Learning to Program in Java
Using Lego™ Mindstorms® Robots
And LeJOS

By
William J Rust
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization section</td>
<td>101</td>
</tr>
<tr>
<td>Listener section</td>
<td>102</td>
</tr>
<tr>
<td>Javadoc</td>
<td>103</td>
</tr>
<tr>
<td>Robotic Behaviors</td>
<td>109</td>
</tr>
<tr>
<td>Finite State Machines</td>
<td>111</td>
</tr>
<tr>
<td>Behaviors in LeJOS</td>
<td>112</td>
</tr>
<tr>
<td>MazeRunner</td>
<td>113</td>
</tr>
<tr>
<td>Exercises</td>
<td>117</td>
</tr>
<tr>
<td>Navigation</td>
<td>118</td>
</tr>
<tr>
<td>History of Navigation</td>
<td>118</td>
</tr>
<tr>
<td>Coordinate Systems</td>
<td>118</td>
</tr>
<tr>
<td>NavBot.java – A Navigation Example</td>
<td>123</td>
</tr>
<tr>
<td>Programming a Robot to go here, there and back</td>
<td>124</td>
</tr>
<tr>
<td>MapBot.java - the main program</td>
<td>124</td>
</tr>
<tr>
<td>Move – the lowest priority behavior</td>
<td>125</td>
</tr>
<tr>
<td>Bump – the middle priority behavior</td>
<td>126</td>
</tr>
<tr>
<td>Home.java – the highest priority behavior</td>
<td>127</td>
</tr>
<tr>
<td>Wrapping up Navigation</td>
<td>128</td>
</tr>
<tr>
<td>Exercises</td>
<td>128</td>
</tr>
<tr>
<td>Communications</td>
<td>129</td>
</tr>
<tr>
<td>PC to RCX Communications</td>
<td>131</td>
</tr>
<tr>
<td>PCSend.java</td>
<td>131</td>
</tr>
<tr>
<td>RCXRecv.java</td>
<td>132</td>
</tr>
<tr>
<td>RCX to PC Communications</td>
<td>134</td>
</tr>
<tr>
<td>Exercises</td>
<td>135</td>
</tr>
<tr>
<td>Graphical User Interfaces – GUI</td>
<td>136</td>
</tr>
<tr>
<td>Flavors of Java</td>
<td>136</td>
</tr>
<tr>
<td>Swing and AWT</td>
<td>137</td>
</tr>
<tr>
<td>HelloWorldG.java</td>
<td>139</td>
</tr>
<tr>
<td>HelloWorldG2.java</td>
<td>141</td>
</tr>
<tr>
<td>DummyMapData.java</td>
<td>142</td>
</tr>
</tbody>
</table>

© 2005, William J Rust
Introduction

Cognition

The purpose of this book is to teach you how to program in Java. Learning to program can be hard. Learning to program well in a language as robust as Java can be really hard. Or, it can be fairly easy. Your attitude can make all the difference.

Java is grounded in object-oriented programming. Objects are a meta-cognitive concept. While that may sound like gobbledygook, understanding what meta-cognitive means is the key to this book. The word cognitive refers to thinking. A cognitive process is simply how you think about something. When you play a game, you usually come up with a strategy to win the game (assuming that you want to win). That strategy is a cognitive process. Meta-cognitive is one step beyond cognitive, it is thinking about how to think. Tic-tac-toe is a game most children learn in school. It doesn’t take long for most people to learn that there is no winning strategy for tic-tac-toe. That is, if both players make the proper moves, every game will end in a draw. The concept of unwinnable games is a meta-cognitive idea.

Piaget, a noted educational theorist, devised a list of stages in cognitive development. He observed children over time and concluded that most children start becoming meta-cognitive around 12 to 14 years old. That seems to be minimum age to learn how to program in Java because of the meta-cognitive nature of objects. While younger children can learn to program, it is our belief that only those who have entered the meta-cognitive stage of intellectual development should attempt to do so.

While meta-cognitive ability is essential to programming, educational researchers over the years have found that manipulatives can significantly improve learning. A manipulative is simply a physical thing that is essential to the instructional strategy (pedagogy). For example, some teachers use counting sticks to help young children learn their addition facts. Having a physical artifact helps
many young children learn a fairly abstract body of knowledge. Similarly, in high school biology classes, dissections are an example of a manipulative used with older students.

For this book, the Lego™ Mindstorms™ RIS is the essential manipulative. Throughout this book, there is example code that uses the RIS. It is expected that the student will build robots and program them as part of the course of study. Without the activity of building and programming, the results of using this book is likely to be very poor. Further, the activities through the book build sequentially. For example, the first chapter has nothing to do with Java programming. It is about building a particular robot, Roverbot, and using the RIS programming environment to control it. This seems to have little to do with learning Java. However, it ensures that you have 1) a working IR link, 2) a working robot and 3) familiarity with robot commands that you will implement later in Java.

Learning to program requires developing skills in using programming tools and skills in learning how to think in programming terms. These are two different types of skills. Becoming proficient in the use of programming tools makes it much easier to do your own programming. In the beginning there will be a lot of programming tools skill development exercises. You will be given correct, or nearly correct, programs to compile and run. Before you actually begin writing programs you will be correcting other people’s programs. While this may seem paradoxical, how can you correct programs before you know how to program, this technique is known as scaffolding and will provide you with the tools for successfully programming robots.

**Programming as technical writing**

Programming is a form of technical writing. This is a major theme of this book and will be repeated often. When you write something, you write it with an audience in mind. Schoolwork is intended for a teacher. Diaries are intended for yourself. Newspaper articles are intended for your community which can be either the people in your school or the people in your town, depending upon the newspaper. Programs are intended for a computer. Computers are not very smart; when you write for a computer you must be very
precise in both the language you use and the ideas that you want to convey. But computers are very fast so even though computers can only do simple operations, they do so many simple things in such a short period of time that they can accomplish significant feats. The goal of a computer programmer is to write so clearly and concisely that the computer does what the programmer wants to accomplish.

**Outline of topics**

This book starts by installing and using the Lego® Mindstorms™ RIS package. There are several reasons for starting with this package. First, it introduces you to a variety of computer topics. These include installing software and hardware, using integrated development environments, IDEs, managing files, real-time, event-driven systems, etc. Second, RIS has a training mission section that forces you to become familiar with pieces in the kit and some concepts of robotics. Third, it allows you to intrinsically understand what you want to accomplish on the computer before you try to write in Java. Next, the book leads you step by step through the installation of the IDE, compiler, and robot environment. Just getting the tools necessary to write a program installed is a difficult task and it is extremely important to accomplish this task before trying to write a program. Third, the book explores what an object is and how that concept is important when programming in Java. Fourth, the book compares RIS code to Java code and then steps through the writing of fundamental classes for programming robots. Fifth, the book introduces several topics related to robots in the real-world including programming using behaviors, navigation, communications and a very basic introduction to display data on host PCs using Swing and AWT. Finally, the appendix is an abridged version of the Java Language Specification. The book also includes a plethora of code examples that can be compiled and run on the robots.

This book is an introduction to Java.
Robotics Invention System -- RIS

Install RIS software

Install the RIS software on the computer you will be using to do your Java programming exercises. The computer needs a minimum of roughly a 400 MHz Pentium class processor, 256 MB RAM, 200 MB of free disk space, CD ROM and Windows 98 operating system. These are minimum requirements. More memory, a faster processor or a later version of Windows, e.g. XP, are all “good things”.

Installing the RIS software is simple. Insert the RIS disk into the CD ROM drive and the installation should start immediately. At every opportunity click on the “OK” or “Next” button (unless you have a reason to specify something other than default). Besides the actual RIS software, DirectX and QuickTime may also be installed. If you are asked, install both. At the end of the installation, your computer will want to re-boot. Click “OK”. After the re-boot, RIS will be started. At this point, you will need to install batteries in the RCX and the IR tower.

If the installation does not immediately start when the RIS disk is inserted, you probably have autorun turned off. To start the installation manually, double click on the “my computer” icon on the desktop and then double click on the CD ROM drive. This should bring up a directory of the RIS disk. Find and double click on “setup” to start the installation. Then follow the instructions above.
When you start the RIS software, the screen on the right appears. Clicking on run causes the login in screen to appear (note, the introduction screen may not appear and you may be presented with the login screen immediately). If you have used the software before, your login name will appear in a list underneath the text input field. If so, double click on your name. If not, type your name into the text field and click on the “new user” button.
Do the training missions

Once you are logged in, the screen at left appears. You should explore a little by clicking on the tour button and then look at some of the features of the RCX by clicking on the settings button. After you are done exploring, click on the missions button and the mission command screen appears. Click on the training missions button and the training missions screen appears. Complete all the training missions which should take approximately two hours. You can stop at any time and the RIS software will remember where you stopped. Doing all the training missions is important. Later on, we will be creating and using programming models of the various Mindstorms components. The training missions provide a good introduction to each component and being very familiar
with these components aids in using them in programs. We will also be covering the topics contained in the training missions in
greater detail. The visual nature of RIS programming will give you a deeper understanding of programming that is directly applicable
to learning how to program in Java.
Build the Roverbot

The RIS software provides a variety of well-designed robots, examples at right, with clear directions on how to build them in the Constructipedia and pictures in the RIS software. The Roverbot, pictured at left, is one of these robots. It is engineered to be rugged; dropping off a table onto the floor usually results in no damage. It can support a variety of interchangeable sensor assemblies including a one-touch sensor assembly, a two-touch sensor assembly and a light sensor assembly. It also can be fitted with small wheels, large wheels, tracks or legs. It is easy to change gear ratios to make it slow and powerful or fast and weak. And it has a zero turning radius, making it useful for robotic navigation exercises. You should build the Roverbot just to practice building robots and to gain experience on what constitutes good construction instructions.

