Using Karel with Eclipse

Once you have downloaded a copy of Eclipse as described in Handout #5, your next task is to understand how to write Karel programs using the Eclipse framework. Although it is not all that hard to create new Eclipse projects from scratch, it certainly reduces the complexity of assignments if we provide starter projects to get you going. That way, you can ignore all the mechanical details of making new projects and focus instead on the problem-solving aspects of the assignments.

Downloading starter projects

The first step in working with any Karel assignment is to download the starter project for that assignment. If you go to the CS106A assignment page (go to the CS106A web site and click the **Assignments** link), you'll see a display that looks like something this:

Programming Assignments

| Assignment	Instructions	Starter Code	Sample Program	Distributed
1. Karel	07-assignment-1-karel.pdf	Assignment1.zip		Lecture 2
2. Simple Java	13-assignment-2-simple-java.pdf	Assignment2.zip		Lecture 6
3. Breakout!	19-assignment-3-breakout.pdf	Assignment3.zip	Breakout Demo	Lecture 10
4. Hangman	27-assignment-4-hangman.pdf	Assignment4.zip		Lecture 14
5. Yahtzee!	35-assignment-5-yahtzee.pdf	Assignment5.zip	Yahtzee! Demo Yahtzee! Display Yahtzee! Magic Stub	Lecture 18
6. NameSurfer	39-assignment-6-name-surfer.pdf	Assignment6.zip	NameSurfer Demo	Lecture 22
7. FacePamphlet	42-assignment-7-facepamphlet.pdf	Assignment7.zip	FacePamphlet Demo	Lecture 25

If you click on the link for **Assignment1.zip**, your web browser will download the starter folder. In some cases, the browser will also unzip/extract the folder automatically, assuming that you have the appropriate software for expanding files from a ZIP archive. The notes on the assignment page above tell you what software you need. The unzipped contents of the ZIP file is a directory named **Assignment1** that contains the project. Move that folder to someplace on your file system where you can keep track of it when you want to load the project.

Importing projects into the workspace

From here, your next step is to start up Eclipse, which will bring up the Eclipse window shown on the last page of Handout #5. Find the small icon in the toolbar that looks like:

This button is the **Import Project** button and is used to copy a project folder into the Eclipse workspace so that you can work with it. Click on this button and then use the **Browse** option to find the **Assignment1** folder. When you do so, Eclipse will load the starter project and display its name in the **Package Explorer** window like this:
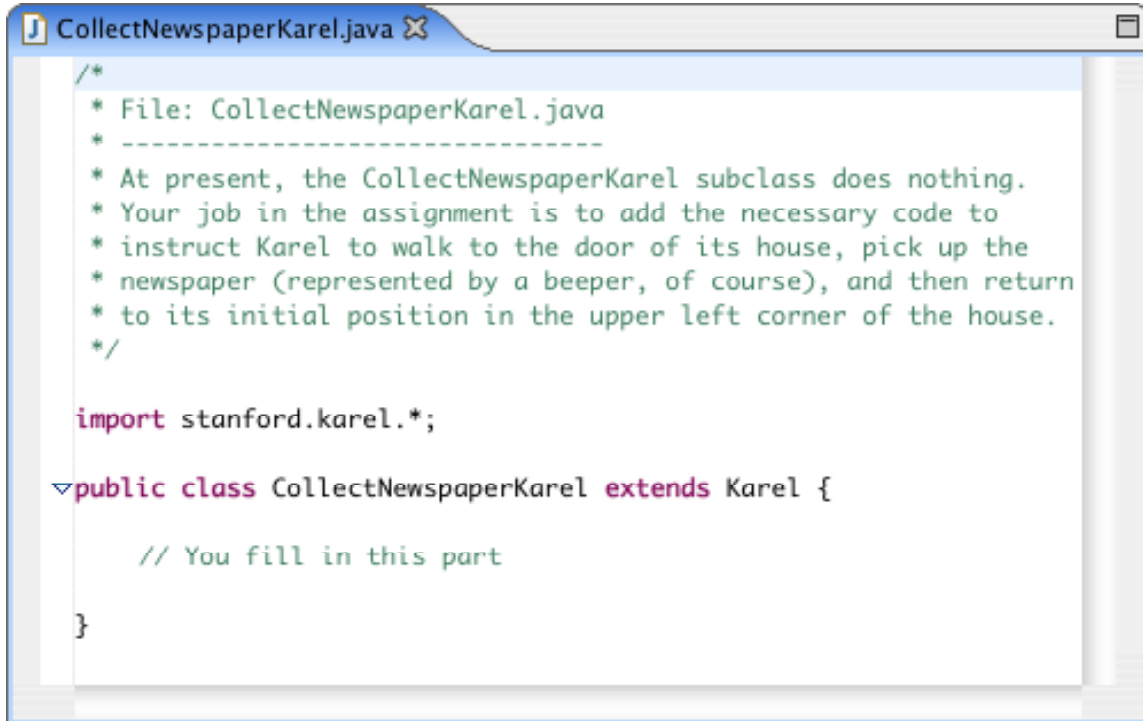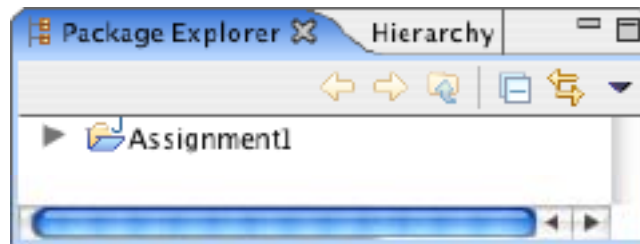

