the keyboard. Open the folder and then open `Eclipse.app`. You will be prompted to select a workspace, with Eclipse making a suggestion. Select “OK” and wait a short while for the Eclipse welcome screen. Click on the “X” in the tab to get to the Eclipse Java perspective.

Eclipse is a popular industrial-strength Java environment with many features. It is also open source, meaning that you can (later, not as part of this laboratory) download your own version from

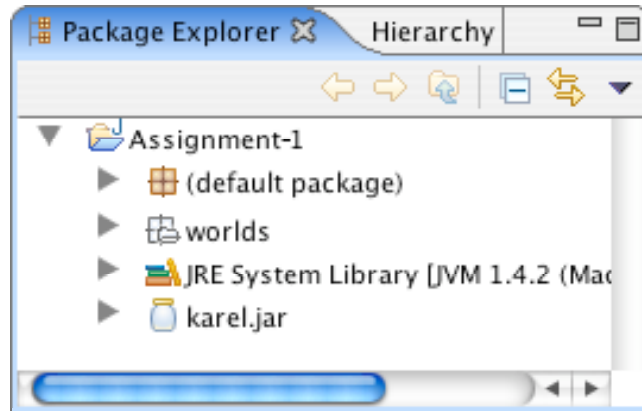
<http://www.eclipse.org/>

and then customize it to work as you wish. Eclipse has been downloaded to the ETC laboratory computers already. This course will require you to know only a minimal portion of Eclipse’s features, but you are encouraged to learn more and then share what you learn with your classmates and instructor on the course mailing list.

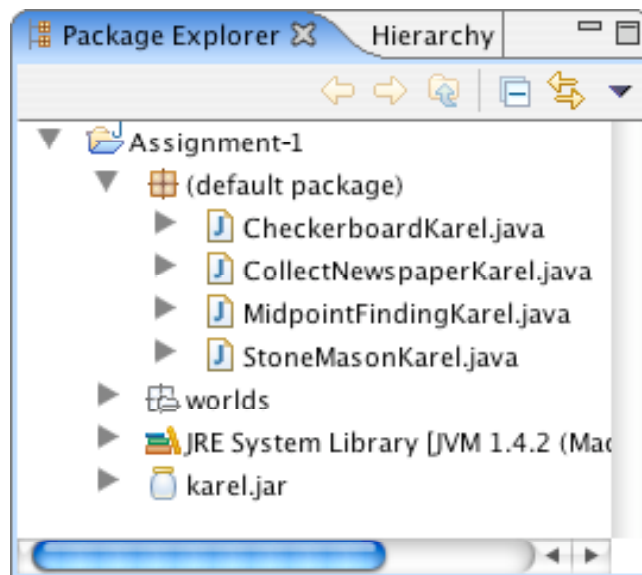
Import the Java project for today’s assignment. Click on “File” in the Eclipse menu bar, and then click on “Import...” in the File menu. When the “Select” window comes up, open “General” (click the little triangle to its left, or double click the word), then select “Existing Projects into Workspace”, and then click the “Next>” button at the bottom of the window. When the “Import Projects” window comes up, select “Browse...” to select the root directory, navigate to the copy of **Assignment1** on the desktop (*be sure to use the desktop copy, not the copy in your home directory*), and click on the “Choose” button. Now the “Import Projects” window is showing that the desired folder is to be placed into a project called **Assignment1**. Click on the “Finish” window. The Eclipse workspace should now show the **Assignment1** project in its Package Explorer window, something like this (but without the dash):



The small triangle to the left of the folder name indicates that you can open it to reveal its contents. When you click on the triangle, it exposes the first level of the package, something like this but with a “demos” package not shown here:



Things look a little more promising. At least there is something about Karel. But things get still more interesting when you open the default package, again by clicking on the triangle:



There—right on the screen—are the Java files for each of the problems in Assignment #1. You can open any of these files by double-clicking on its name. Double-click on `CollectNewspaperKarel`, for example, to see the following file appear in the editing area in the upper right section of the Eclipse screen:

```

CollectNewspaperKarel.java ✕
/*
 * File: CollectNewspaperKarel.java
 * -----
 * At present, the CollectNewspaperKarel subclass does nothing.
 * Your job in the assignment is to add the necessary code to
 * instruct Karel to walk to the door of its house, pick up the
 * newspaper (represented by a beeper, of course), and then return
 * to its initial position in the upper left corner of the house.
 */

import stanford.karel.*;

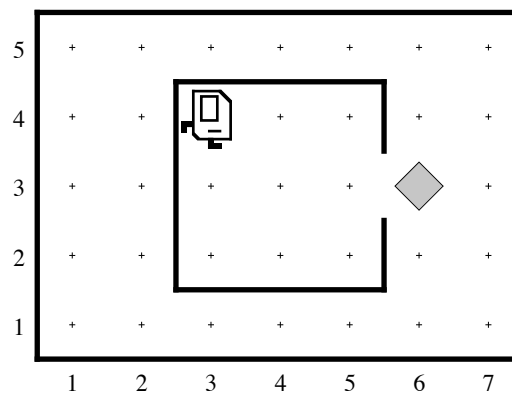
public class CollectNewspaperKarel extends Karel {

    // You fill in this part

}