Once you have built a Roverbot, you can choose to build your own robot. The essential feature of the Roverbot that we will be using is that it has a zero turning radius, ZTR. That is, it steers by having one motor turn off while the other stays on, “turn” or having one motor reverse while the other remains going forward, “spin”. This is the method used by a tank or bulldozer. The other method of steering normally uses a rack and pinion. Cars use this method of steering. When we start using navigation, ZTR makes programming much simpler and is more typical of commercial robots.
Programming tasks

At this point, you should have a working robot. If you haven’t already, download the firmware and try running the built-in programs. If you don’t know how to run the programs, go back and do the training missions.

Before we start talking about programming concepts, you need to do some programming (yes, this is putting the cart before the horse). There are two programming tasks that follow, maze runner and line follower. They are both easy to describe. A maze runner navigates (goes through) a maze. A line follower follows a line on the floor. For a person, these tasks would be very easy. For a robot, you have to program them.

There is one very important thing to remember as you start programming your robot. This is supposed to be fun. Make sure that you smile and you laugh as your robot does really stupid things because you screwed up the program.
**Be the Robot**

Be the robot is a phrase I use to describe how to think about building algorithms for robots. The next sections lead you through creating algorithms to accomplish two traditional robot tasks. But thinking about these algorithms is much easier if you “be the robot”.

It’s been said many times in this book that programming is a form of technical writing. All writing is done by people who expect their writing to be read by someone. Otherwise, what is the point of writing? And the best writers know how their audience thinks and what words their audience uses. That way, the author can be sure that the thoughts that he wrote are the same thoughts that his audience has after reading what he wrote. Now, if programming is a form of writing and all writing is done with an intended audience in mind then who is the audience for the programs that you are going to write? Why, the robot of course. So, to test your algorithms before you turn them into programs you need to be the robot.

What does “be the robot” mean? It’s a silly, and in a group of people, very fun game. You become the robot. Your right leg is your robot’s right motor. Your left leg is the left motor. If your robot has touch sensors then your left hand is sensor 1 and your right is sensor 3. And so on. You have a group leader read your algorithm step by step and you, as the robot, follow each step exactly using only the commands available to program your robot (you can find the list of available commands in the RIS big block list). If, for example, your algorithm says turn right, you keep turning right until something in the algorithm stops you. If the algorithm says turn away from the wall, you can’t because you don’t have that command in your vocabulary. Your have to refine your algorithm so that turn away from the wall becomes turn right for .2 seconds. And so on.
One of the classic tasks of robots is to maneuver through a maze. A maze runner is a ZTR robot with two touch sensors mounted on the front, e.g. a Roverbot. Its mode of operation is to go forward until one of the sensors registers contact with a wall. It then backs up and turns away from the wall. It then proceeds forward until it hits the next wall. This is a very simple task to program in RIS, at least for first cut.

It is easy to build a maze. You can use bricks, boards, books, or anything else that is relatively flat and heavier than your robot. Arrange the objects in a pattern that has a space for the robot to go through. Then try having your robot run the maze with this program. If your robot does not make it through the maze, and with this program making it through the maze is highly unlikely, analyze why it does not. Some of the problems you will encounter have to do with calibrating the program, i.e. changing the times to better match the robot that you built. Other problems, e.g. getting stuck in a corner, have to do with this program not being very robust. The job of an engineer is to solve interesting problems. Solving problems requires thinking and perseverance.
Line follower is similar to maze runner. As the robot moves, a light sensor is pointed at the floor. Over dark places, the light sensor has a low reading. Over light places, the light sensor’s reading is high. What a line follower attempts to do is run along the edge of a line, the place where the black and the white come together. The sensor effectively averages the readings from the black and the white sides of the edge of the line. The objective of a line follower is to go forward while turning away from areas that are either too white or too black.

Putting this into concrete terms, assume that black has a sensor reading of 30 and white has a reading of 50. We want our robot to follow the boundary between the black and the white. Since the sensor averages the area that it is pointed at, we want to maintain the reading in the range of 35 to 45. So long as the reading is between 35 and 45, we want our robot to go forward. If the reading goes below 35, the robot is getting into a region that is too black and needs to turn away from the black. If the reading goes above 45, the robot needs to turn in the opposite direction.

The next page shows the panels that set up a light sensor. In the program above, there are two light sensor listeners. Note that both are set for sensor 2. One is triggered when the reading gets brighter than a threshold value; the other is triggered when the reading becomes darker than a different, lower value. Your robot goes forward when the value is in the middle and turns when the value is either too high or too low. Notice that we use the turn big block. What happens if we use spin instead? Is it better or worse for following the line?
You may need to calibrate your robot to get it to work. Get out the test pad from the RIS kit and see what values are best. Does this program do what you want or do need to change it? You can select the threshold values by clicking on the sensor. May sure that there is a space between the light and dark threshold values (this difference is called hysteresis). The panel on the right allows you specify these values. You can see the value of the sensor reading by viewing it on the LCD of the RCX after you select it using the view button of the RCX (the training missions covered this too).
What is a program?

The last two sections had you writing and debugging (fixing) programs for a robot. Now, it is time to start thinking about the nature of programs.

The old definition of a computer program is an ordered-list of computer-instructions. While this definition is less accurate than it used to be, it is still a good place to start.

You should be familiar with the concept of lists. Most people make up a list of items to buy before they go to the grocery store. This is an unordered-list because it doesn’t really matter which item you put in your cart first. On the other hand, a list of errands is usually an ordered-list because it is important that you “work out at gym” before you “buy ice cream to take home”. Doing those errands in reverse order would be bad, unless the temperature outside is below freezing.

Consider the code fragments on the left. They both have the same blocks. But they are not the same. Your robot will wind up at a different spot when you run one versus the other (try it). For computer programs, the order of instructions is important.

Computer-instructions are simply the specific things you want the computer to do in terms that the computer can understand. At right is a simple RIS program. When you download the program and run it, your robot goes forward for 1 second.
Two terms have just been used: code fragment and program. Both are ordered-lists of computer-instructions. The difference is that a program is a complete list while a fragment is not. You can download and run a program, you can’t run a fragment. Besides a list of instructions, a program needs a name. The program at right is name “untitled”. That is the name for programs before they are saved (you can, in fact, save a program with the name “untitled” but that would be a bad idea). In RIS code, as in Java, the name of a program is also the name of the file that the program is saved in.

**Algorithms**

An *algorithm* is simply detailed instructions to accomplish a task. Note that we did not say computer instructions. An algorithm is written in a natural language like English. To put that into context, having a robot go through a maze is a task. In the previous section, a program was presented that was intended to maneuver a robot through a maze. That program is the implementation of an algorithm. So, what is the algorithm?

The objective of the maze runner is to have the robot move from the start to the end of the maze. The quickest way from one point to another is a straight line so a key part of the algorithm is for the robot to continually move forward. However, simply going forward won’t work (or, it violates the rule of mazes; climbing over a fence is not allowed). Whenever the robot runs into a fence, it cannot go forward anymore. So it has to change its state, blocked by a fence, to a state where it can continue to move forward. The robot needs to back up and turn away from the fence so it is unblocked and then continue forward. The algorithm is to go forward when unblocked and to turn away from the blocking fence when blocked.

If you look at the RIS code for Maze Runner, you see that this is exactly what happens. The robot starts by going forward. When it touches a fence on the right side, it backs up, turns left (away from the fence), and goes forward. Likewise, when it touches a fence on the left side, it does a mirror image maneuver, backing up, turning right (away from the fence), and proceeding forward.
There are two important considerations in implementing algorithms in programs: robustness and calibration. Robustness is how well an algorithm works in different situations. Being more robust is, in general, better. Calibration deals with adapting an algorithm to the real world. In Maze Runner, the times are wrong. The time of 1.0 second for moving forward is probably too short while the backing up and turning times are probably too long. The word “probably” is used because, until you test, you cannot be sure. After you try running your program, it is quite likely that you will find that you need a better algorithm.