The small triangle to the left of the folder name indicates that you can open it to reveal its contents. When you click on the triangle, it exposes the first level of the package:
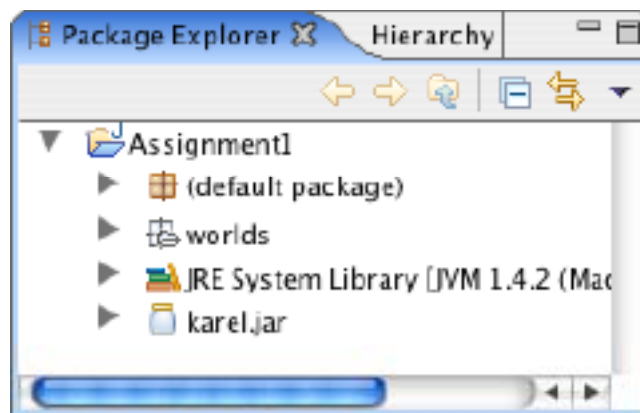
Note that you likely may <u>not</u> see the "JRE System Library" line displayed in the graphic above (or if you do, it may have a different number like 1.5 or 1.6, rather than 1.4.2). Nevertheless, at this point things look a more promising – there is something about Karel there on the last line. But things get more interesting when you open the default package, which is where the code you will write this quarter will go. Opening this package reveals
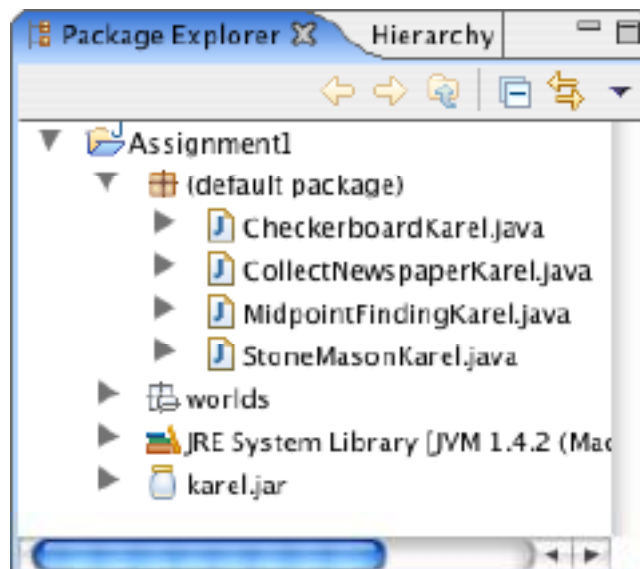
Now things have gotten much more exciting. There—right on the screen—are the Java files for each of the assignments. You can open any of these files by double-clicking on its name. If you double-click on CollectNewspaperKarel, for example, you will see the following file appear in the editing area in the upper right section of the Eclipse screen:

```
/*
 * 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

}
```

Note that the comments at the top of the file many not display initially and may need to be "expanded" by clicking the small '+' sign next to the comment header line.

