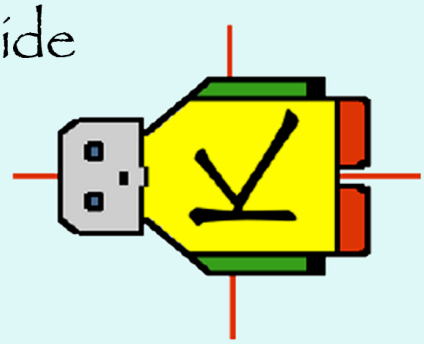
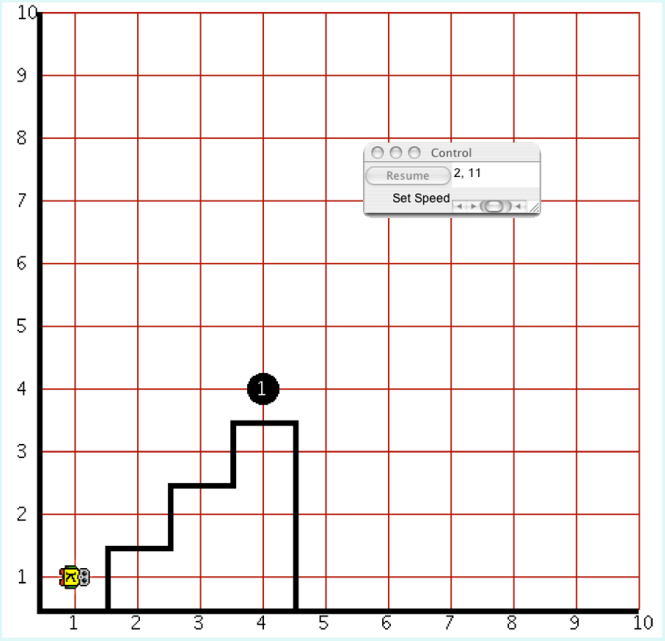
Karel J Robot
Simulator User's Guide



Joseph Bergin



Programmer _____

Karel J Robot
Simulator User's Guide
Joseph Bergin

March 15, 2005

1 Introduction

This guide will explain the workings of the simulator that accompanies the text: *Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*. The text is available from http://www.cafepress.com/kareljrobot. This document will not describe how to program in the Karel J Robot (KJR) world. For that you need the book.

2 Using the Simulator in Various Environments

2.1Just the JDK

You may want to use KJR without any integrated development environment (IDE) at all. For this you only need the Java Development Kit (JDK) from http://java.sun.com. Any version of the JDK since Java 1.1.7 should do, though a few of the older ones were buggy. The simulator was written being very cautious about features, so that it should run well on older versions. It has not been tested with Java 1.5 (Java 5) however. Install the JDK and take all of the default options. It should install properly without additional steps that were needed in the past.

You will also need an editor that is capable of creating plain text files. On the PC, notepad is not recommended, but others will serve. I suggest you don't use Microsoft Word for this, though WordPad works fine. When you write robot programs, the extension needs to be .java and some editors, like notepad, really only like the extension to be .txt so you might think you have StairClimber.java but really have StairClimber.java.txt. It won't work if this happens.

On your system, establish a working directory for yourself. It should not be within the distribution directory of the JDK. In particular it should not be in the bin directory of that distribution. That is always a dangerous practice and can defeat the security arrangements of Java. You should create a new directory/folder using system commands. If you work on a PC, it might be something like: c:\JavaProjects. You can do everything within this directory. If you put your robot code into packages, then one directory is enough and the compiler will create others for you as needed. The KarelJRobot.jar file that contains the simulator should be in this working directory. If you use the Test Driven Development subsystem, then junit.jar belongs here also.

There are two things you need to do to use the system. You need to compile your code with the java compiler (javac) and then you need to run it with the runtime interpreter (java). You will need a command/shell window for both of these and you will need to set your "current directory" to your working directory by giving the directory change command in your command window. Usually this is done with the cd command. You always work from this directory. The prompt in the command window probably shows you which directory you are connected to (which is "current").

Suppose you have written a robot program stored in StairClimber.java. Its public class is StairClimber, and suppose also that you have a main function in this class. Then, to compile this class and make it available for the interpreter you give the following command (on a PC):

```
javac -d . -cp .;KarelJRobot.jar StairClimber.java
```

Notice the period after the d option (and a space). If you are running on a unix machine (including linux and Mac OS 10.x) replace the single semicolon in the above with a colon. Each is the respective path separation character. Note that everything here should be considered case sensitive, even if your system (the PC, for example) has a case-insensitive file system. The java tools like javac are case sensitive.

If you get an error that suggests the javac command is not found, your java installation is incorrect.

If there are errors in your java file they will be reported to you in the command window. They must be fixed and the above command reissued until it succeeds.

If your class was in a java package, then a new sub folder with the package name has been created in your working directory and it contains one or more files with the ".class" extension. If it is not in a package, then the class files are in your working directory. You have then completed the first step. You need to repeat this if you ever change StairClimber.java, but not otherwise.

Once you have created the class files with the compiler you may execute the code by using the interpreter. We continue to assume that your StairClimber class contains a proper main function. Then, to execute this main function, beginning the simulation, issue the command:

```
java –cp .;KarelJRobot.jar StairClimber
```

If you put your class into a package named, say, karel, then the class files will be in the karel subdirectory and the above command changes to

```
java –cp .;KarelJRobot.jar karel.StairClimber
```

but is issued from the working directory and NOT from the karel subdirectory.

Note that if the main function of the class does not contain a command:

```
World.setVisible(true);
```

then the simulation window will not appear. See the section on Worlds in this document.

There are a few versions of the JDK in which you might have to spell out the cp option as *classpath* instead.

2.2 BlueJ

BlueJ is an IDE intended for novices. It is very nice and has a class design layer as well. First get BlueJ from http://www.bluej.org. Install it according to the directions.

Before you open BlueJ, create a text file named Karel.tmpl with a text editor. The contents of the file should be (exactly).

```
$PKGLINE
import kareltherobot.*;
/**
 * @author :
 * date:
 */

public class $CLASSNAME extends UrRobot
{
     public $CLASSNAME(int street, int avenue,
        Direction direction, int beepers)
     {
          super(street, avenue, direction, beepers);
     }
}
```

Save this file and put a copy into your BlueJ library folder at: BlueJ ->lib ->english ->templates ->newclass. On the PC this should be easy. On the Macintosh (OSX) it is a bit harder, as you need to open the BlueJ package by right clicking on the BlueJ icon and selecting *Show Package Contents* from the popup. The path in the resulting window is a bit different from the above: Contents ->Resources ->Java -> english ->templates ->newclass. Thanks to Dave Wittry of Troy Highschool in Fullerton, CA for this tip.