When we move on to programming in Java, Line Follower and Maze Runner are programs that you will write. The algorithms, at least at first, are the same for a program in Java and a program in RIS code. But you will find that it is easier to implement more complex algorithms in Java than it is in RIS code.
RIS uses color to distinguish between different types of program elements. Green blocks are used for regular instructions and blue blocks start programs or tasks (sub-programs). At left is an updated version of MazeRunner. This one includes purple blocks for conditional execution, if-then, and light green blocks for manipulating variables. This version of MazeRunner has an improved algorithm that deals with corner oscillation. RIS allows you to have several variables. Here I created 2, sideHit and sideCnt.

sideHit is used to keep track of which side was last touched. A negative number indicates that the sensor on 3 was last touched while a positive number indicates that sensor 1 was last touched. In Java, I would have used a boolean variable, a variable that can have only the values true or false. However, RIS code does not have boolean variables so we use what we have.
sideCnt is used to keep track of number of consecutive, alternating touches. That is, repeatedly touching on one side and then the other side. Put simply, if we continue to touch on alternate sides, we are probably trapped in a corner. If we are trapped, we need to something really different to get back to moving forward through the maze.

The purple blocks are if, or conditional, statements. Examine the code for sensor 3. The first block tests variable sideHit. If sideHit is positive, that means that sensor 3 was the last sensor pressed and that we are alternating pressing sensors. So, we add one to the alternating pressed counter and continue in the normal way. If however sideHit is negative then we have not had alternating touches. In this case, we set the counter back to zero and continue, knowing that we are not oscillating in a corner. The code for counting touches for sensor 1 mirrors the sensor 3 code just like the back up and turn away from the wall code for each sensor mirrors the other. Finally, the alternating pressed counter has a watcher. When the count gets up to five, it fires and the robot backs up and then moves randomly. After we have alternated five touches, we assume we are stuck. Hopefully, the random move will get us out of the corner.

The new algorithm builds on the original MazeRunner. We enhance the old algorithm by counting the alternative touches. If we have 5 alternative touches in a row, we assume we are trapped in a corner. When that happens, instead of backing up, turning a little and going forward, we back up and do something different before continuing going forward thereby hopefully breaking the oscillating cycle in the corner. This algorithm is more robust than the old algorithm. But it will need to be calibrated to determine good parameters for the “get out of corner” code fragment.

Experiment with this new MazeRunner, varying the back up time and spin time to see if you can make them work. Then, think about a better way of determining that you are stuck in a corner and how to get out. If you have a rotation sensor, think about how it might help you create a much more robust MazeRunner.
**Interrupts and polling**

At left is the maze runner program that was introduced in a previous section. It is an RIS program because it has a blue program block on the far left. It also has two additional blue blocks. These are sensor blocks; when the condition specified on the block is met the code fragment attached to it is executed. This is called *event-driven* programming and it is a very sophisticated programming technique.

Each blue block is a separate *task* and each task appears to run on a separate *thread*. A task is an independent sub-program. The blue block labeled “MazeRunner” is the main program and is the *default* task. That is, it runs when nothing else wants to run. Note that it is repeat forever or *infinite* loop. If it were not a loop, the robot would do nothing except respond to touches once it finished the main program’s instructions. The other two tasks are sensor listeners; when the condition is met, the code under the listener is executed. This is called *interrupt* or *event driven* programming. Main is the first task started when the program is run. When one of the sensors is pressed, the main task is interrupted and the code list attached to the sensor starts running. When the event code completes, the main task resumes running at the point where it was interrupted. Since main has repeat forever block, the robot always goes back to going forward.
As we said previously, the old fashioned definition of a program is an ordered-list of computer-instructions. The program at right is MazeRunnerP and is written in the old fashioned style of polling. In this style, all the code is in the main program. The program is continually asking, polling, the sensors if one has been pressed. If a sensor is pressed, the program does the same thing as the event-driven maze runner. If not, it goes forward for a very short period of time.

The fundamental difference between the two styles is where the locus of control resides. Locus of control is basically about who is in control of what happens next. Here, the program is in control. The logic that determines what happens next is inside of the program. In the event-driven maze runner, the locus of control is outside of the program; it is in the firmware of the RCX. Creating the sensor listeners tells the firmware that your program wants the specified code fragments to run whenever the event you specified occurs. The firmware takes care of asking the questions; you simply tell it what you want done when something happens.

Both polling and event-driven programming work reasonably well. The reason that event-driven programming is important is Java is an event-driven language. As it turns out, programs that use a windows user interface, e.g. Mozilla, MS Explorer, MS Word, JCreator, etc., use the event-driven style because it makes programming much easier. This is also referred to as object-oriented programming because events are tied to things or objects. In this case, the event, sensor pressed, is a thing, the sensor.
Setup Programming Environment

Overview of programming tools

Programming is an activity that requires a set of tools. Just like a carpenter needs a place to work in, tools to work with, lumber to work on and an idea of what he is going to do, a programmer needs exactly the same things, albeit in a different form.

The place where a programmer works is called an IDE or Integrated Development Environment. An IDE is like a workshop; it forces you to organize your programming. Typically an IDE contains an editor, a compiler, a debugger and a file manager. For Java, an IDE also typically includes a form designer for making graphical user interfaces, GUI’s, although the IDE we will be using does not; there is no GUI on a robot. It is possible to program in Java and LeJOS without using an IDE. It is also possible to build a house without power tools. But just because something is possible does not mean it is a good idea. A good IDE enhances your ability to get things done and the one that we will be using, JCreator LE, is both simple to get started with and powerful enough to get things done.

Programming, in its simplest form, is a four step process. The first step is a thinking step; you must create an algorithm to accomplish your task. For example, the algorithm for a simple maze runner has three steps:

1. go forward (called the default behavior)
2. back up when you run into a wall
3. turn away from the wall

That is all there is to an algorithm for a maze runner (notice that I said “an” algorithm not “the” algorithm because there are many maze runner algorithms that are better, more robust, than this simple one).
The second step is to convert the algorithm into Java source code. Source code is text, like a letter or an essay. In order to be Java source code, the format and key words have to conform to the Java Language Specification, which essentially defines the grammar of the Java language. Just like when you write something in English, you must capitalize names of people and indent paragraphs. Java has a similar set of rules and is very picky about people following the rules. At right is a simple program written in Java.

I assume that you are familiar with writing using a word processor. The IDE has a word processor like thing, called a source code editor, built-in. You use the editor to type in Java code that implements the algorithms that you have designed. In the “Hello World” example, above, you may notice that some words are in different colors. A good source code editor uses to highlight the meanings of different words. Think about how helpful it would be if your word processor displayed all the verbs in green and nouns in red. If it did, making grammatical errors would be much harder.

The third step is converting your source code from a human readable form to a computer readable form. The tool that does this is called a compiler. The compiler actually does two things: it checks that your program is correct according to the rules of Java and, if it is correct, creates an output file that contains byte code, the form of code that your computer or RCX can understand. Note that the compiler thinking your program is correct does not mean that your robot will do what you want it to do. It simply means that your code follows the rules of Java. Your algorithm could be bad, or not robust (work in a few situation but not many), or your Java code may not correctly describe your algorithm. In either case, you can have a program that runs but does not do what you want.

The fourth, and final, step is running your program. Within our IDE, we will be compiling to two different targets, our computer and our RCX. Usually, compilers generate programs for the machine that the compiler is running on, in our case the PC we
are using. The “Hello World” program, above, runs on the PC and in a few pages, we will compile and run that program. However, we want to use Java to program robots. The RCX is a different computer than the one we will be compiling its programs on. The reason for this is that the RCX is a very small computer, so small that it is impossible to run a Java compiler on it. So we use a PC to compile programs for the RCX and then download the compiled program and run it on the RCX. The process of compiling a program on one kind of computer, a PC, and running it on another, an RCX, is called cross-compiling.

When you have a program that compiles, you want to run it to see what it will do. This process is called linking and loading or downloading and it is separate tool in the IDE. The linking part combines our little program with other java executable code. In the “Hello World” example, the method println is called. This method takes a string, “Hello World”, and displays it on your computer in a special window. Somebody wrote Java code to do that displaying. Linking takes the byte code that they wrote and combines it with the byte code you produced by compiling your source code. Loading is what happens after your program is linked. Your program, and all the methods it uses, is loaded into your computer’s RAM and control is passed to your program. This is when your program does its thing. Downloading is similar. But instead of loading your program into RAM on the PC, it is loaded into the RAM on the RCX. Then, you start your program running by pressing the run button on the RCX.