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:
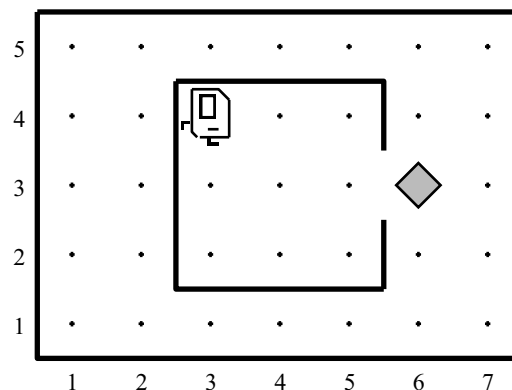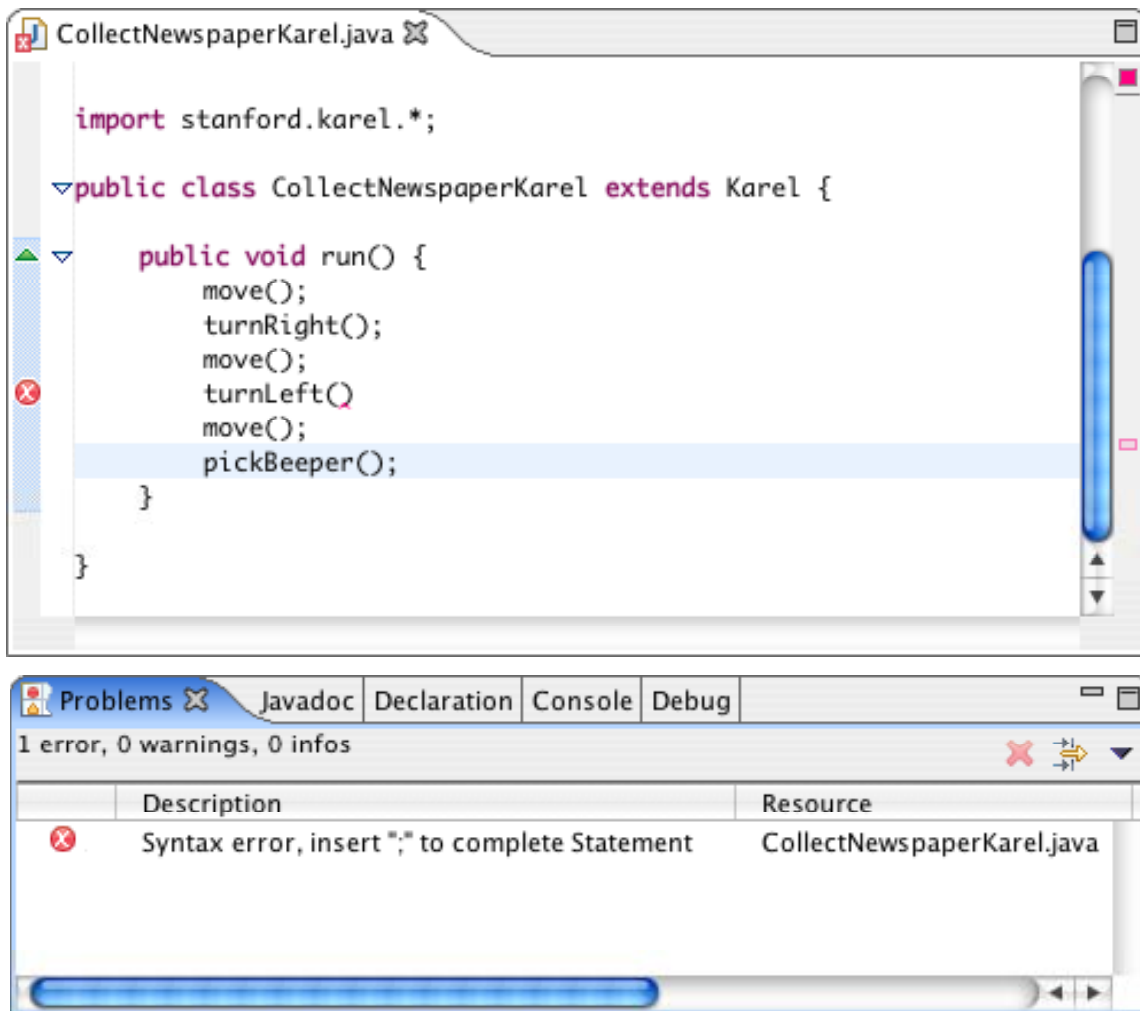

Suppose that you just start typing away and create a `run` method with the steps on the following page:

```
public void run() {
    move();
    turnRight();
    move();
    turnLeft()
    move();
    pickBeeper();
}
```
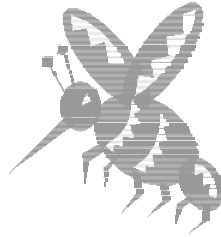
The bug symbol off to the side lets you know that this program isn't going to do exactly what you want, but it is still interesting to see what happens. Eclipse compiles your program file every time you <u>save</u> it and then tells you abut any errors it found. In this case, saving the file generates the following information in the two right-hand windows:

CollectNewspaperKarel.java ✕

```
import stanford.karel.*;

public class CollectNewspaperKarel extends Karel {

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

}
```

Problems ✕ | Javadoc | Declaration | Console | Debug

1 error, 0 warnings, 0 infos

	Description	Resource
✕	Syntax error, insert ";" to complete Statement	CollectNewspaperKarel.java