You can also make the Karel J Robot JavaDocs available within BlueJ. To do this:
Find the file bluej.defs in the lib folder. (On the Macintosh OSX you must again *Show Package Contents* for the BlueJ application. From there the path is: Contents->Resources->Java)

In that file, find the section that describes the help system. Just after it add the following lines

```
bluej.help.items=karel
bluej.help.karel.label=Karel J Robot
bluej.help.karel.url=c:/myjavastuff/KJRDocs/index.html
```

You will need to supply your own url path to the KJRDocs index file, of course. Note that the slash is forward. On the Macintosh, you can start with your own home directory, if the KJRDocs are there, rather than the system root.

Once you do this, you will see a Karel J Robot item in the BlueJ Help menu that will bring up the JavaDocs for KJR in your default browser.

Once you have tailored BlueJ for Karel J Robot, do the following:

1. Open BlueJ and create a new BlueJ project (from *File* menu). I named mine 'karel', but that doesn't matter much.

2. In BlueJ, select *Tools-Preferences-Libraries*. Select *'Add'* and add the file 'KarelJRobot.jar' from your Karel installation as a user library to BlueJ. Quit and restart BlueJ. BlueJ now knows the Karel J Robot classes and you can refer to them.

3. Create a package if you like. It will give you an new package icon, which you can enter by double clicking on it. In your new 'karel' project (within the package of you like), click the *New Class* button. You will get a dialog. Give the class a name and select *Karel* in the list of bullets. Click *OK*. You should now have an icon with the name of your class. If you double click this icon you can edit the file.

4. Notice that your file has
`import karelтherobot.*;`
at the top of the class and it extends UrRobot. You may need to change the parent class. It also has one necessary constructor that you can edit as needed. If you created the file inside a package it will also have the package header.

5. Copy any world file such as 'test.kwld' from the original download directory into the new 'karel' project directory.

6. When your class is complete, compile and run. When you compile you will be guided to fix the errors if any.

Also in BlueJ you need a way to get the world to show and to otherwise manipulate the world itself. I suggest that you create an Initializer class and give it to your students for modification. It should be in whatever package your students build in. It will therefore be visible in the BlueJ main design pane and you can send the initializeWorld message by clicking as usual.

```
/*
 * Modify this to manipulate the World.
*/
public class Initializer
{
    public static void initializeWorld()
    {
        karelтherobot.World.setSize(12, 12);
        karelтherobot.World.setDelay(50);
        //karelтherobot.World.readWorld("folder", "worldfile");
        //karelтherobot.World.showSpeedControl(true);
        // See the JavaDocs for class World
        //     for more options
        builder = new karelтherobot.WorldBuilder(true);
```

```
        kareltherobot.World.setVisible(true);
    }

     static kareltherobot.WorldBuilder builder = null;
}
```

It is strongly recommended that you put all your code into a package other than the default, unnamed, package. It can have your own name, for example. The default package is discouraged in Java. You cannot import a class defined in the default package into a class in another package (or at least I haven't figured out how).

Note that a flaw in BlueJ exposes some of our implementation details to the student. There is a lot of stuff hidden in our simulator that should not be so exposed. (being fixed)

BlueJ also integrates with the JUnit testing framework discussed below.

2.3 Eclipse

Eclipse is a professional, free, and open source, development environment with many capabilities. It is written in Java and so runs nearly everywhere (except Mac OS 9). To use Eclipse you must create a project for each program, though a program may consist of many files. A project contains all the necessary files and it manages the compiling and running of your code itself.

1. Download and install Eclipse from the web site (http://eclipse.org).
Eclipse manages its own files and will make copies of the files in the download directory as necessary. Unzip the karel distribution file in a place of your choice and then open Eclipse. The Left Hand pane of the window should show the *Hierarchy View*. Right click in this pane and select *New* > *Project*. Select *Java Project* in the dialog and then click *Next*. Give your project a name, say StairClimberProject. If you want to also use the Test Driven Development framework click *Next* again (otherwise click *Finish*). To use Test Driven Development, click the *Libraries* tab and then the *Add External Jars* button. Navigate to the plugins directory of your Eclipse installation and select the org.junit... folder (currently org.junit.3.8.1, but this may change). Inside that folder select junit.jar and click *Open*. Then click *Finish*.

2. Find the window in your operating system that you used to unzip the distribution and drag the KarelTheRobot.jar file over the Project icon in the Hierarchy pane and drop it there. Alternatively, you can add this as an external jar just as you did the junit.jar above. Only the navigation will be different.

3. Right click again on the new project file (with the name you just created). Select *New* > *Package* in the popup menu. Give this package the name "karel" without the quotes, and with no capitalization. You will create your classes in this directory. Note that this assumes you want your classes in the karel package. If you don't want to use packages at all (though, really, you should) you can skip this step altogether. If you don't want packages you will be using the *default package* in any case.

You can drop existing java files onto this package icon provided they have a "*package karel;"* statement at the beginning, of course.

You can also create new classes. Right click on the package icon and select *New* > *Class*. Give your class a name and a superclass. You may want karelterobot.UrRobot, perhaps, if you are creating a class of robots. You can browse to the superclass with the Browse button. That opens a window that has a search field at the top. Type the initial part of a class name here and it will show all of the possibilities that begin that way. Back on the Class naming dialog, don't forget to fill in any needed interfaces. You can browse for these also. *karelterobot.Directions* is frequently needed. (BTW, "*Enclosing type*" is for creating inner classes.) The three check boxes at the bottom let you add a main automatically, if your class needs one, as well as putting in stubs for inherited abstract methods and superclass constructors. This is helpful for robot classes, of course, where you need the constructors.

Once you have your Java files you can build from the *Project* menu and run from the *Run* menu. Errors are shown in the *Tasks* pane at the bottom, which becomes the Console when you are actually running. The *Run* menu will also let you set command line parameters for an execution.

Eclipse has many other options (*Window* Menu) including one to automatically build when you save a file. It also integrates nicely with the JUnit testing framework mentioned below. You can find the project preferences by right clicking on the project icon in the left pane. The preferences option is at the bottom of the popup.

2.4 JCreator (PC Only)

JCreator (http://www.jcreator.com) is a simple but relatively complete, inexpensive, Java IDE that runs on Windows. To use it you create a "project" for each program, though a program may consist of many files. A project contains all the necessary files and it manages the compiling and running of your code itself.