Finally, the IDE has file manager tool. When you write source code, you want to be able to save it and re-use it. The file manager lets you have a bunch of programs, each in their own files, which you can work on. In JCreator, the hierarchy is workspace, project and file. We will create one workspace for all of your programming. Projects are used to group similar programs and file together. We will start with two projects. One project will be for programs targeted to the PC, like Hello World. The other project will be for programs targeted to the RCX, like maze runner. Files contain individual programs or parts of programs. Over time, we will create more projects and more programs within each project. Remember that there is a lot of space on a computer’s disk drive, so keeping old programs around is a good idea that costs very little.
**Obtaining the tools**

All of the software tools that you need are available on-line for free. This is the best way to get tools because it assures you that you will have the latest version. The following section details where to get them on-line. However, the tools are large files. For people without fast internet connections, I have created a CD that contains downloads of these tools.
JDK and JDK documentation

Go to java.sun.com. On the right, you will see a related links box (pictured at right). Select the J2SE link. This takes you to a screen that lists the available downloads. Scroll down on that screen and you will see these choices (pictured at left). You need to download both the J2SDK, Java 2 Software Development Kit (SDK), and the documentation by clicking on the hyperlink “Download”. You \textbf{do not} need to download the JRE (Java Runtime Environment). You will have to agree to license terms, which essentially state that you cannot sell the downloaded files or claim them as your own.

As part of the SDK download, you will have to choose (pictured below) which platform to download. Select the “Offline Installation”. At the picture indicates, the SDK is a large file, around 50 MB. On a cable or DSL line, it will several minutes to download; on a dialup line, it will take several hours to download.

For both downloads, you be asked which directory you want the files to be saved in. If you don’t already have one, create a new directory called “\downloads” and save all your downloaded files in it.
LeJOS and LeJOS documentation

Go to lejos.sourceforge.net. Click on the download button (pictured at left).

Download the current version, in this example it is 2.1.0, of the Win32 LeJOS (pictured at right) and the API docs. As with the JDK you should save both of these downloads in your download directory. These are zip, compressed, files that you are downloading. If you are using Windows XP or later, you can open this files as folders. If you are using an earlier version of Windows, you must use a zip utility, see below, to extract the files and create a usable folder / directory structure.

Download Area

**Linux/OSX/Solaris leJOS**
- Download version 2.1.0
- View Unix install Notes

**Win32 leJOS**
- Download version 2.1.0
- View Win32 Installation Notes
IDE – JCreator LE

Go to www.jcreator.com. On the JCreator site, the title frame at the top and the table of contents frame on the left are always visible (assuming that Xinox has not gotten a new webmaster). Click on the word “Download” in the table of contents. You should see the picture below. The LE version is free for most uses and is adequate for our purposes; you may wish to purchase the more advanced version. If there is more than one LE version available, as in this example, download the latest one, in this example v3.00. Once again, save the downloaded files in the downloads directory.
Zip files

Most of the files you have downloaded are compressed or zip files. In order to unzip the files, you need a zip utility. Windows XP has one built in so if you are using that operating system, you can skip this step. If you already have a zip utility, you can also skip this step. Otherwise, go to www.winzip.com and download the evaluation version of winzip. This will allow you to unzip these files and then you can decide whether or not to purchase the full product.
**Installing the tools**

Open an explorer window on your download directory or open the CD with setup files and open its download folder. Something like the window at left should appear (note, this is Windows XP; other versions of Windows will look slightly different). There should be at least these five files present. The two with names beginning with j2sdk are the Java distributions from Sun. The two starting with lejos are distributions that allow Java to run on the RCX. The fifth file, starting with jcrea, is the JCreator distribution that is the IDE we will be using.

Installing Java for the RCX requires several steps:

1. Install the j2sdk. To do this, double click on the program icon, next to the j2sdk…. This program will ask you a series of questions; answer yes to all of them. One of the questions will ask which directory to install the j2sdk into. Remember the name because that is where the j2sdk doc directory should go.
2. Install the j2sdk doc directory. The second file that begins with j2sdk is the documentation, or doc, file. As with most of the other files in this directory, it is a zip, or compressed, directory file. If you are running Windows XP or later, simply double-click on the file icon and then drag, or copy and paste, the docs directory into your j2sdk main directory (which you conveniently remember from step 1). If you are running an earlier version of windows, use winzip to extract all the files from the zip file. Again, you should extract the files into your j2sdk directory. Note, this is a large file and the uncompressing may take ½ hour or more.

3. Install the LeJOS sdk. Again, LeJOS comes in a zip file and you need to extract it. The main directory of LeJOS should go in the root directory, “c:\”, on your computer.

4. Install the LeJOS docs. LeJOS has docs files just like j2sdk. The docs come in a zip file. You should extract the docs to the root directory. When you do in Windows XP, you should get a warning message saying that you are about to copy into an existing directory. Click on the “yes to all” button.

5. Install JCreator. This is yet another zip file that needs to be extracted. However, this zip file contains a setup program. If you are not using Windows XP or later, you should extract the two files into a temporary directory, i.e. create a new folder in your download directory named “temp” and extract the two files into it. Then, click on the setup icon and this will lead you through installing JCreator. After you are done, you can delete the “temp” directory.

At this point, you are done with installing all the software needed to program RCX-based robots using Java. The next section will lead you through configuring JCreator to make it easy to get your programs compiled and running on your robots!
Configuring JCreator

JCreator is an IDE or integrated development environment. The purpose of an IDE is to make software development easier. For example, in order to compile a program in JCreator, you click on the build button. That’s all. If you did not have an IDE, you would have to open a DOS window, type in a cd command, e.g. “cd \lejos\src\test”, and then type the LeJOS command. As you can see, using the IDE makes it easier to program.

When you open JCreator for the first time, the accompanying screen appears. Notice that it has the standard menu items, File, Edit, Help, etc. and several tool bars that we use when programming.

The main part of the screen has five parts: along the top is the main menu and tool bars, upper left is the workspace panel, middle left is the package panel, bottom is the output panel and the large panel on the right is the editing panel. In this picture, all of them are blank.
The workspace panel shows the list of files that we are working with. The package panel is like a UML diagram, showing the methods of the classes we are working with. The editing panel is much like a word processor. This is where we actually type in our Java code. The output panel is where error messages from the compiler and messages from our programs are displayed. As we work through setting up JCreator, things will start appearing in these panels.

The installation program configured most of JCreator. At this point, you could write and compile Java programs to run on your computer. However, we need to do some additional configuration to be able to compile programs for robots. That is what the remainder of this chapter is about.
Setup LeJOS in JCreator.

By default, JCreator sets up the standard JDK. That allows you to compile and run programs by clicking on a button. However, JCreator does not know anything about LeJOS. So, we have to tell it.

LeJOS is like a second Java compiler. The standard JDK compiles and runs programs on your computer. LeJOS compiles programs for the RCX on your computer and then downloads the programs to the RCX so the programs can run there. This is called cross-compiling. The reason for cross-compiling is that the RCX is not a powerful enough computer to run a compiler, at least a Java compiler.

Cross-compiling is not something that JCreator normally does. However, it is a powerful enough IDE to be able to do it. We just have to configure JCreator so that it will know where LeJOS is. To begin, click on the configure menu item. A popup menu gives you the choice of options or customize.
Clicking on options brings up the options dialog box. The tree on the left shows all the option categories that you can set. You select a category by clicking on it. By default, the General category is selected. Categories with sub-categories are displayed preceded by a plus or minus sign. Clicking on the plus sign displays hidden sub-categories while clicking on a minus sign hides displayed sub-categories.

In the picture at right, the “JDK Tools” item has already been selected. In order to run LeJOS inside of JCreator, we need to create another JDK tool so that we can easily compile and download programs for the RCX. To do this, click on the copy button.
Setting up the compiler, lejosc.exe

When you click on copy, the dialog box, right, appears. To configure the LeJOS compiler you need to configure both the command and the parameters for the command. Together, the command and the parameters form the command line. This is what you would type in a DOS box if you weren’t using an IDE. Since you are using an IDE, you need to specify it once and then the IDE remembers it for you. Start by filling in a name for this configuration. In this example, it’s highlighted in blue and is simply lejos. The name can be anything and has no significance other than it will be the tool that you pick to compile your robot Java. Then use the file chooser button to specify where the LeJOS compiler is located. If you installed LeJOS in a directory called LeJOS on the C: drive in the root directory, as was recommended in an earlier section of this chapter, then the compiler will be “C:\lejos\bin\lejosc.exe” as shown in the example. If you put it somewhere else then you need to find it. If there are spaces in the path to the compiler then it is a good idea to surround with path with double quote marks, e.g. “C:\Program Files\LeJOS\bin\lejosc.exe”. Click the boxes as shown unless you want them to be different.

The command part of the command line is the executable DOS program that runs the compiler. In this example, the compiler, lejosc.exe, is located in the \lejos\bin directory. When you unzipped the LeJOS, your computer put the compiler in the bin directory. You select the command by clicking on the file chooser button and using the filechooser dialog to specify the full path to lejosc.exe.
In order to complete configuring the LeJOS compiler, you must switch between the command and parameters panes by clicking on the parameters tab. The parameters screen is pictured at left. You will need to type in the `-target "1.1"` exactly as it appears in the Parameters line. Clicking on the use output path and compile all files boxes will create the other parameters as displayed. That is, you do not need to type in anything other than the target parameter; clicking the checkboxes does the rest of the typing for you.

