Programming with Karel
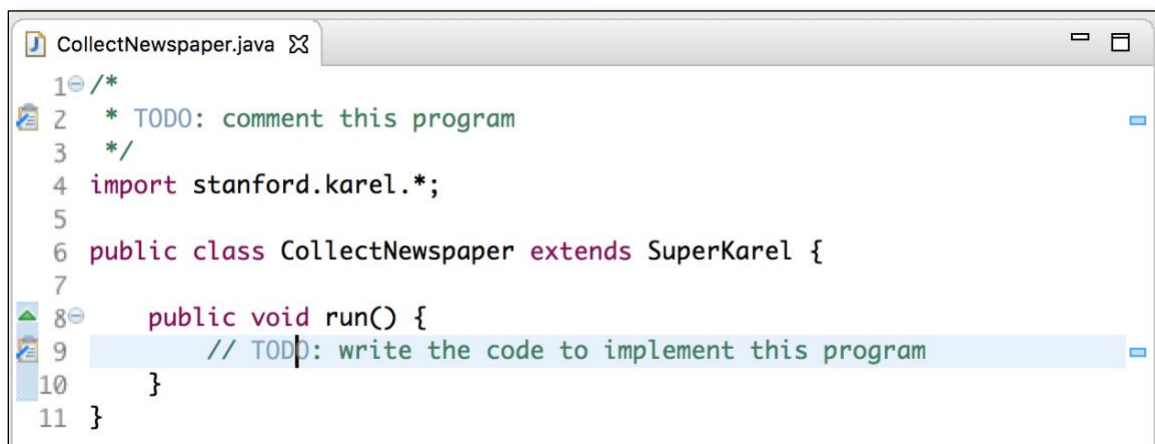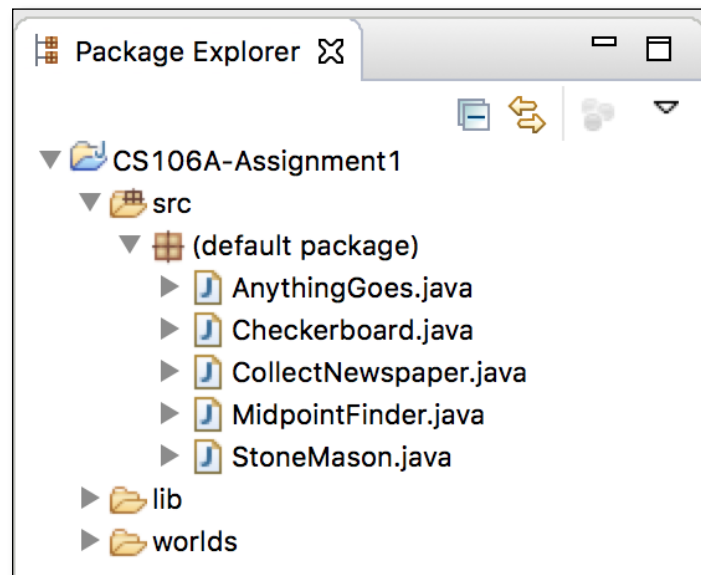
To get started with Karel, first go to the Eclipse page on the course website; follow the installation instructions, and read the instructions to import and run the **Assignment1** Eclipse project. Import the project into Eclipse; once you've done so, your expanded project in the Eclipse sidebar should look like this:

```
Package Explorer

▼ CS106A-Assignment1
    ▼ src
        ▼ (default package)
            ▶ AnythingGoes.java
            ▶ Checkerboard.java
            ▶ CollectNewspaper.java
            ▶ MidpointFinder.java
            ▶ StoneMason.java
    ▶ lib
    ▶ worlds
```

Your next task is to understand how to write Karel programs using Eclipse. Go ahead and double-click on one of the java files in the project, **CollectNewspaperKarel.java**, to make it open in the editing area:

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

As you might have expected, the file we included in the starter project just contains the header line for the class and some TODOs (which you should remove once you're done!). If you look at the assignment handout, you'll see that the first problem is to get Karel to collect the "newspaper" from outside the door of its "house" as shown in this diagram:
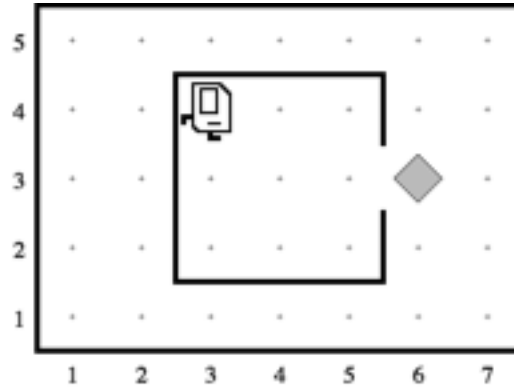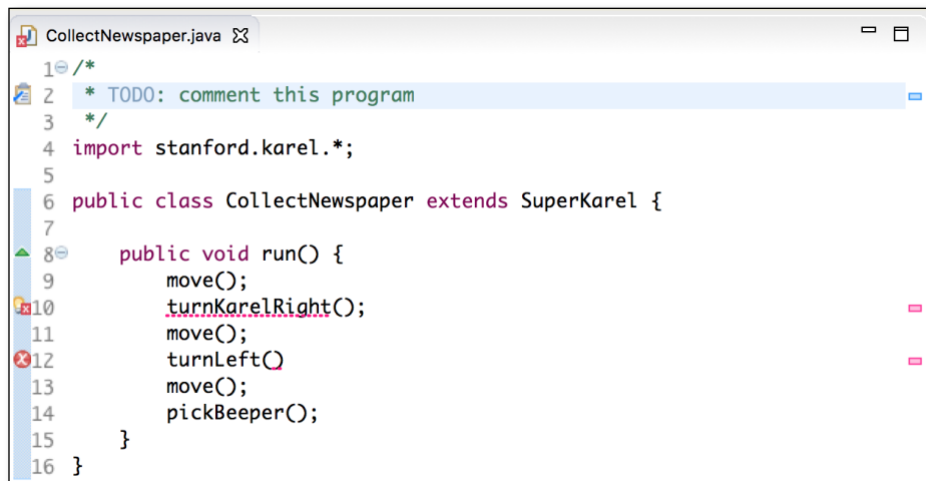

Suppose that you just start typing away and create a **run** method with the steps below:

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

As you type, Eclipse compiles your program automatically and tells you about any errors it finds. Unfortunately, our code above is buggy and the editor view displays two errors:

```
 1  /*
 2   * TODO: comment this program
 3   */
 4  import stanford.karel.*;
 5
 6  public class CollectNewspaper extends SuperKarel {
 7
 8      public void run() {
 9          move();
10          turnKarelRight();
11          move();
12          turnLeft()
13          move();
14          pickBeeper();
15      }
16  }
```