```

As you might have expected, the file we included in the starter project doesn't contain the finished product but only the header line for the class. The actual program must still be written. If you look at the assignment handout, you'll see that the problem is to get Karel to collect the "newspaper" from outside the door of its "house" as shown in this diagram:



Suppose that you just start typing away and create a **run** method with the following steps:

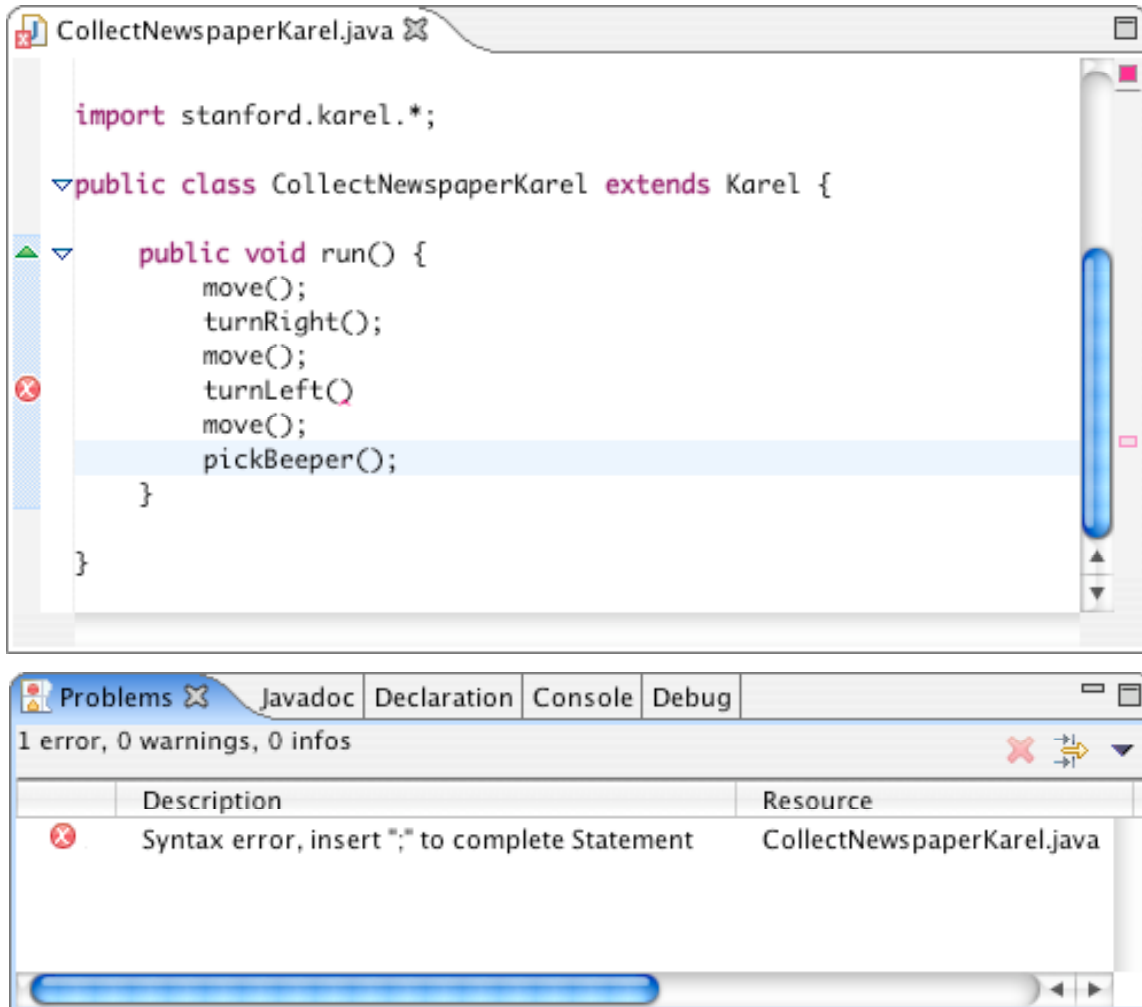
```


public void run() {
    move();
    turnRight();
    move();
    turnLeft();
    move();
    pickBeeper();
}

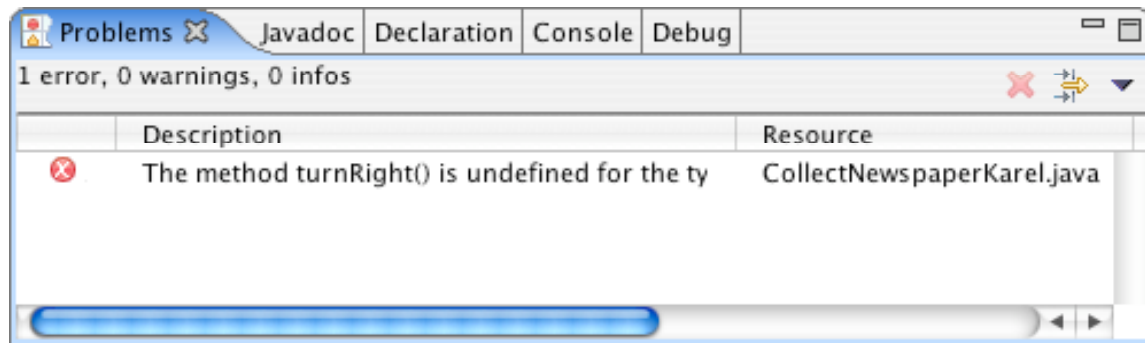
```



