

CS 106A: Programming Methodology

Using Eclipse and Karel

based on a similar handout written by Eric Roberts, Mebram Sahami, and Keith Schwarz; modified by Marty Stepp

Once you have downloaded and installed a copy of Eclipse as described on the course web site, 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 mechanical details of making new projects and focus instead on problem-solving aspects of assignments.

Downloading starter projects:

The first step in working with any Karel assignment is to download the starter project for that assignment. If it's still early in Week 1, you can get a starter project by going to the Lectures web page and downloading [all-lecture-??zip](#).

Once HW1 has gone out, if you go to the Homework section on the CS 106A page, you'll find a link labeled "Project Starter ZIP". If you click on this link, your web browser will download the starter folder.

After you download the starter ZIP, you need to extract the files from the ZIP archive. The unzipped contents of the ZIP file is a directory that contains the project; if it's the HW1 starter, this folder will be named **cs106a-hw1-karel**. Move that folder to some place on your file system where you can keep track of it when you want to load the project.

Importing projects into the workspace:

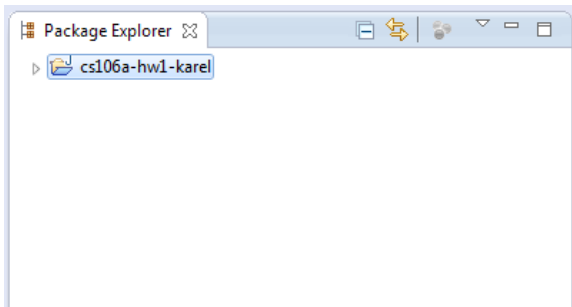
From here, your next step is to start up Eclipse, which will bring up the main Eclipse window. Find the small icon in the toolbar that looks like the icon at right. This button is the **Import Project** button and is used to load a project folder into Eclipse so that you can work with it. Click on this button and then click the **Browse...** button to find and highlight (click on) the **cs106a-hw1-karel** folder, then click **OK**. Now, make sure that the check box labeled "Copy projects into work space" is *not* checked (if the box is checked, just click on it to uncheck it). Then click the **Finish** button. When you do so, Eclipse will load the starter project and display its name in the **Package Explorer** window (on the left-hand side of the Eclipse application) like the picture below at left.



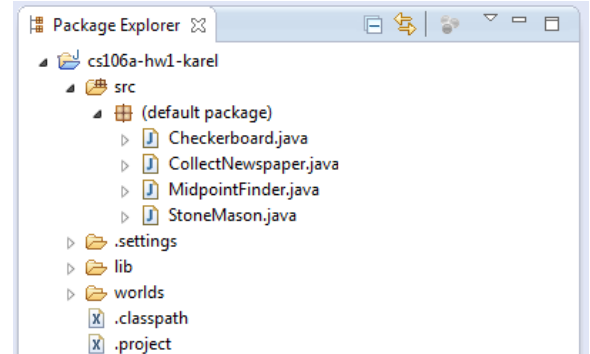
The small triangle-sign icon to the left of the folder name indicates that you can open it to reveal its contents. Clicking on the plus-sign/triangle exposes the first level of the package. Things get more interesting when you open the **src** folder ("src" is short for "source code") and then open **(default package)**. This reveals the Java files you will edit in the assignment, like the picture below at right.

This same Import process works for homework starter code, or for downloading ZIPs of lecture code posted after each class.

Before clicking ► triangle icon:



After:

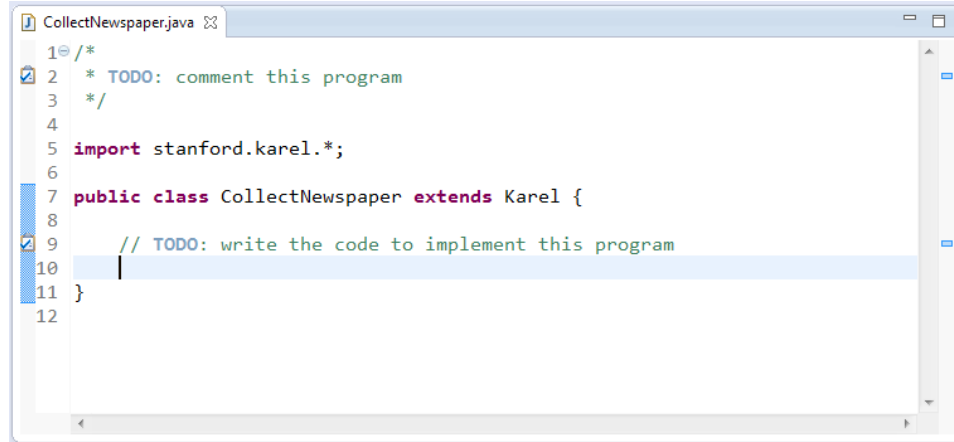


A typical Eclipse project in this course contains **folders** and files such as the following:

- **src/** Source code. The .java files for you to edit are stored here.
- **lib/** Libraries. Supporting code that we provide to you, such as the Karel or Stanford/ACM Java libraries.
- **bin/** This is where the compiled binary version of your Java files are placed. They have the extension **.class**.
- **res/** Resources. Input data files, images, sound files, and any other resources used by your project.
- **worlds/** Karel world files. (Only for Karel the Robot projects, such as Homework 1 and Week 1 lectures.)
- **.classpath**, **.project**, and **.settings/** These are Eclipse's internal information and settings about the project.

Editing Java code files:

You can open any of the Java source code files for the assignment by double-clicking on its name in the left Package Explorer area. If you double-click on **CollectNewspaper.java**, for example, you will see the following file appear in the editing area in the upper middle section of the Eclipse screen:



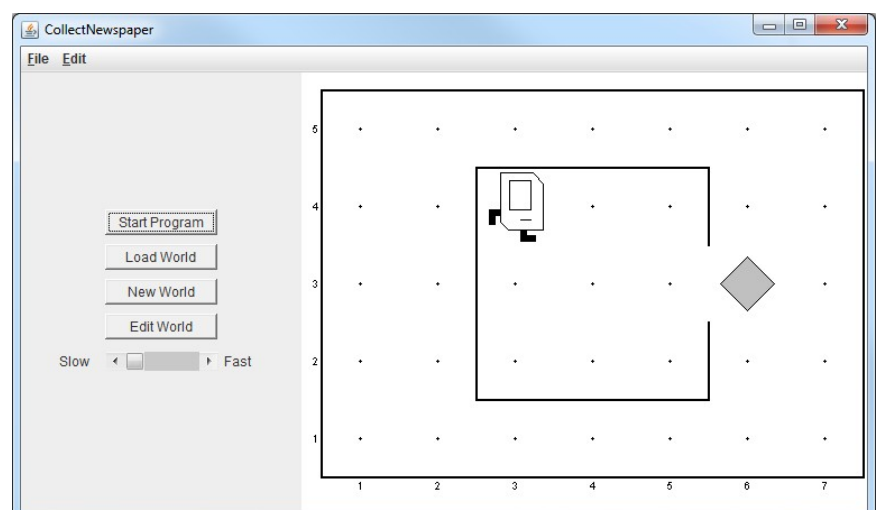
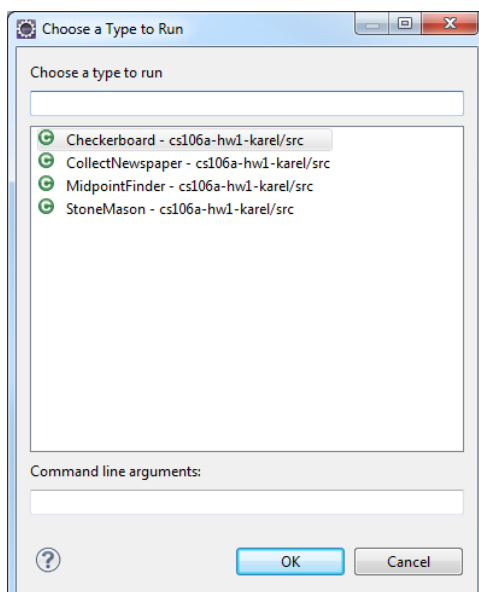
```
1 /*
2  * TODO: comment this program
3  */
4
5 import stanford.karel.*;
6
7 public class CollectNewspaper extends Karel {
8
9     // TODO: write the code to implement this program
10
11 }
12
```

Note that the comments at the top of the file may 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.