Eclipse adds error symbols in the margin, and also underlines the code that is causing the issue. To view more information about each error, you can simply hover over either the underlined text or the error symbol in the margin. For instance, if we hover over the first red underline, the following appears:
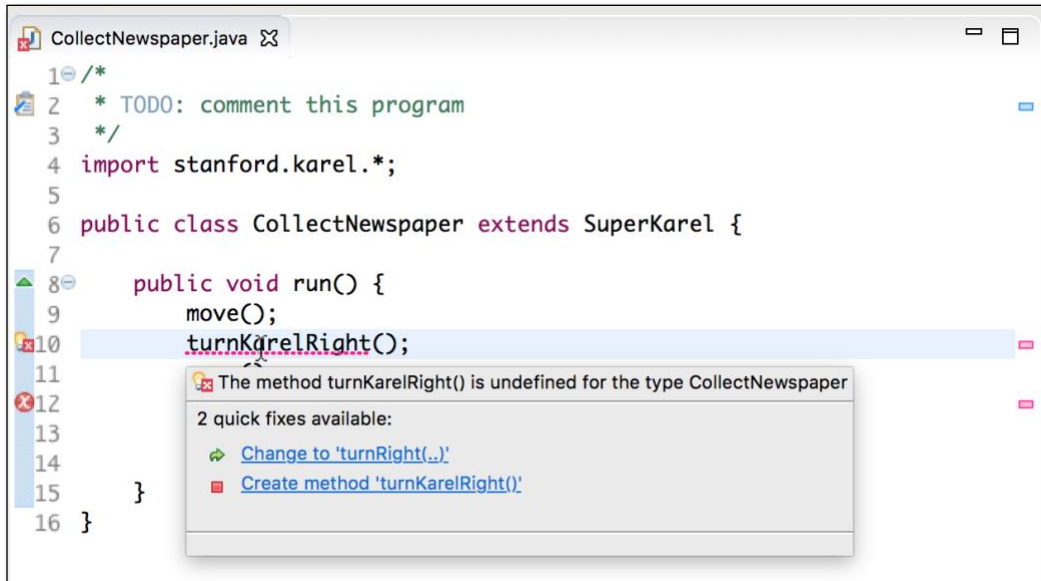
You can also view a summary of all problems in the **Problems** area at the bottom of the screen. Click the arrow to expand and view more information about them:
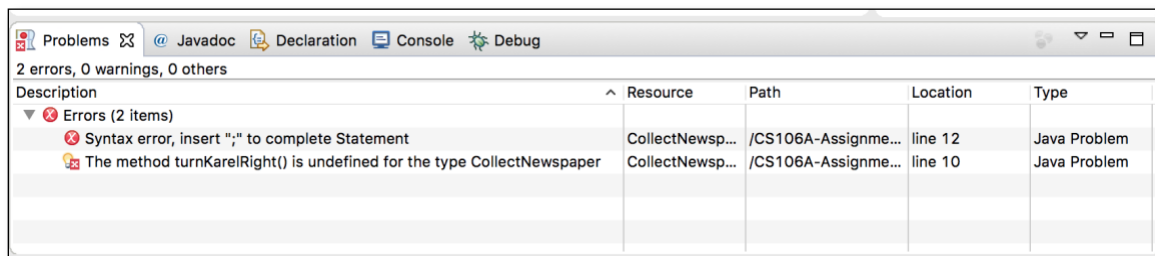

The first error message says that there is a missing semicolon at the end of the indicated line. This type of error is called a **syntax error** because you have done something that violates the syntactic rules of Java. Syntax errors are easy to discover because Eclipse finds them for you. You can then go add the missing semicolon and save the file again.

The second error says that SuperKarel doesn't understand the command `turnKarelRight()`; it only understands `turnRight()`. To fix this, correct the name of the command. Fixing this problem leads to a successful compilation in which no errors are reported in the **Problems** screen.

We're not done yet—Karel doesn't return to its starting position and we haven't decomposed the problem to match the assignment requirements—but you should try