Create a working directory for yourself and put the files extracted from the distribution zip file there. This is the same as developing a working directory in the first section of this chapter. Your java and karel world files belong here.

JCreator uses workspaces as well as projects. A workspace is a set of related projects. You can have one workspace for all your Karel J Robot projects, for example. Create a new project from the *Project* menu. Select the *Basic Java Application* icon (or *Applet* if you prefer), and give it a name. For a location you can fill in (or browse to) any directory. I suggest that you browse to the working directory mentioned above, and then fill in a name for your project. The project files will be put in a folder named for your project within the working directory, so all will be easy to find and manage. Say, you call it TestProject.

JCreator will also create a file with the name of your project and add it to the project (TestProject.java). You can either discard this file (right mouse button) or edit it if you like. It will have a package statement, that you will want to change, probably to "karel". You can add

other files to the project by right clicking on the project icon itself in the left pane or selecting *Add Files* from the *Project* menu.

You need to make JCreator aware of the KarelJRobot.jar file. to do this, select *Project Settings* from the *Project* menu and click the *Required Libraries* tab. Click *New*, and then type a name (Karel) into the top field and click *Add* (Add package) and navigate to the jar file and select it and click *Open*. Then *OK*. Finally, make sure the little box next to the thing you just created (Karel) is checked and click *OK* again (Phew).
Now you can build your project. There are buttons for this, but you can also do it from the Build menu or use F7.

Before you can run, however, you need to put a copy of KarelJRobot.jar into the project directory. It needs to be both in the project itself, AND in the project directory. Not ideal, but I haven't found a shortcut.

Next, you need to tell JCreator which file is your main file. When you *Execute Project* (F5) from the *Build* menu you will get a dialog. Use it to navigate to the .class file of your main. This should be below your project directory in a directory called "classes" and further in a subdirectory named for your package (karel). Perhaps it is KarelMain.class in this directory. Select it to run.

2.5 CodeWarrior

Codewarrior (http://www.metrowerks.com) is a professional quality development environment that is inexpensive for education and very complete. It runs on many platforms and supports many languages. If you want to do Java on Macintosh OS 9 this is about your only choice, except for the online version in the JJ system. To use CW you create a "project" for each program, though a program may consist of many files. A project contains all the necessary files and it manages the compiling and running of your code itself.

Create a directory/folder in which to put the files from the distribution. Unzip the distribution file here. Your project files can be in this directory or anywhere else you like. To get started on a new program/project, open Codewarrior and select *New* from the *File* menu. Select *Java Stationery* in the dialog and give your project a name. You can navigate to the location in which you want the new folder to be created. Codewarrior likes to create a new folder for your project. In the next dialog you can select *Java 1.1* (or other, but this is sufficient), and then *Java Application* (or applet if that is what you want). CW will create a new project window for you that contains listings of all your files. If your system supports Drag and Drop, then drag the KarelJRobot.jar file to this window. Otherwise select *Add Files* from the *Project* menu and add it that way. Add any other files you want to the project similarly. If drag and drop works you may also be able to drag from one project window to another.

In addition to the window, CW has created a new folder for you as suggested above. On a Macintosh (OS 9) you may need to put the four supplied gif files into that folder. Otherwise I don't think you will need them (there are copies in the jar, actually).

CW has also added a file named TrivialApplication.java to your project. You can either discard this, or edit it, renaming it as appropriate. Remember that Java expects the file name and its single public class to have the same name. So if you edit it you may need to *Save AS...* so that you can change the name. You can also create new files and add them to your project.

Next, you need to tell CW what your main class is. In the project window there is a set of buttons near the top. One of them (looks like a page with wings to the left) brings up the *project* options (not the same as IDE preferences from the Edit menu). It is also found at *Java Application and Release Settings* in the *Edit* menu. In the project options select *Java Target* and fill in the name of your main class. It is TrivialApplication by default and you need to edit this. Remember also, that the name of a class includes its package structure, so if KarelMain is in package karel, then you need to enter

```
karel.KarelMain
```

into the main class field. You can set other options, of course, but if you work on a Macintosh (OS 9) you should leave the Java Output option set to its default "jar file" though the name doesn't matter. This is because OS 9 has a limit on the length of file names and with package prefixes, these can get quite long for inner classes. Files in a jar don't have this restriction. Save your changes to the project options and close the window.

World files should also go in the project folder so that they can be found without a path.
CW has a "feature" on the Macintosh (OS 9) that can be irritating. If you import a world from another system, it won't have its internal type and creator set. In fact it won't even have a "resource fork." If the file's type isn't 'TEXT' the CW won't open it, so you can't read it. If you open it in another editor that isn't so fussy (BB Edit, say or QUED) you should be able to save it to have its type changed to 'TEXT' so that CW can open it. Note that you don't change the EXTENSION of the file. That has no effect on the Macintosh. You need to change its internal type. The same is true if you use the world builder to create a world file. Java applications don't set the file's type, so you need to do this some other way if you think you need to. The extension can stay kwld or anything you like.

The restrictions mentioned above for OS 9 do not apply to OS X or to other systems.
Note that some versions of Codewarrior seem to require that the Direction class, which defines the actual values of North, South, etc., be referred to by Directions.Direction instead of just Direction. If you get a class not found error on karel.therobot.Direction, try this substitution. The error message should also point you to the line that needs the change.

2.6 JBuilder

JBuilder (http://www.borland.com/jbuilder) is a Java environment that is inexpensive or free for education. It runs on many platforms (Windows, Macintosh OS X, and unix). To use it you create a "project" for each program, though a program may consist of many files. A project contains all the necessary files and it manages the compiling and running of your code itself.

Create a working directory for yourself and put the files extracted from the distribution zip file there.

Open JBuilder and select *New Project* from the *File* menu. Give your project a name and navigate to the folder in which you want to store the files. I suggest a subdirectory of your working directory. Click *Finish*.

You will now have a project AND a new directory. Copy KarelJRobot.jar to this new directory along with any world files and any already existing java files you want in the project.

Go back to the project itself and right click on its icon in the left pane of the IDE. Select *properties* from the bottom of the menu. Click the *Paths* tab then the *Required Libraries* sub tab. Click *Add*. In the new Dialog, click *New...* Type a name into the top field of the new dialog (Karel), set the Location to be *Project*, and click *Add*. In the next dialog, navigate to the KarelJRobot.jar file and select it. Click *OK*. Back at the earlier Dialog click *OK* again. Back in a yet earlier dialog select the name you just added to Project and click *OK*. Back in the properties dialog click *OK* once again.