The output path parameter simply tells the compiler to put its output files, your byte code programs, in the directory that you specified when you set up your project. By default, most projects just store their byte code files in the same directory as their source code files. Usually, these is a good idea but as you gain experience you might want to do things differently. The compile all files command simply compiles all the files in a project. This can be a mixed blessing but it is usually the right thing to do. Be very careful to not check the use class-path parameter.

Note that lejosc.exe is not a standalone compiler. It uses the compiler in the JDK but supplies it with the LeJOS classpath. This is not important now but it may answer some questions you have later on.
Setting up the executor, lejos.exe

Just as you had to add a LeJOS option to the compiler, you also need to add a LeJOS option to the executor. The option screen is the same but you need to select the tool type “Run Application”.
When you copy the default and then edit it, panels similar to the compiler appear. The command to execute is lejos.exe and is found in the same place as lejosc.exe. The parameters are as shown on the parameters panel. Again, do not click on use class-path.

lejos.exe does not start the program running. You do that by pushing the run button on the RCX. What lejos.exe does do is download the program to the RCX. Thus, while lejosc.exe is just an extension to the programs in the JDK, lejos.exe is a complete replacement for java.exe, the JDK’s java virtual machine. Again, this probably does not mean much to you now, but in the future it will be important.
Adding additional tools

Go back to the options dialog box. In addition to the compiler and executor, we need to set up two more tools. Click on “Tools” in the box on the left. The tools dialog appears in the right part of the options dialog, as shown here. You need to add two tools to complete setting LeJOS in JCreator.

The first tool is lejosfirmmdl. This tool is used to download firmware to the RCX. Firmware is the operating system for the RCX just like Windows XP or Windows 98 is the operating system on the computer you are using. When you program in RIS code, you must download RIS firmware before you can download any programs. LeJOS works exactly the same, although the firmware, operating system, is different for LeJOS. That is, you can Java programs only when the LeJOS has been downloaded to your RCX and you can run RIS programs only when the RIS has been downloaded. The only real problem in switching between RIS and Java programs is that it takes about 3 minutes to download the firmware.
The second tool to set up is javadoc.exe. Javadoc is the documentation that comes with the JDK and LeJOS. It’s stored as web pages and you can read the documentation with any web browser. The javadoc tool allows you to make your own web page documentation. All of the sample programs for this text have javadoc comments included to help you understand what is going on. It’s a good habit for you to include javadoc in all the programs that you write.

To add the tools, click on new. The popup menu at left appears. You should click on Program. This brings up a file chooser dialog box. Look in your \lejos\bin directory. You should see the programs listed at left. Pick lejosfirmedl and click on Open. Repeat this process for javadoc by going to the JDK bin directory, selecting javadoc and clicking on Open.

After you have finished opening the tools, the tool box should look like this. The order of the tools is significant but not important (how’s that for a confusing statement). In JCreator, you can execute a tool by holding down the control key and pressing a number. In this example, ctl-1 runs lejosfirmedl and ctl-2 runs javadoc. Once you start programming, it is very convenient to use these shortcut keys. Just remember that if you use a different computer, check which tools are assigned to which shortcuts.
Set up lejosfirmdl.exe

lejosfirmdl is used to download the firmware to the RCX. Firmware is the operating system for the RCX, just like Windows is the operating system for most PCs. The firmware must be downloaded each time the RCX loses power, i.e. when the batteries are changed. For this program, you need to type in both the arguments and the initial directory. There is only one argument, -f, for this program. This argument tells lejosfirmdl to attempt to run in fast mode. Since downloading the firmware takes several minutes, doing it as fast as possible is a good thing. The initial directory is the bin directory of your LeJOS distribution, usually c:\lejos\bin.

The options selected than for lejos or lejosc. Capture output is not selected. This causes a console box to appear instead of the output going into JCreator’s output pane. The reason for this is that lejosfirmdl displays a percent completed as it proceeds through the download. The display is much better in a console than in the output (check capture output to see the difference). Close console on exit is not checked because if an error in the downloading process occurs, we want to see. If close console on exit is checked, the console is closed immediately whether the download ends normally or ends without successfully completing.
Set up javadoc

Javadoc is a tool that comes with the j2sdk distribution. It is used to help document all of the Java functions in all Java distributions. You can use it to document your programs as well. The arguments and initial directory for javadoc are the file name and the file’s directory.

The tool options are not important; capture output and show command line if you choose.

Javadoc itself is well beyond the scope of this document to describe. The example programs that you will see later all have javadoc embedded in them and the html pages that are created. Conceptually, javadoc is a compiler that produces a different kind of output file. Whereas a compiler usually converts source code into something a computer can understand, the javadoc compiler converts source code into something that programmers like you and me can understand.
Create new workspace for JDK.

To create a workspace, click on the File menu, then select New and Blank Workspace (as pictured at left). Enter a name, like robots, and click on Finish. Note that the Back and Next buttons do not work on this form. That is because creating a new workspace is a very simple operation and there are no other options that make sense.

The final steps in setting up JCreator are creating a workspace, a project and a sample program. A workspace is a place to hold all of your code. A project is place to store code for a specific program or related programs. For example, you can create a workspace for your programs having to do with robots. Within that workspace, you will have at least two projects. One project is for programs that run on the RCX and the other is for programs that run on your PC (later on, we will be writing programs that allow your PC to talk to your RCX for a variety of purposes). As your programs get more complex, you may well want to create a project for each program and not just dump everything into one project.
Create a new project

After you have created a workspace, you need to create a project. Click on the File menu, followed by New and Project. The picture at left will appear. Click on empty project and then click on the next button. The box, below, will appear. Enter a name in the name box, preferably without using spaces, and then click on the Finish button. This adds the newly created project to your workspace. Note that there are additional steps that we could do which are listed in the left part of the box. For the present, we will skip them.
Create a program to test that the installation worked. Click on File, New, File. The picture at left will appear. Select Java File and click on next. Type in the name HelloWorld, no spaces, and click on finish (note that the location displayed in the file wizard panel will vary depending upon where you set it up).
At this point, you need to type in a program. Hello World is the classic test program. It dates from Kernighan and Ritchie’s work on C in the late ‘60’s. The entire program is pictured at right. If you did it correctly, you should have a blank text pane where the Hello World program is sitting. Type in the program exactly as it is written.

Then click on the Build menu and compile the file (click on Build then Compile File). If you get no errors, execute the file (click on Build then Execute File). Your result should be exactly like what is in the general output pane.
Setting up an RCX project

At this point, you should create a second project, called rcx (see directions above). After you have created it, right click on rcx in the file view and a popup menu appears (see right). Click on properties and a properties dialog box appears.
Select a tool type of compiler and click on lejos so that its box is checked.

Change the tool type to run application and again check the lejos box.

Make sure that the output path on both the compiler and the run application are set to your rcx directory.
At this point, you are finished setting up the JCreator IDE. Before you start programming your RCX, you will have to do a couple more things, which are described below. Note that the run box has the name of a class in it. This is the name of the program that will download to your RCX when you click on execute project. As you add more programs to your project, more names will appear in this drop down box. Always pick the program you want to download to your RCX in this box before you click on the execute project button to download your program to the RCX.
Set up Environment Variables

There are two little things left to do before you can program your robots. First, we have to tell lejos where the Java compiler is located. Second, we have to tell lejos and lejosfirmdl which port the IR tower is connected to. We do both of these things through the use of environment variables. The way you set environment variables depends upon your operating system. In Windows XP and later, you set them through the system control panel. In earlier versions, you edit the \autoexec.bat file. Since this process requires going through many screens, we will just present a narrative here and put most of the screen shots in an appendix.
Windows XP and later

In XP, go to the start menu. Click on control panel and set it to classic view if you don’t already have it set that way. Click on the System icon. This will bring up a window labeled System Properties. Click on the advanced tab at the top. This will bring up the picture at right. Click on new in the top section, user variables, and enter values for path and rctxty. The value for path will be the directory name where you installed j2sdk (I told you to remember it) with \bin appended. If the variable path already exists, select it and then click edit instead of new. Then add a semicolon followed by the j2sdk directory name and the \bin. After you are done, click OK. The value for rctxty is the port your IR tower is attached to. For USB towers, the value is usb while for serial towers it is either com1 or com2.
Windows 98 and ME

Earlier versions of Windows do not allow you to set environment variables in the control panel. Instead, you must edit the file `\autoexec.bat`. The easiest way to edit `\autoexec.bat` is with notepad. Go to the start menu and under accessories you should find the application notepad. Start it and then click on File and Open. The screen at right will appear. Ignore the directory that is being displayed. Instead, in the File name box type `\autoexec.bat` and click on open.
You should see a file that looks similar to the one pictured at right. Add the two lines at the bottom of the file; the one labeled path and the one labeled rcxtty. Do not change anything else in this file unless you really know what you are doing. Make sure that the value you put in the path file corresponds to the directory name of your j2sdk bin directory. Make sure you have a ‘;’ between the %path% and your directory name. Also, make sure that the number on the j2sdk is correct. The 1.4.2_02 shown here is almost certainly wrong. The value for the rcxtty variable corresponds to where you have your IR tower plugged in. If you have a usb tower then the value is usb. If you have a serial tower, it will probably be com1 or com2. Make sure that it is correct.