Running a Karel program:

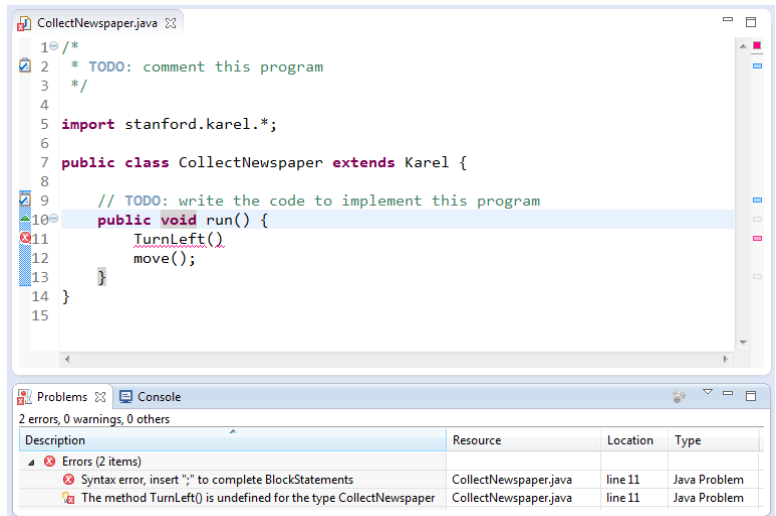
Running a program under Eclipse makes use of the two buttons on the tool bar that look like the icons at right. The button on the *right* causes Eclipse to search the workspace for all runnable programs and ask you which one you want to run, as shown in the figure below at left. Since all four programs from Homework 1 are part of the project, clicking this button will generate a list containing the names of the four Karel classes. The button on the *left* 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 **CollectNewspaper** from the list of programs that appears, Eclipse will start the Karel simulator and display a window that looks like the figure below at right. If you then press the **Start Program** button, Karel will go through the steps in the **run** method you supplied.



Syntax errors:

Eclipse tries to compile your program file from Java code into binary format every time you **save** it. If you have typed any code that violates Java syntax, Eclipse will alert you about any **syntax errors** it found by underlining the illegal code in red. In this case, saving the file generates the following information in the two windows (the first in the upper middle part and the second along the bottom of the Eclipse window):

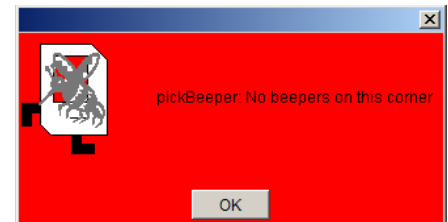
The **Problems** pane shows the error messages, which are also underlined and highlighted with the  symbol in the editor window. Clicking the small '+' sign next to "Errors" lists the errors that Eclipse has detected in your program. In the example above, the first problem is that there is a missing semicolon at the end of line 11. You can then go back, add the missing semicolon, and save the file again. The second error is because the user has misspelled the `turnLeft` command with an uppercase T. Names must match exactly, including capitalization.



Logic errors:

Suppose your program's run method begins with a `pickBeeper();` statement, but there is no beeper at Karel's initial location. You will see a message pop up that looks like the one at right.

This is an example of a **logic error**, where 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 does not offer help for logic errors. The program you've written is legal; it just doesn't do the right thing.



The error message box may contain some information about the **line numbers** on which the error occurred. This can help you track down the faulty part of the program. If several lines are listed, it means that the first method was called by the second, which was called by the third, and so on. (This is sometimes called the program's "call stack").

Debugging:

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 with practice. Even so, it is never too early to learn the most important rule about debugging:

In trying to find a 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, as shown at right, 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 figure below:



This menu of tools gives you everything you need to create a new world. The two wall-building 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!