From the Project menu you can select *Add Files/Packages* to add additional files to your project, or you can create them with *New* from the File menu.

You can then *Make* your project from the *Project* menu. You must correct any errors in your java program and the *Make* again until there are no more errors.

You can then *Run* your project from the *Run* menu (or a button). You will need to navigate to the class file in the popup dialog. You can simply enter karel.KarelMain (or whatever the name of your file is).

If you want to change the main for a project once you have set it, select *Configurations* from the *Run* menu.

2.7 Other IDEs

For other IDEs the general principles are that you need create a project to write a program and you need to add the KarelJRobot.jar to your project. Some simple IDEs may let you write a program without a project, but this probably won't work that way. You may also need to teach the system to do something sensible with packages. If you get class not found exceptions it is either that the jar file is not correctly installed, or your package setup is incorrect.

3 Access to the World

3.1Building Worlds

Run the WorldBuilder in the jar file itself.

```
java -cp KarelJRobot.jar karelтherobot.WorldBuilder
```

The world builder will run, showing both the world, and the world builder dialog. See the figure, below.

On many systems, double clicking on the KarelJRobot.jar file should launch the world builder.

In a robot program, you can open the world builder by creating a new one:
```
WorldBuilder builder = new WorldBuilder(true);
```
If the parameter is true, as here, the speed dialog will also be shown. The speed dialog is independent of the world builder, but has a way to speed up/slow down robot execution, as well as pause the action (only) of multi-threaded robots.

Pick up a tool in the tools (button) panel. The name of the current tool is highlighted.
Click where you want to place an item controlled by that tool.

For beepers, click near an intersection when using the beeper tool. The intersection will be shown in the world builder dialog when you hover the mouse near the corner.

For walls, click across streets (for vertical walls) and across avenues (for horizontal walls) when using the corresponding tool. Likewise, the location at which the wall will be placed will appear in the dialog.

To put more than one beeper on a corner, click more than once with the beeper tool active.

To lower the number of beepers on a corner, control-click the corner with the beeper tool. On the Macintosh, use option-click instead of control-click.

To put an infinite number of beepers on a corner, control-click (with the beeper tool) when there are none on the corner. On the Macintosh, use option-click instead of control-click.

To clear all the beepers from a corner, shift-click the corner when using the beeper tool.

To remove a wall from across a street, control-click the wall segment with the appropriate wall tool. On the Macintosh, use option-click instead of control-click.

To change the number of streets or avenues in the world, enter an integer into the corresponding text field and hit enter. Note that this doesn't restrict the world in any way. It just determines how much of it is drawn with street and avenues.

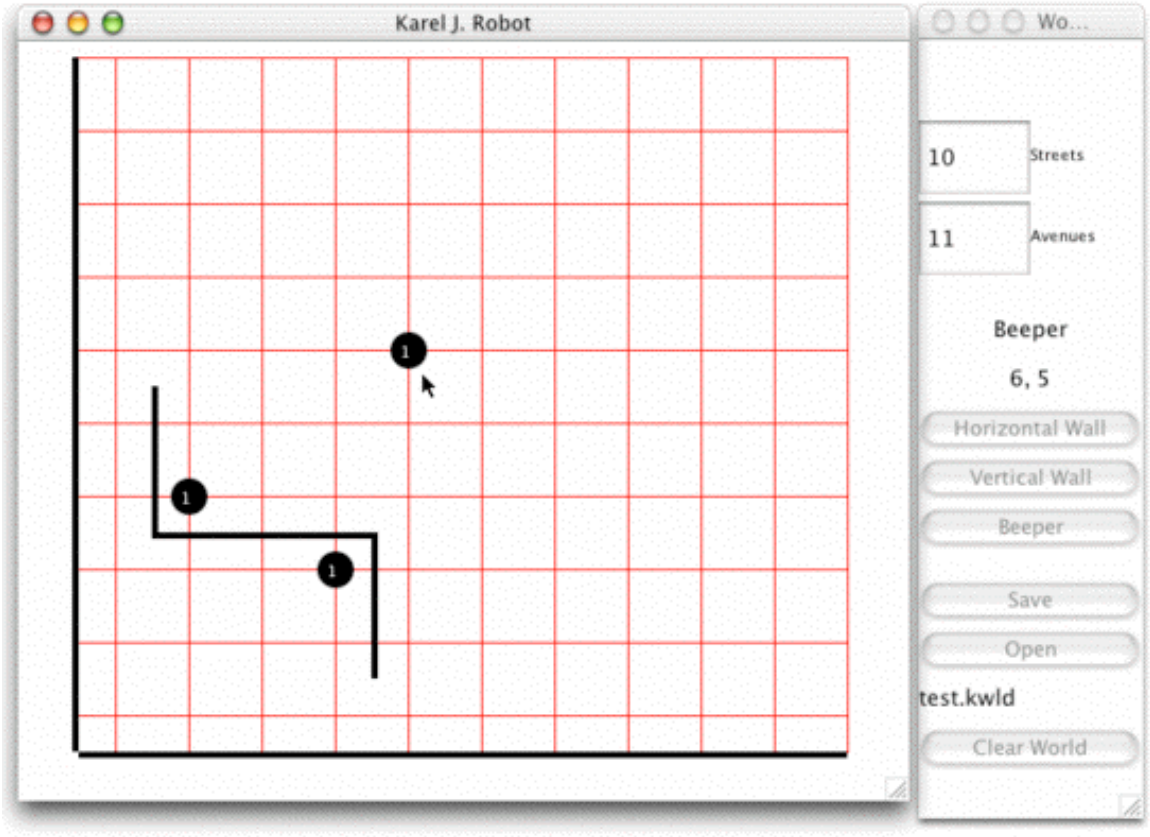
Save your world whenever you like. I've used an extension kwld here, but there is nothing special about it. It is neither assumed nor automatically appended.

You can also open an existing world to edit/modify it.
The *Clear World* button will remove all elements from the world. (Be careful, there is no UNDO here.)

Quit the world builder by closing either of its windows. If your world has been modified since the last Save you will be prompted to save it again. Note that if you QUIT using a menu or command key, this extra Save prompt will NOT occur.

If you use an IDE and want to run from there, make a separate project for this and include the jar file. Make karelther obot.WorldBuilder the main file for the project. In Codewarrior (on the

Macintosh at least) you need to set the output to be a class folder rather than a jar file, since the main is in the existing jar file, and not in any code of your own. After you build this class folder will be empty.