The bug symbol off to the side lets you know that this program isn't going to do exactly what you want, but it is still interesting to see what happens. Eclipse compiles your program file every time you save it and then tells you about any errors it found. In this case, saving the file generates the following information in the two right-hand windows:



The **Problems** screen shows the error messages, which are also highlighted with the  symbol in the editor window. Here, the error message is extremely clear: there is a missing semicolon at the end of the indicated line. This type of error is called a **syntax error** because you have done something that violates the syntactic rules of Java. Syntax errors are easy to discover because Eclipse finds them for you. You can then go back, add the missing semicolon, and save the file again. This time, the **Problems** screen shows



Even though part of the error message is cut off, the reason for the problem is clear enough. The Karel class understands `turnLeft` as a command, but not `turnRight`. Here you have two choices to fix the problem. You can either go back and add the code for `turnRight` or change the header so that `CollectNewspaperKarel` extends `SuperKarel` instead. Fixing this problem leads to a successful compilation in which no errors are reported in the **Problems** screen.

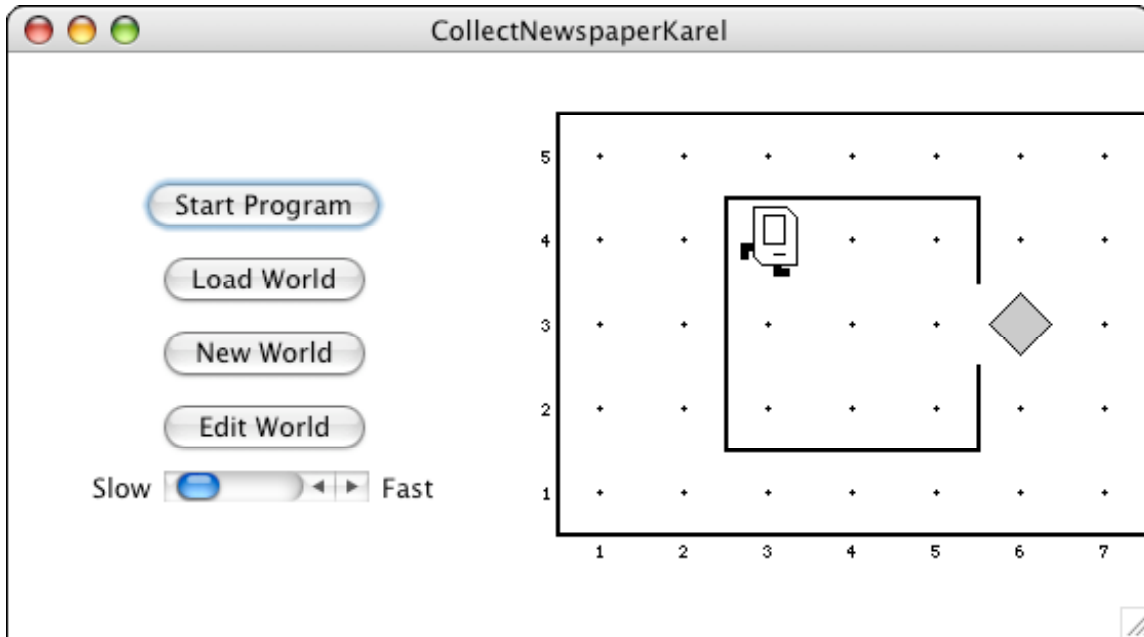
Even though the program is not finished—both because it fails to return Karel to its starting position and because it doesn't decompose the problem to match the solution outline given in the assignment—it may still make sense to run it and make sure that it can at least pick up the newspaper.