Finally after you make these changes, save the file and close notepad, you must reboot your computer for the changes to take effect.
To start testing your robot, you first need to download the firmware by typing control 1 (ctl-1) in JCreator. After that is completed, it is time to write a program for your robot (It takes about 5 minutes to download the firmware).

Type the program at left into JCreator (or get it from the appendix). In order to do that, you need to create a file. Right click on your rcx project and then click on add. Choose add a new file. If it gives you a choice, click on java file. Then type in the name “Testbot”. Capitalization does not matter but the name must match exactly with the name on the class line (the second non-blank line in the example). Spaces are not allowed in the file name. Compile the program using legosc by clicking on the build menu item and then on compile file. Once you see “process complete”, download the program to your robot using legos by clicking on the build menu item and then on execute file. Press the run button on the RCX to start the program. If the robot does not start moving when you press the run button, you need to debug the program and the environment. Some common problems are:

- Firmware not loaded (run legosfirmedl, ctl-1).
- Robot not turned on.
- Interference on download (make sure IR tower is near RCX and protected from excess light (put it in a box)).

Once the robot starts moving, you have succeeded in installing and testing your LeJOS and JCreator environment. At this point, you are ready to start learning how to really program in Java.

```
import josx.platform.rcx.Motor;

class Testbot {
    private static void pause(int time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
        }
    }

    public static void main(String[] arg) {
        for (int idx = 0; idx < 5; idx++) {
            Motor.A.backward();
            Motor.C.forward();
            pause(1000);
            Motor.C.backward();
            Motor.A.forward();
            pause(1000);
        }
    }
}
```
Coding in Java vs. RIS

To some extent programming is programming, no matter which language you use. Java is a text-based, object-oriented language while RIS is a visual, procedural language, which makes the two very different. But they do have many similarities. This section displays RIS code, which you have learned in previous chapters, and Java code. Some of the Java code will be complete programs while other code will be code fragments. You should copy the Java code into your editor, compile it and download it to your robot.

The RIS IDE was designed to be a very easy to use tool. As happens in life, something that is really easy to use tends to be not very powerful. JCreator is a fairly powerful IDE and Java is an extremely powerful language. What that means is that you have a lot to learn if you want to program robots in Java. Using JCreator, or any other IDE, to write Java is a six step process:

1. Create a file to hold your program.
2. Enter your program text in the text editor.
3. Save your file.
4. Compile your program (translate from human readable to computer readable files).
5. Fix any errors (repeat step 4).
6. Download and run on your robot.

I assume that you have installed the JDK, the IDE and LeJOS per the instructions in a previous chapter. If you haven’t, go back and do it now. You learn how to program by programming, not by reading about programming. Some of this material may seem redundant, bear with me. I find it easier to have something repeated than to have to flip back and forth. So, let’s begin with step 1.
There are several examples that follow. Each requires a file in your RCX project. To create a file, right click on your rcx project. Then click add and then click new file. The screen at left shows these steps.

Finally, you should type in the file name in the name box on the file wizard panel. The name must correspond exactly, including upper and lower case, to the class name. The first example is the class “DoNothing”. The name of its file must be DoNothing.java. The IDE will add the “.java” for you. The location, by default, is your project directory. You can change this location but that is usually a bad idea for beginning programmers. Again, the file name must correspond exactly to the class name. The important point about this is you cannot have spaces in the file name. After you type in the file name, click finish and JCreator returns to displaying its text editing box. Once you have created the file, enter the example code and then save it.
After you have copied the example code into the appropriate file, it is time to compile and test it. To compile a file in JCreator, click on the build menu item and then compile file. In the output window at the bottom of the screen, you should get the message “process complete”. If not, you have probably made a typing error. Check carefully to make sure that your code is exactly like the examples and that your rcx project is set as the active project.

Once you have successfully compiled your code, you need to download it to your RCX for testing. Your rcx project is set up to do this by you giving JCreator the execute file command. Again, “process complete” should be displayed in the output window if all went well. There are several errors that are possible when you download a file. One is that you did not turn on the RCX. Another error is that you have not downloaded the LeJOS firmware to your RCX. These are the most common errors. If there are other RCXs around or you have bright lighting, you might need to stick the tower and RCX into a box and cover it. Finally, once you are successful in downloading the program, press the green run button on the RCX.
Examples

At left is a complete RIS program. At right is the equivalent Java program. Both programs can be downloaded to the RCX and both do nothing.

```java
import josx.platform.rcx.*;

class DoNothing {
    static void main(String[] args) {
    }
}
```

At left is a complete RIS program. Its name is Forward and it has two statements, a small block that sets the direction of the motors and a second block that turns on the A & C motors. The Java seems more complex. It starts with an import statement that makes RCX objects available to our program. The class statement defines a Forward object, whose “objectness” we ignore in this simple program. We define one method, main, which is our entry point for our program. main has a parameter, an array of Strings, which we also do not use. Again, you need to create a new file called “Forward”. Enter the code as shown, then compile and download it. Does it do the same thing as the RCX code?

```java
import josx.platform.rcx.*;

class Forward {
    static void main(String[] args) {
        Motor.A.forward();
        Motor.C.forward();
    }
}
Forward, the main program, at right, has one statement, a big block Forward. Forward starts both motors in the forward direction for one section. Big blocks are subroutines, the equivalent to a Java method. The Forward big block is expanded at far right. The algorithm for Forward has four steps: set the motors’ direction, turn the motors on, wait for one second and then turn the motors off. The Java code, below, does the same thing. The entry point is main and there is a single statement, goForward(1000). Java times are in milliseconds so 1000 milliseconds is one second. The method goForward has the same statements as the Forward big block. The Motor.A.forward method both sets the direction and turns the motor on. Thread.sleep suspends execution for the specified time just as the Wait For block does in the RIS code. Finally, the Motor.A.flt statement turns off the motors. Again, goForward is static; it can be used without an associated object.

Copy this program into your text editor and see what it does.

```java
import josx.platform.rcx.*;

class Forward {
    static void goForward(int time) {
        Motor.A.forward();
        Motor.C.forward();
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {} 
        Motor.A.flt();
        Motor.C.flt();
    }

    static void main(String[] args) {
        goForward(1000);
    }
}
```
Declaring Variables

Declaring and assigning values to variables in RIS is done through a set variable small block. But RIS, like Java, requires declaring variables before they can be used. In RIS, you pick a variable to assign a value to through the set window. You can pick an existing variable or create a new one and then assign a value to it. RIS has only one variable type, an integer that is displayed as a floating point number with only one decimal place and a range from –3276.8 to 3276.7.

Java has 8 different primitive types: boolean for true or false, 4 sizes of integers (byte, short, int and long), two sizes of floating point numbers (float and double) and char for storing letters. The biggest two, long and double, are not supported by LeJOS because there is no practical reason for such large numbers in a small robot. Variables should be declared as close as possible to their first use.

Variables exist only within the block where they are declared and within blocks contained within that block. The code fragment at left illustrates this principle.

```java
boolean flg = true;
int intVar = 0;
float floatVar = 0.0;
char charVar = 'a';

{  
  int idx = 0;
  if (idx == 0) {
    int jdx = 0;
    // idx & jdx exist
  }
  // idx exists
  // jdx does not exist
}
```
Conditional Execution

if statement

Conditional execution, the if statement, refers to executing code fragments depending upon the values of variables in the program. At right is the RIS code for conditional execution and the Java code is at left. In RIS code, if the condition is true, the code in the Yes path is executed, otherwise the code in the No path is executed. In Java, if the condition expression evaluates to a boolean true, code fragment 1 is executed, otherwise code fragment 2 is executed. In both languages you can nest if statements, that is, one of the code fragments can be another if statement. In RIS code, because of its visual nature, it is clear which yes and no fragments are in a given if statement; in Java it may not so clear because you are not required to have an else with each if as you are in RIS.

The rule in Java for matching else’s with if’s is simple; each else goes to closest preceding, unmatched if at the same block level. That is, if you enclose an if statement inside a block, place it between “{“ and “}”, then it doesn’t match with an else outside the block. The example at right demonstrates if’s and else’s matching. Notice that each if matches the first else that follows it except for the c4 if. That is because the c4 if enclosed within a block and there is no else within the block.
**switch statement**

Java has two conditional statements that RIS does not have: switch and try-catch. The switch statement appears at right. The statement works by evaluating the caseVar, which may be a simple variable or an expression, and goes to case label that has the literal value of the caseVar. For example, if caseVar had the value 1, code fragment 1 would be executed. The break statement that follows the code fragment causes execution to jump to the end of the switch statement. Without the break, execution continues through the rest of the cases until a break is encountered. The default is executed whenever no case label matches the caseVar.