Here is the World Builder running on a Macintosh (OSX). The look is similar on other platforms. Since it is written in Java 1.1 using AWT, it uses the look and feel of your platform. This is an older figure. It now has a button to show/hide the speed dialog.

World builder shows the position of the mouse in street,avenue coordinates if you have selected any tool. If the current tool is the Beeper tool, it shows the nearest corner (where the beeper will be placed if you click). If the current tool is the Horizontal Wall tool, it shows where the corner below the wall that will be placed. If the current tool is the Vertical Wall tool it shows the corner to the left of the wall that will be placed.

You can also create a world using program statements by sending messages to the World. You may then save them with the saveWorld command. See the JavaDocs. The basic options are:

```
World.placeBeepers(int street, int avenue, int howmany);
World.placeEWWall( int NorthOfStreet, int atAvenue,
     int lengthTowardEast);
World.placeNSWall (int atStreet, int EastOfAvenue,
     int lengthTowardNorth);
World.saveWorld(String filename);
```

Finally, you can run a robot program and save the results, including new placements of beepers that occurred as the program ran, by saving the world at the end.

3.2 Using Worlds

Remember the name you used when you saved it. Say test.kwld.

Include a statement in your robot program to open it. Include this at the beginning of your main task block.

```
World.readWorld("test.kwld");
World.setVisible(true);
```

The world file will be opened and its content merged into your existing world. If you store your world files in a directory other than the working directory for your project, there is another form of the readWorld method in which you can name a path to your file in a separate (first) parameter. See the JavaDocs for Karel J Robot for details.

The world is not visible by default here, so the setVisible command is needed.

Note on the files created by the world builder.
The format is compatible with that of the very early Karel J Robot simulators. However, there is no need to use the older version as this is an extension.

You can actually write the world files with a text editor.
Lines that do not start with a keyword for the builder are ignored. Elements can appear in any order. They are NOT case sensitive.
Here is a sample file.

```
KarelWorld
Streets 10
Avenues 20
beepers 3 4 1
beepers 4 2 1
eastwestwalls 3 2 4
northsouthwalls 4 2 2
northsouthwalls 4 3 3
northsouthwalls 1 5 5
northsouthwalls 1 4 4
```

The first line (KarelWorld) is not a keyword, but it might be in the future. It might also contain a version number eventually. It is ignored when reading now, but written when you save a world.
The following gives the meaning of the current keywords. This is not expected to change.

- Streets --The number of streets to draw.
- Avenues --The number of avenues to draw.
- beepers --The data is: street, avenue, number of beepers, without punctuation of any kind.

- eastwestwalls -- The data is: North of Street, first avenue crossed, last avenue crossed. Last should be bigger than first (or equal).
- northsouthwalls -- The data is: East of Avenue, first street crossed, last street crossed. Last should be bigger than first (or equal).

The world builder writes out a separate wall command for each segment of a wall (one block long). If you do it by hand, you can combine them and create long walls in one command.

If you create a world in a Robot program using the programming interface and save it, it will be consistent with the world builder files.

You can erase everything (including robots) in a world with
```
World.reset();
```

4 Writing Your Robot Classes

When you write robot code you need to include a few things in your classes.

I recommend that you put your code into a package of your own naming. The package statement must come first in your program. I don't recommend you use the karelltherobot package that contains the simulator itself, but a karel package is fine.
```
package karel;
```

Next should come your imports. You probably want to import everything in the karelltherobot package
```
import karelltherobot.*;
```

In many ways it is better to import individual classes, rather than the above. In this case you often need something like
```
import karelltherobot.UrRobot;
import karelltherobot.World;
import karelltherobot.Directions;
```

If you give your robots color badges, you also need to import java.awt.Color.

You should write one (only one) public class in each java file. Inner classes, if you use them, don't count here. They may be public or private as you prefer.

Your robot class will need at least one constructor. Each of your constructors must invoke one of the two constructors of its superclass unless that superclass happens to have a no-argument constructor. UrRobot and Robot do not have such constructors so you need to invoke super in most cases giving a street, avenue, Direction, number of Beepers, and (optional) a badge color. Usually you just provide a constructor with these same parameters.

```
public StairClimber(int street, int avenue, Direction direction,
int beepers, Color color)
```

```
{
     super(street, avenue, direction, beepers, color);
}
```

In some problems it makes sense to use default values for some of these, so the constructor form isn't strictly required, but the super call must supply some values for each argument. For example, if the problem requires that all such robots carry an infinite number of beepers, then the constructor might reasonably be:

```
public RoomFiller(int street, int avenue, Direction direction,
Color color)
{
     super(street, avenue, direction, infinity, color);
}
```

The Color parameter is always optional.

You will need a main or a task (see the next section). If you write a main you can put it into one of your robot classes or into one by itself (public, in a separate file). Main functions need to be in a public class to be useful. If you use a separate class (recommended) it doesn't extend any robot class and therefore must implement the Directions interface if you want to use the direction names or `infinity` within the main method. You can give world commands as well as create and exercise robots in the main.

I usually separate the world commands from the main, however, into a static block. A static block is an anonymous method that is automatically executed when the class that contains it is loaded.

```
static
{
     World.setDelay(1);
     World.readWorld("worlds", "stairworld.kwld");
     World.setVisible(true);
     World.showSpeedControl(true);
}
```

However, if you write a robot class that you intend to use as a superclass of other classes, like the supplied AugmentedRobot.java, then you should not use a static block there, as it will be executed whenever you create a robot in any subclass of this one because the superclass must be loaded also.

5 Options for Running (main vs task)

5.1 main

In Java, any public class can have a main method and any of these can begin the execution of a task from the operating system. Usually main is very small. It creates a few objects and sends a

message or two and the objects do the real work. Chapter 2 of the text shows many examples of main methods. The form is required by Java and must be strictly adhered to. Even IDEs like BlueJ that let you avoid main, still have one, though it is hidden from the user. When you issue a java command from the command line, you always name a class that contains such a main method.

5.2 task

If you want to avoid the ugliness of main, there is another option supported by Karel J Robot.

Build a new in package *karelltherobot*. Your class must implements the *RobotTask* interface. (This extends *Directions*, so you don't need to implement both). Give this new class a method with protocol `public void task()`. Put all of your robot commands and any world commands inside this task method. Here is an example:

```
import karelltherobot.*;

class Test1    implements  RobotTask
{
     public void task()
     {
          World.readWorld("box.kwld");
          UrRobot karel = new UrRobot(3, 3, East, infinity);

          karel.turnLeft();
          karel.move();
          karel.turnLeft();
          karel.move();
          karel.turnLeft();
          karel.move();
          karel.turnOff();
     }
}
```

The RobotTask interface requires only this one method.