Running CollectNewspaperKarel under Eclipse. For obscure reasons, we are going to run our Karel programs as Java applications, but from next week on we will run all of our Java programs as applets. The only operational consequence is that the pending, somewhat complicated, ritual of getting Karel programs to run will be replaced by a simpler procedure starting next week.

Select the “Run” menu from the Eclipse menu bar, and then select “Run...” from the resulting menu. In the “Create, manage, and run configurations” window, select “Java Application” and then the “New” button. Change the name from `New_configuration` to `CollectNewspaperKarel`. Select “Arguments” and then in the “Program arguments” window enter

```
code="CollectNewspaperKarel"
```

Here the double quotation marks are part of what you type. Return from the “Arguments” tab to the “Main” tab and for “Main Class” enter `CollectNewspaperKarel`. Then select “Run”. Eclipse will start the Karel simulator and, after several seconds, display a window that looks like this:



If you then press the **Start Program** button, Karel will go through the steps in the **run** method you supplied.

In this case, however, all is not well. Karel begins to move across and down the window as if trying to exit from the house, but ends up one step short of the beeper. When Karel then executes the **pickBeeper** command at the end of the **run** method, there is no beeper to collect. As a result, Karel stops and displays an error dialog that looks like this:



This is an example of a **logic error**, which is one in which you have correctly followed the syntactic rules of the language but nonetheless have written a program that does not correctly solve the problem. Unlike syntax errors, the compiler offers relatively little help for logic errors. The program you've written is perfectly legal. It just doesn't do the right thing.

Debugging

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

— Maurice Wilkes, 1979

More often than not, the programs that you write will not work exactly as you planned and will instead act in some mysterious way. In all likelihood, the program is doing precisely what you told it to. The problem is that what you told it to do wasn't correct. Programs that fail to give correct results because of some logical failure on the part of the programmer are said to have **bugs**; the process of getting rid of those bugs is called **debugging**.

Debugging is a skill that comes only with practice. Even so, it is never too early to learn the most important rule about debugging:

In trying to find a program bug, it is far more important to understand what your program is doing than to understand what it isn't doing.

Most people who come upon a problem in their code go back to the original problem and try to figure out why their program isn't doing what they wanted. Such an approach can be helpful in some cases, but it is more likely that this kind of thinking will make you blind to the real problem. If you make an unwarranted assumption the first time around, you may make it again, and be left in the position that you can't for the life of you see why your program isn't doing the right thing.

When you reach this point, it often helps to try a different approach. Your program is doing *something*. Forget entirely for the moment what it was supposed to be doing, and figure out exactly what is happening. Figuring out what a wayward program is doing tends to be a relatively easy task, mostly because you have the computer right there in front of you. Eclipse has many tools that help you monitor the execution of your program, which makes it much easier to figure out what is going on. You'll have a chance to learn more about these facilities in the coming weeks.

Creating new worlds. The one other thing you need to know about is how to create new worlds. The three buttons on Karel's control panel

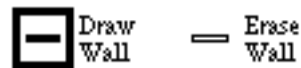


Do pretty much what you'd expect. The **Load World** button brings up a dialog that allows you to select an existing world from the file system, **New World** allows you to create a new world and to specify its size, and **Edit World** gives you a chance to change the configuration of the current world.

When you click on the **Edit World** button, the control panel changes to present a tool menu that looks like this:



This menu of tools gives you everything you need to create a new world. The tools



allow you to create and remove walls. The dark square shows that the **Draw Wall** tool is currently selected. If you go to the map and click on the spaces between corners, walls will be created in those spaces. If you later need to remove those walls, you can click on the **Erase Wall** tool and then go back to the map to eliminate the unwanted walls.

The five beeper tools



allow you to change the configuration of beepers on any of the corners. If you select the appropriate beeper tool and then click on a corner, you change the number of beepers stored there. If you select one of these tools and then click on the beeper-bag icon in the tool area, you can adjust the number of beepers in Karel's bag.

If you need to move Karel to a new starting position, click on Karel and drag it to some new location in the map. You can change Karel's orientation by clicking on one of the four Karel direction icons in the tool area. If you want to put beepers down on the corner where Karel is standing, you have to first move Karel to a different corner, adjust the beeper count, and then move Karel back.

These tools should be sufficient for you to create any world you'd like, up to the maximum world size of 50x50. Enjoy!

Reminder: Remember that you have been modifying only the desktop copy of the **Assignment1** folder. To keep your work, again drag this folder into your home **Math121** folder to place a copy of it there. In general, the work that you save on the ETC laboratory computer is most likely to be lost unless you move it to your home directory at the end of your work session. Move it back from your home directory to the desktop at the beginning of your next session, and so on.