running **CollectNewspaperKarel** to make sure that Karel can at least pick up the newspaper. If you haven't already, see the Eclipse instructions on the course website for how to run programs. When you run it, Eclipse will start the Karel simulator and, after several seconds, display a window that looks like the picture below:
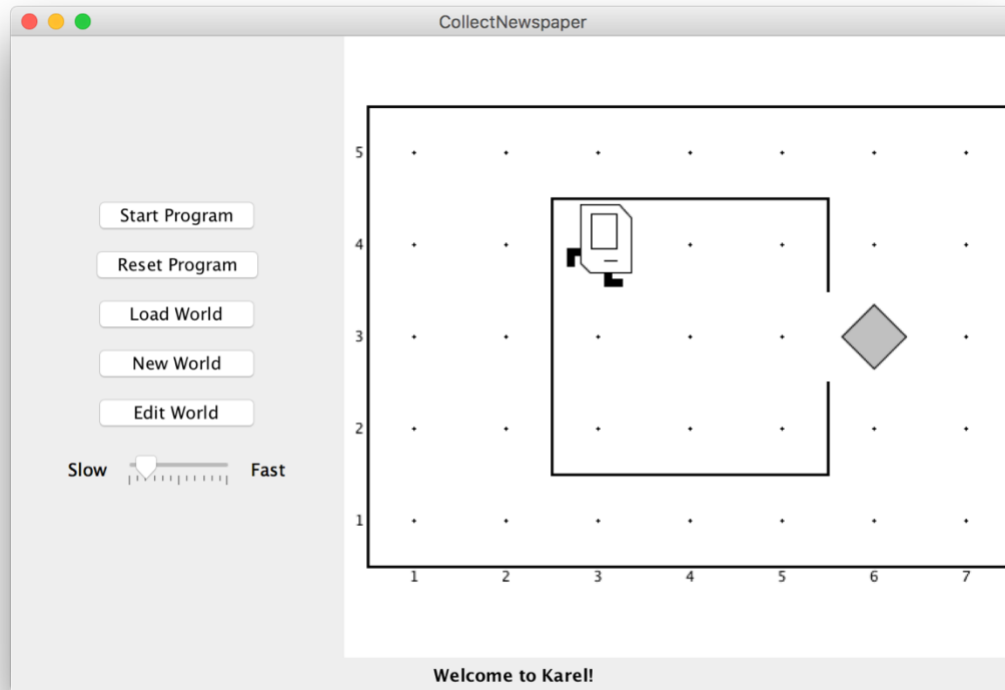
If you then press the **Start Program** button, Karel will go through the steps in the **run** method you supplied. Afterwards, if you press the **Reset Program** button, Karel will reset to its initial position, ready to run again.

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:
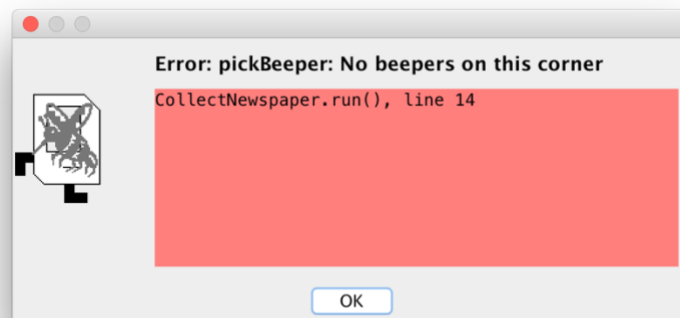
This is an example of a **logic error,** which occurs when you have correctly followed the syntactic rules of the language but nonetheless have written a program that does not correctly solve the problem. Unlike syntax errors, the compiler offers less help; it can tell you what the error is and sometimes what command caused it (line 14), but not how to fix the program so it functions correctly. Occasionally, it will list multiple lines in the error box, which means that the first method was called by the second, which was caused by the third, and so on (This is sometimes called the program's "call stack"). So, the program we'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 do. 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 <u>is</u> doing than to understand what it <u>isn't</u> 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. Each world is saved as a file within your project's **worlds/** folder. To modify worlds, see the three "World" buttons on Karel's control panel:

> Load World
>
> New World
>
> Edit World

The **Load World** button lets you select an existing world to open, **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 below:

> Save World
>
> Don't Save

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

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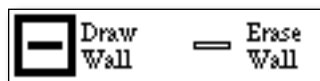
allow you to create and remove walls. The dark square shows that the **Draw Wall** tool is currently selected. If you go to the map and click on the spaces between corners, walls will be created in those spaces. If you later need to remove those walls, you can click on the **Erase Wall** tool and then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