Now run your program with either of the following commands:
`java -cp .;KarelJRobot.jar karelltherobot.KarelRunner Test1`
or
`java -cp .;KarelJRobot.jar karelltherobot.KarelRunner`

In the last case, you will be prompted for the name of the class and you can respond Test1. If your class is in a package, don't forget to prefix the name of your class with its package name. For example, if you put your code into package karel, then the full class name is karel.Test1.

If you give the classname in the above, there are two additional options you can give. An option begins with either a – or a / character immediately followed by b or w (upper case accepted also).

A b option (-b or /b) will cause the world builder to be shown along with a speed dialog. If you give the w option you then leave a space and give the name of a world file to load for the first run. For example: –w stairworld.kwld.

After the task will is run you will be prompted at the end to ask if you want to run it again without restarting. The prompt appears on System.out. Any response that does not begin with the letter 'n' (or 'N') will be treated as a yes--even just a return. The task then be re-executed, including any world setup commands. There are two sample such test programs packaged in the distribution zip.

The prompt given is "Run robot task? Y/n/c/w " The first letter of your reply determines what is done. If you type just a return the default (Y) will be taken. If you type anything that begins with y or Y the task will be run. A response beginning with n or N halts the program. If your response begins with c or C the world will be cleared of all elements (walls, beepers, robots) before it is again run. Finally if your response is w (or W), then follow that letter with a space and then the name of a new world file to be loaded before your program is run. With this last option you can run the same program in different worlds without quitting. If you don't give the w option, the same world, if any, will be loaded again.

Note that on the Macintosh under OS 9 there is normally no command line, so there, the last form of the run is equivalent to double clicking on the application jar file that your IDE (Codewarrior, for example) creates. But that only works if you make karelltherobot.KarelRunner your main class, of course. This is the case if you use ANY IDE on the Mac or otherwise.

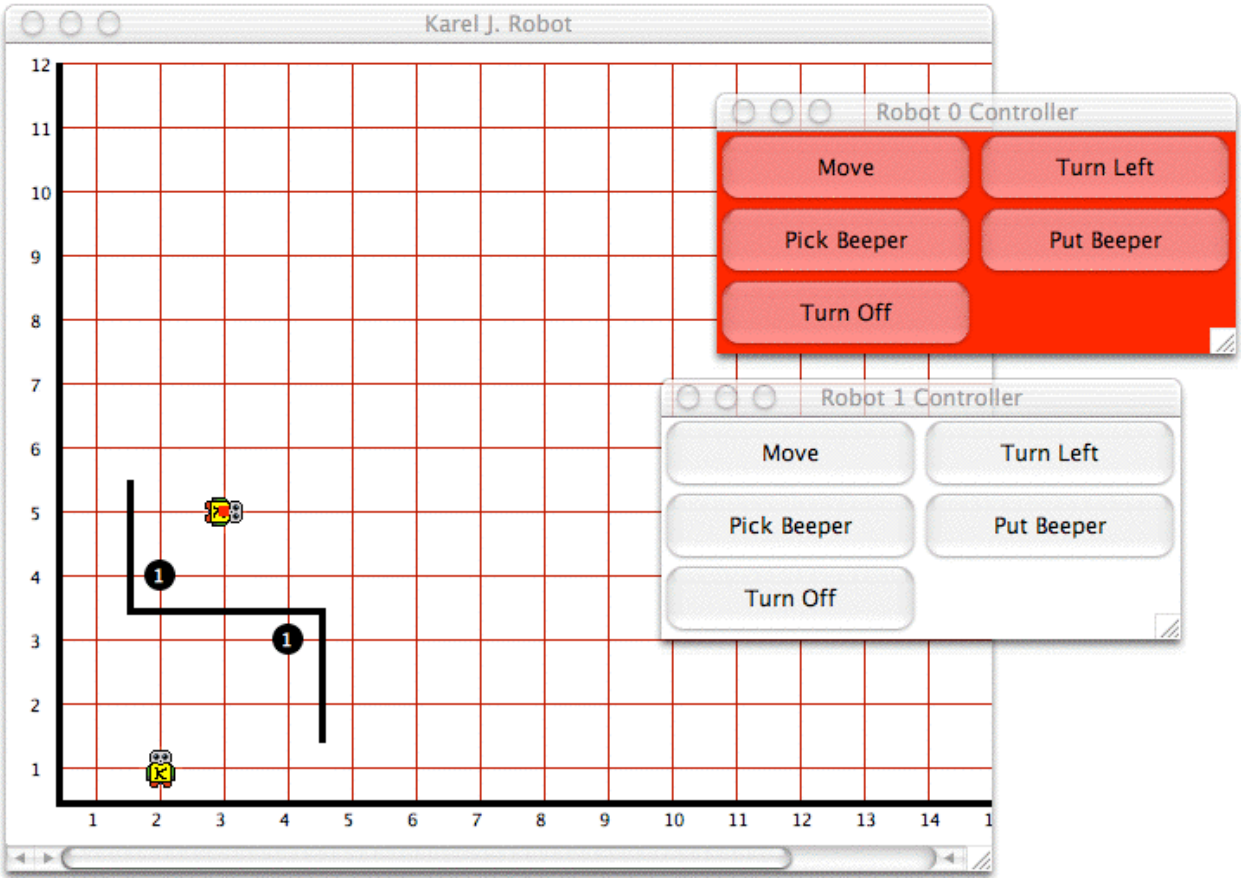
Also, if you run a task in which all robots run in separate threads, the prompt for rerun will come immediately after the robots are created. This can be a little disconcerting, as the task hasn't run yet and you are being asked to run it again. Eventually the robots will complete and your response to this prompt will have effect.

NOTE: March 15, 2003. KarelRunner was called KarelTest until recently. RobotTask was previously called RobotTester. This is a change for naming consistency with other items in the system. Also note that a RobotTask class should **not** also extend a robot class. These task classes only work if they have a no-argument constructor available and usually robot classes do not have such a constructor. RobotTasks are to exercise robot classes, not to BE robots. For example the RobotTask above (class Test1) exercises UrRobots.