```
switch (caseVar) {
    case 1:
        Code-fragment 1
        break;
    case 2:
        Code-fragment 2
        break;
    default:
        Code-fragment d
}
```

**try-catch-finally statement**

The try-catch-finally statement is a modern innovation. The block between the try and the catch are checked for runtime errors. In this case, an array of five ints is created. The program then tries to assign a value into the array position 6. Since not enough space has been allocated, this causes an error. The catch block is executed if and only if the specified error has occurred. A try can have multiple catches if each catch specifies a different type of error. User defined methods can specify different errors that they can throw and any method that specifies an error in its definition must be inside a try-catch statement. Finally, the finally clause of a try-catch-finally statement is optional. If a finally clause is given, its code is always executed, whether or not an error occurs.

```
try {
    Int[5] intArry;
    intArry[6] = 3;
} catch (Exception e) {
    // handle error code
} finally {
    // optional clause
}
```
**Loops**

The next section compares RIS code fragments to their equivalent Java code fragments. You can experiment with these fragments by creating the called method, e.g. `turnLeft`, in the Forward class file and replacing the `goForward(1000)` method call with the respective code fragment.

These fragments are flow-control statements. If-then, conditionals, allow the execution of a program to proceed along different paths. Loops are the second type of flow-control statements and there are three distinct types of loops: for, while and do until. The `repeat` statement in RIS translates into the three different types of loops.

**for loop**

The for loop is equivalent to the RIS `repeat` with a count in the bottom, which is pictured at left, and at right, the Java code is listed. In Java, the for statement has three parts that are contained within the parentheses and are separated by semicolons. The first part is the initialization part, the second is the test part and the third section is the increment part. The initialization part usually contains the declaration of one or more variables, the loop variables, and their initialization to 0 (note that this is the usual case and not, in any way, required). The test section compares the loop variable to some control value, in this case 3. The increment section changes the value of the loop variable after all the statements within the body of the loop have been executed. The for loop is the most complex of the three different loop types.
The while loop is equivalent to the RIS repeat with a while in the bottom. With a while loop, the condition is tested first. If the value is true then the body of the loop is executed. The body continues to be executed as long as the condition remains. When it becomes false, the next statement after the body of the loop is executed. Note that the body of a while loop may, or may not, be executed depending upon whether or not the condition evaluates to true when the while is executed the first time.

The do until loop is equivalent to the RIS repeat with an until in the bottom. With a do until loop, the condition is not tested first; the body of the loop is always executed at least once.
repeat forever

The repeat forever loop in RIS does not have a unique Java equivalent; it is just a special case of a while loop. In this case, the logical literal true replaces a condition. This is called an infinite loop because it never exits normally; there must be some kind of test inside the loop that interrupts the normal flow of the program.

break, continue, return & throw

There are four statements that alter the flow inside of a loop: break, continue, return and throw. break terminates the loop and starts executing the statement immediately after the body of the loop. continue skips to the end of the body of the loop and performs the loop test. Note that in a for loop, the increment part of the for statement is executed before the test is performed. return acts as a return does anywhere else; it exits the current method and continues execution in the method that called the method that contains the loop. Finally, throw throws an exception that must the caught in the catch part of a try-catch statement.
Exercises

1. Write a maze runner program in RIS using sensor listeners. Repeat using polling. Which is better and why?
2. The three different loop types, for, while and do until, can all be used to do the same things. Using a for statement, write Java code that is equivalent to a while and a do until. Do the same for the other cases. You should wind up with six different code fragments.
Introduction to Objects

This chapter is an introduction to objects. What is an object? Simply, it is any thing. That is, if you can name it, it is an object. Car is an object. House is an object. Love is an object. Anything with a name is an object. This is a very profound concept. Until you understand this concept, do not try to learn Java. A specific thing is an instance of an object. My car is an instance of the object Car. An instance can be derived from several objects. I am an instance of the object Person, the object Man and the object Author. The relationships between objects and instances will be discussed in great length later on.

Introduction – programming as model of “real world”

Programming has always attempted to model the “real world”. The first application of a computer was to calculate artillery tables. The question was, at what angle do you set a cannon in order for a shell to land at a specific distance? The computer was programmed to create a table that had the firing angle needed for a list of distances for a specific cannon with a specified shell and powder charge. This was one of the first computer models. It is a model because soldiers or sailors could have gone out and fired real cannons. They would have had to carefully measure the angles the cannons were fired at and the distances the shells traveled. They would have had to fire a lot of expensive shells and hoped that the weather and wind remained constant. And whenever the gunpowder got a little stronger or the shells a little heavier, the whole, expensive process would have to be repeated. As has happened many times, military programs have paid for technology development that has subsequently resulted in tremendous civilian products.

Today, computer models are everywhere. A word processor is a model of pen and paper. An inventory control system models a warehouse. Even the programs that you will write to control a robot are models. Explaining it now is hard; very soon you will have enough experience to see that the concept of computer programs as models is obvious.
Relationship of programming objects to physical objects

Object-oriented programming creates computer objects to represent real objects. What that means is simple. When you look at the world, you see objects. Everything is an object of some sort. Trees are objects. Books are objects. Thoughts are objects. Anything that can be named is an object. OOP recognizes the object nature of things and uses that nature as an organizing mechanism. Objects can be alike or different. Two spoons share the same characteristics. A spoon and a fork have a lot in common but are distinctly different. A car and a pig are not alike at all. When we program a computer, we look at the characteristics of the things we want to model. Then we try to impose some sort of order to make our programs better.

Most of you have studied your natural or first language. Since I am familiar with English class, I’ll describe what I learned there and hope that you learned the same things. The study of English starts with the concepts of parts of speech. The parts of speech include nouns, verbs, adjectives, and proper nouns, among other things. In programming terms, a noun is a class and a proper noun is an instance. That is, a class describes the characteristics of a generic object. An instance describes a specific object. Adjectives help describe a specific object and are, in programming terms, properties. Verbs are action words and are called methods.

For example, dog is a class. Shelby, my dog, is an instance. Weight is a property of dog and in Shelby’s case the value of weight is 42 lbs (18 kg). Feed is a method of dog; one that Shelby really enjoys being used. Shelby is a Basset hound. Basset hound is a derived class of dog. Basset hound inherits all the properties and methods of dog. But by being a derived class Basset hound extends dog by adding additional properties and methods, by creating new constraints on existing properties or by changing what the existing methods do. Shelby, like most Bassets, likes to eat a lot. Obesity is a real problem with Bassets. So, the Basset hound class may override the feed method of dog with its own feed method. The Basset hound feed method may limit the amount of food that the instance can receive at any time.
Sometimes I want to work with a group of dogs. I have had three dogs in households over the years, Blacky, a mostly German Shepherd mixed breed, Ginger, a Dachshund, and Shelby, a Basset hound. All three are dogs, so they all have the feed method and the weight property that comes from being a dog. But they are also all instances of their respective breeds. Shelby is a Basset hound but she is also a dog because Basset hound extends dog. When I go through my list of dogs and feed each one, Shelby will be fed using the limit imposed because she is a Basset hound. Ginger and Blacky will be fed using the generic dog method unless, of course, their respective breed overrode the feed method. The ability to deal with instances through their generic class, dog, and their specific class, Basset hound, is called *polymorphism*. As you will see later, this is a very useful ability.

There are two kinds of properties and methods: *class* and *instance*. Class properties and methods are global while instance properties and methods apply to a specific instance. The feed method is an instance method. It is called to feed one specific dog. Likewise, weight is an instance property; it is the weight of a particular dog. An example of a class property would be maximum weight. This property does not apply to a particular instance; it limits the weight of all dogs.

Properties can be simple things, like weight or name which are numbers or strings, or more complex things, like feeding schedule or nutrition requirements. These more complex things are objects in their own right. Creating objects that have complex properties is called *composition*. 
Java’s concept of an object - java.lang.Object

At left is a UML (Universal Modeling Language) diagram. UML is a visual language. That is, it uses text in diagrams to convey more information than text alone could convey. UML is a fundamental tool of computer science education and has radically transformed the way computer science is taught since 1990. UML is specific to object-oriented programming. This includes the languages of Smalltalk, C++, C# and, most prominently, Java. UML is not a programming environment or a programming language. UML is a conceptual and visual tool. UML includes a variety of diagrams that cover different aspects of programming. These include class, use case, state, activity, and implementation diagrams, among others. Covering these topics in even a very minimal way is beyond the scope of this work. We will use only the UML class diagram. However, you are well advised to understand that class diagrams are only a very small part of a very rich programming methodology.

A class diagram is a way of visually organizing information about an object. It has three parts: a name, a list of properties or attributes and a list of methods or operations. The name is usually the same as the physical thing the programming object represents. In the previous section, we discussed the class dog and “Dog” would be an appropriate name for the programming object. Properties, Java term, or attributes, UML term, are characteristics of an instance of the object, the “blanks” that you need to “fill-in” to describe a specific instance. For example, dog would have attributes of weight, age and color that would need to be filled in for a specific dog. Methods, Java term, or operations, UML term, are things that an object can do or have done to it. Dog would have a feed method that would set the happy property if feed were called often and unhappy if feed were called infrequently.