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:
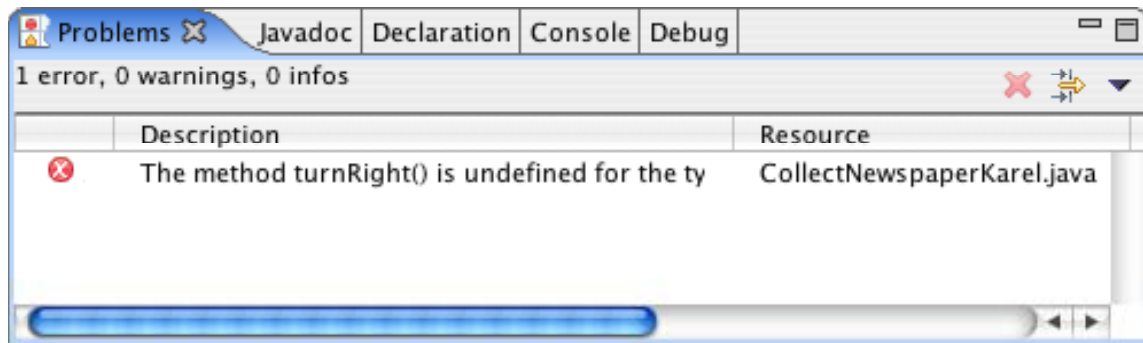
Even though part of the error message is cut off, the reason for the problem is clear enough. The Karel class understands **turnLeft** as a command, but not **turnRight**. Here you have two choices to fix the problem. You can either go back and add the code for **turnRight** or change the header so that **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 a Karel program under Eclipse

Running a program under Eclipse makes use of the two buttons on the tool bar that look like this:

The button on the left causes Eclipse to search the workspace for all runnable programs and ask you which one you want to run. Since all four programs from Assignment 1 are part of the workspace, clicking this button will generate a list containing the names of the four Karel classes. The button on the right is a "faster" version of the run button that skips the search for runnable programs and just runs the same program you ran most recently during this Eclipse session.

If you then select **CollectNewspaperKarel** from the list of programs that appears, Eclipse will start the Karel simulator and, after several seconds, display a window that looks like the picture on the next page:
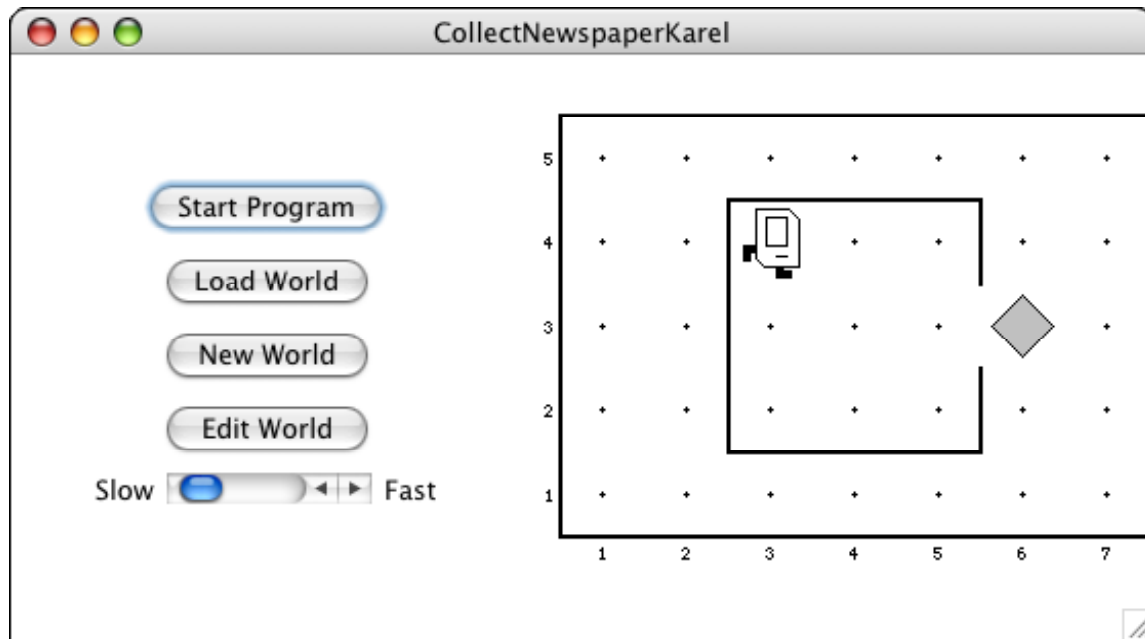
If you then press the **Start Program** button, Karel will go through the steps in the `run` method you supplied.

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 might want to know about is how to create new worlds. The three buttons on Karel's control panel

Load World

New World

Edit World

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 the picture on the next page:

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!