6 Remote Control of Robots

There are two ways to remotely control a robot. You can use these to directly control a robot rather than writing a program that does so. A simple remote controller is a dialog linked to a UrRobot. These are created together. Here is a pair of commands that creates two robots and their associated controllers. (See Color in the Additional Features section of this document.)

```
RemoteControl controller = new RemoteControl(5, 3, East, 0, Color.red);
RemoteControl controller2 = new RemoteControl(1,2,North,0,null);
```



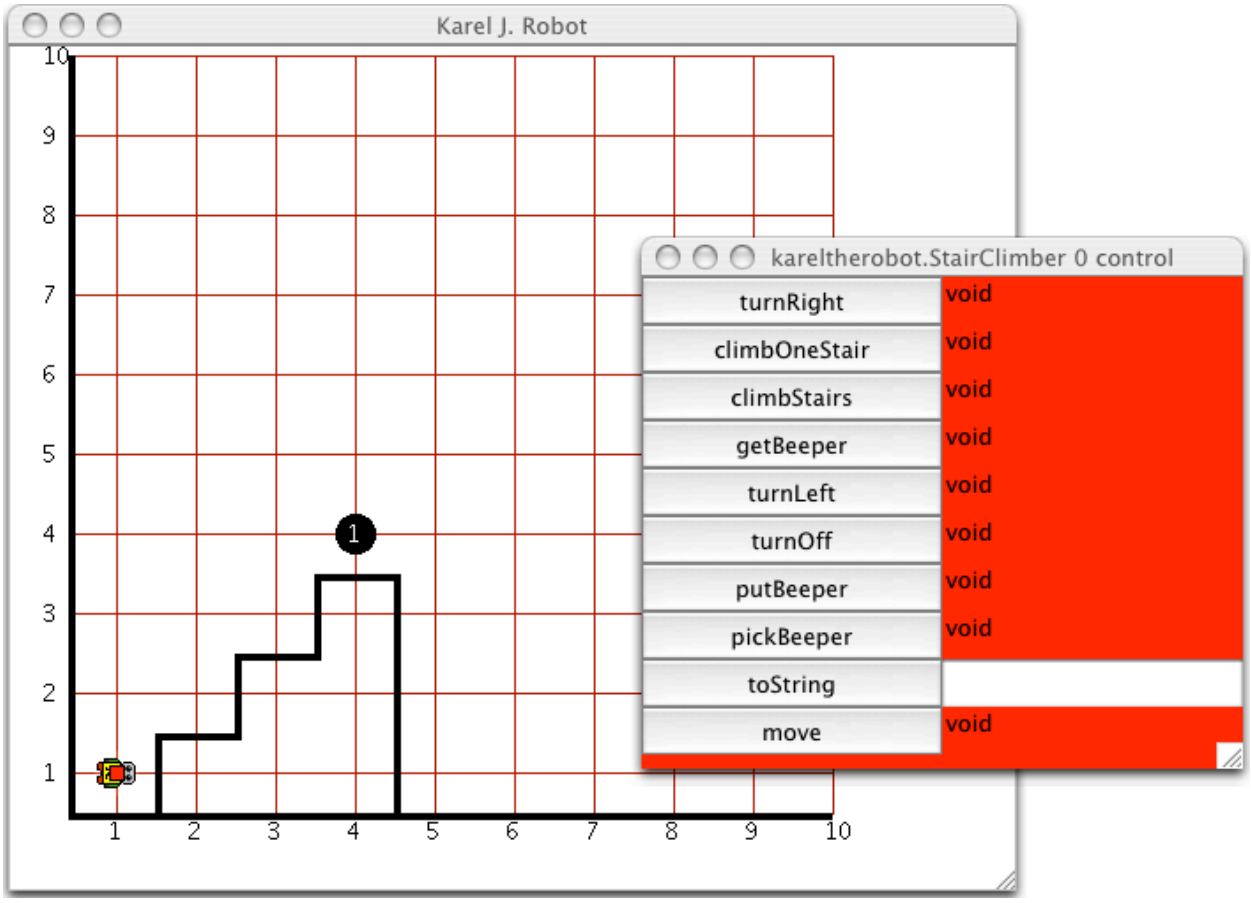
Notice that a robot with a red badge has a controller with a red background, and one with no badge has a white background.

Once you have the remote controller, you control the robot by pressing the buttons.

While the above will only create a remote controller for a UrRobot, you might want something similar for your own classes. Suppose you have StairClimber.java and want a controller. From a main or task block you can create an "AutoRemote."

```
World.readWorld(stairworld.kwld);
AutoRemote climberControl = new
AutoRemote(kareltherobot.StairClimber 1, 1, East, 0, Color.red);
```

Note that when this was written it was necessary to create classes used by AutoRemote within the kareltherobot package. This will be changed in the future. At that time, StairClimber was in that package also.



The first argument in the constructor of an AutoRemote is the fully qualified name of the robot class to instantiate. AutoRemote itself has another constructor (without a Color), but your class must implement the constructor with signature:

```
public MyRobot(int street, int avenue, Direction direction,
        int beepers, Color color);
```

or you will get an Exception. You can create as many of these as you like in your main, of course.

AutoRemote has a constructor with no arguments to create an UrRobot at 1,1, facing North with no beepers.

AutoRemote has another constructor with a single argument to name the class whose robot should be created. It will be created at 1, 1, facing North, with no beepers. This is much like what you get with RobotController described above.

7 Stepping Robots

You can now step through the operation of your robot programs. There are two options for this: Primitive stepping, and User stepping.

7.1 Primitive Stepping

You can make any robot pause just before executing any of the primitive instructions defined in Chapter 3 of the text or any of the predicates defined in Chapter 5 of the text with this feature. The robot will pause until you type a return/enter into the console window.

To make it pause you must turn on primitive stepping for this robot by sending it the setPause(true) message. You can do this anytime, but if you want to step through the entire program, do it just after you create the robot. Then, just before executing any primitive instruction or predicate it will pause and a message will appear in the console window telling you what it is about to do. So if your robot is named karel, you can use

```
karel.setPause(true);
```

to turn stepping on. Then, if karel executes a turnLeft instruction, it will first display (in the system console)

```
RobotID 0 is about to turnLeft.
```

(assuming it is the first robot (ID 0) you create) and will wait for you to type *enter/return* to continue. You can also turn off stepping for the robot by sending it the `setPause(false)` message. It will then behave as usual. It will neither print the messages, nor wait to continue.

7.2 User Stepping

You might also want to have the simulator pause elsewhere in your robot program: for example at the beginning of the methods that you write yourself. This requires two steps. You turn pausing on for any given robot by sending that robot the `setUserPause(true)` message. You can turn it off similarly. This alone will have no effect, however. You must also send the robot `userPause("do something")` messages at the points at which you want it to actually pause. If you have `setUserPause(true)` earlier in the program, then the robot will write the string you include as a parameter into the console window and will wait for you to type a return/enter. Actually the string will be prepended with "RobotID x is about to " so with the example message (above) sent to karel the console will actually show

```
RobotID 0 is about to do something.
```

Use this information to make your strings informative.

The usual way to use this is to put userPause messages at the beginning of your methods. For example

```
public void turnRight()
{       userPause("turn right");
        turnLeft();
        turnLeft();
        turnLeft();
}
```

Here the message goes to the "this" robot of course, just as the turnLeft messages do.

Note that by itself userPause(...) does nothing unless you turn on user pausing for that robot. This way you can leave userPause(...) messages in your programs and then have them pause or not at these points by turning user pausing on or off with setUserPause(true) and setUserPause(false).

Note finally, that you can turn on both Primitive stepping and User stepping. These are independent of each other.

8 Threaded Robots

You can run robots in several threads. This is discussed in Chapter 8 of the book, but can be used earlier with only a couple of tools. The first two sections of Chapter 8 are quite simple and explain all you need to make it happen. You could do a multiple-robot harvesting task quite easily this way. In order to coordinate action in a more complex case, however, you need to read more of Chapter 8.

To run treaaded robots you need the speed controller, since the threads startup all paused and you need to resume them before anything happens. The speed controller does this as well as set the speed at which they run. See Speed under Additional Features in this document for how to make it visible.

9 Test Driven Development

9.1 JUnit Testing

A modern technique for developing programs is called Test Driven Development in which tests are written before the code is. The "sound bite" is "No code without a failing test." The simulator supports this by providing a class, KJRTest, that extends JUnit's TestCase class. This class allows you to make assertions about the state of robots and the world. It is interesting, that while you cannot ask a robot where it is, you can assert where it is.

Complete details may be found online at:
http://csis.pace.edu/~bergin/KarelJava2ed/testInfected/

9.2 KarelFixture

Also useful for testing is another program, not part of the distribution, called KarelFixture. It permits you to exercise robots without a main (without regular programming, actually), as well as make assertions. KarelFixture is keyed to a special tool called Fitnesse (http://fitnesse.org), which also provides a wiki that you might find useful. You can learn more at http://dps.csis.pace.edu:8077/KarelFixture.

In particular, you can give an executable specification of a robot program using this fixture, by creating a table within FitNesse. Here is an example:

Here is what a specification for the StairClimber task looks like in this system:

karelfixture					
karelfixture.StairClimber					
Tester	karelfixture.KJRTest	local			
World	reset				
Read World	stairworld.kwld				
Create	karel	1	1	East	0
Message	karel	getBeeper			
Assert	BeepersInWorld	0			
Assert	BeepersInBeeperBag	karel			
Assert	FacingEast	karel			
Assert	NotRunning	karel			
Assert	At	karel	4	4	

It is important to note that this is an executable specification. You execute it by pushing a test button on the page that contains it (not here, visit the link above). When you test it with a correct program you get something like:

karelfixture					
karelfixture.StairClimber					
Tester	karelfixture.KJRTest	local			
World	reset				
Read World	stairworld.kwld				
Create	karel	1	1	East	0
Message	karel	getBeeper			
Assert	BeepersInWorld	0			
Assert	BeepersInBeeperBag	karel			
Assert	FacingEast	karel			
Assert	NotRunning	karel			
Assert	At	karel	4	4	

10 Additional Features

10.1 Color

The standard color used to draw the streets and avenues is a dark red. You can change this by sending the World a message.

```
World.setStreetColor(Color.blue);
```

will set the streets and avenues to blue. You must import java.awt.Color for this to work, of course.

You can also change the color of the beepers and the walls.
```
World.setBeeperColor(Color.magenta);
World.setNeutroniumColor(Color.green.darker());
```
will set the beepers to magenta and the walls (made of Neutronium) to a dark green.

You can now give any robot a small color "badge" that it will wear. If you have several robots in your world you can give them different color badges to keep them separate visually. If you have imported java.awt.Color you can create an UrRobot with a blue badge with
```
UrRobot karel = new UrRobot(1, 1, North, 0, Color.blue);
```
You will need to implement a constructor in your own classes to make this available to them, of course.

```
class StairClimber extends UrRobot
{
    public StairClimber(int street, int avenue,
          Direction direction, int beepers, Color badge)
      {   super(street, avenue, direction, beepers, badge);
      }
...

}
```

You may set the background color of the world with
```
World.setWorldColor(Color color).
```
See the Java Docs. Note that the colors you can now set, such as this one as well as walls, etc., are not saved in the world files. They are just a feature of the current run. The world file definition is unchanged.

10.2 Speed

You can set the speed of robot execution from the program by telling the world
```
World.setDelay(int howMuch);
```
The delay can range from 0 (fast) to 100 (very slow). On some systems a delay of 1 is faster than 0 due to a bug int the thread manager on such a system.

You can also set the delay with a slider in the speed dialog. You can make this dialog visible by telling the world:
```
World.showSpeedControl(true);
```
Note that the pause/start button only works when your robots run in threads, since they control the threads, not the robots. If we were to pause the thread in a program that only has one we couldn't start it up again (the dialog itself would be "running" in a paused thread), so for simple execution of robot programs this button has no effect.

You can make a robot named karel sleep for a bit with
```
karel.sleep();
```
The robot will sleep for a time dependent on the delay time that you set in the program or with the slider. The robot's thread will also yield to other threads when this happens.

10.3 Miscellaneous

By default, the robots print traces of their behavior on System.out. You can turn this on and off with the command
```
World.setTrace(boolean on);
```

At any time you can send a robot its `showState(String s)` message and it will print its current state on System.out. The string will prepend the normal output.

Normally your main function will want to create robots. It is usually necessary for the class that contains this main to either extend some robot class, such as UrRobot, or to implement Directions. Otherwise North, for example, becomes Directions.North.

The size of the visible portion of the world can be set with
```
World.setSize(int numberOfStreets, int numberOfAvenues);
```

The world is still conceptually infinite in two directions. This only affects how much is shown. A robot might walk outside this region while a program runs and later return to it (or not). Actually there are about four billion (2^{32}) streets and the same number of avenues. Not infinite, but large.

The robots now scale in size as you change the window size and/or the number of streets. This might help if you project a screen for students. At the present time scaling causes robots to occasionally disappear briefly.

There is an alternate version of World.setVisible that also has the dimensions of the window (pixels) in which to show the world. This is independent of World.setSize mentioned just above.