In the previous section, we discussed derived classes. BassetHound and GermanShepard were derived classes of Dog. An instance of BassetHound has all the properties and methods of Dog plus a few that are specific to Basset hounds. That raises the question, is Dog derived from another class? In biological terms, the answer is yes. Dogs are mammals and mammals are invertebrates and invertebrates are living things, etc. So we could create a hierarchy that has these classes and more. However, in programming, we model only things that we decide are relevant to the problem at hand. So, we can decide that a Mammals class is beyond the scope of our model. So we can safely omit that complication. However in programming terms, all things are objects so, in Java, all objects are derived from Object, either directly or, as in the case of BassetHound and Dog, indirectly.

The UML diagram shown is the class diagram for the Java class Object. Object is the base class for all classes in Java. That is, every object in Java has these properties and methods. So what properties and methods did the designers of Java believe to be so fundamental that they are in every class?

As it turns out, they found that there was no property that was so fundamental that it was needed for every object. Remember that properties are things that describe a specific instance of an object, or in rare cases, describe the object itself.

They did decide that 12 methods are so fundamental that they were required for all objects. The diagram lists them in the order that they appear in the JDK source code; we will describe them in functional groups¹.

The methods getClass, toString and hashCode are used to identify a specific instance of an object. getClass returns the class of an instance. toString takes the contents of an instance and converts it to a string that in some way identifies the instance. hashCode

¹ We skip describing the registerNatives method. The minus (-) sign preceding its name indicates that it is a private method. That means that it is unavailable outside of object itself. It is not documented in either the source code or the JLS.
returns a number, the hash code, that identifies, but not uniquely, the instance. Hash codes are used extensively to make programs run faster. getClass is a method that should never be overridden. toString, on the other hand, is frequently overridden to provide more meaningful information about an object. hashCode is in the middle, it should be overridden only by programmers who understand how to create a good hash method and who have determined that they need to override the default.

The methods clone, equals and finalize are used to manipulate instances of objects. clone is used to make a copy of an instance. equals is used to determine if two instances are equal. finalize is called by the JVM when an instance has been thrown away. All three methods should be overridden when needed. For example, clone is used to make a copy. But what a copy is depends upon the object. You could, for example, create an object Pack that contains a collection of instances of Dog. What does it mean to clone an instance of Pack? Does cloning a pack clone the dogs in that pack or not? That is, do you wind up with two packs with the same dogs in each or do you wind up with two packs, each with its own set of dogs? If you intend to use copy instances, you should probably override the clone method. Likewise, what equal means depends a lot upon the object. In the real world, twenty pounds of all-purpose flour is probably equal to any other 20 pounds of all-purpose flour. But any 20 karats of diamonds is probably not equal to any other 20 karats of diamonds. In the Dog class, equal could mean that it is the same dog or it could mean two dogs are the same age or weight or breed. If you use the equals method, you almost certainly want to override it. finalize, on the other hand, is hardly ever overridden. Try and come up with an example of when you might need to.

The last five methods, notify, notifyAll and the three waits are only used in multi-threaded situations. notify and notifyAll are used to tell a thread that an instance is available while the waits are used to suspend a thread until an instance is available. The three wait methods differ only in that one is a wait forever while the other two provide two different ways to specify a maximum time to wait. What, by the way, is a thread? Threads are independent execution paths. Remember in the RIS programming where you set up sensor watchers. That is, you had a block that said wait for touch sensor 1 to be pressed and then do this code fragment and another
block that said wait for touch sensor 3 and it had another code fragment. Each sensor watcher is a thread. The thread runs whenever its sensor is touched. Each sensor watcher is waiting for its signal. Effectively, each thread has called the wait method on their respective sensors. But there is a third thread that you don’t see and it, essentially, owns both instances of the sensors. What it does is constantly check to see if a sensor has been touched. When one is touched, it calls the notify or notifyAll method of the sensor and the sensor watcher executes. There are many details that go along with this that will discussed in later sections. Note that these methods are usually not overridden.

Before leaving this introduction to the UML class diagram, there are a couple of details that should be clarified. Note that all the methods in Object start with a “+” sign. A plus sign indicates a public method or property. A public method or property is one that can be used by any other object. A “-“ sign indicates a private method or property. Private methods and properties can only be used by the class itself; even derived classes cannot use private methods or properties. A “#”, sharp, sign indicates a protected property or method. Protected properties and methods are halfway between public and private; they can be used and overridden in derived classes but not in other classes. UML uses an underline (dog) to indicate class, as opposed to instance, properties and methods. Object has none but they will appear in later class diagrams.
Examples of Lego motor as object

A Lego Mindstorms kits contains over 700 individual pieces or physical objects. Two of the pieces are motors. You used motors on the Roverbot and programmed it to move around. What are the attributes and methods of a motor?

Two obvious methods are turn on and turn off. Equally obvious are methods to tell the motor to run forward or backward. As for attributes, being able to tell which power source the motor is connected to also important. What other attributes and methods would be useful? The next section contains the object model for motor from the LeJOS environment. Before reviewing it, write up a list and see how your list compares to the creators of LeJOS.

The diagram to the left is a UML class diagram. This is how the authors of LeJOS decided to describe a Lego motor. The diagram is divided into three parts: the top cell is the name, the middle cell is a list of properties and the bottom cell is a list of methods. The plus, “+”, sign indicates a publicly accessible property or method, one that you can use in programming, while a minus, “-“, sign indicates a private method, something that you are not allowed to use. Underlining indicates that an entry is static or applies to the class. No underline indicates an instance method or property.

Looking at the list of methods, there are six basic commands that should be on your list: setPower, forward, backward, reverseDirection, stop and flt (float – remove power but do not set the brake). There are five boolean methods: isForward, isBackward, isMoving, isFloating and isStopped. Booleans return simply true or false. So these methods ask about the state of the motor. Two other methods, getId and getPower, ask about properties of the motor. Note that these 13
methods are instance methods. They ask about or set properties of a specific motor. Notice the naming convention used. Action verbs are used for commands while these verbs are combined with is or get to form questions that inquire about the state of the motor.

That leaves one method, Motor. Motor is special. Notice that its name is the same as the class. This method is called the constructor. The constructor is the method called when you create a new instance of a class. Typically, a constructor will set up everything so that you can use an instance once you create it. This constructor is very unusual because it is private. That means that you can only create an instance from inside the class. How is this done? It is done through either a class method or class property initialization. If you look into the properties section of the diagram you can see that there are three instances of motor, A, B and C. Since an RCX has three and only three motor outputs, Motor sets them up from the beginning. There is never a need to create another motor object; doing so only wastes memory in the RCX.

Finally, the middle box is the properties of motor. The three motors are static and public so that they can be used anywhere in the program. The three remaining properties are Id, mode and power. Id is simply the name, A, B or C of the instance. Power contains the power level of the motor while mode indicates the state of the motor, forward, reverse, stopped or floating. These three properties are all private. The reason for being private is straightforward. Being public means that your program could get or set these properties. For Motor, getting them poses no problem. Each of the get methods simply returns the value of the property. But setting these properties poses a real problem. Suppose a program could set the Id property and proceeded to set the Id on all three motors to “D”. This would be bad. And so, to make bad things harder to do, most programmers make properties private unless there is a good reason not to.
Writing and programming

Writing a program is similar to many other kinds of writing. The purpose of any kind of writing is to take your thoughts and let other people see them. And you have to follow the rules of writing for your paper to mean anything. For example, when you write a term paper, you have to follow rules for punctuation, paragraphs and footnotes. If you don’t follow the rules, the teacher may reject your paper.

Before we start describing program writing, we take a side trip. Poetry is a form of writing that tends to have more rules than prose or narratives. One form of poetry that most school children learn is Haiku, an ancient Japanese poetry style. The rules for Haiku are very simple but very strict. A Haiku poem is always three lines long and the lines have, in order, 5, 7 and 5 syllables. Take a moment and write some Haiku poems about various parts of robots. Here are a couple of examples:

<table>
<thead>
<tr>
<th>Little motor runs</th>
<th>Sensor on my bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward, forward, on and off</td>
<td>How I wonder what thou art</td>
</tr>
<tr>
<td>Makes my robot move</td>
<td>Touch, light, infrared?</td>
</tr>
</tbody>
</table>

As you can see, the poems are not literary gems. But they do convey an idea of the nature of motors and sensors. Obviously, a computer is not going to accept a Haiku poem as a program. But the process of writing a Haiku poem and a program are similar. You express your ideas in a very structured writing style. As a further exercise, write a limerick about robots.²

The following sections and chapters introduce you to the tools you will be using. There are two distinct things you will have to learn in order to be a Java programmer: development tools and the Java language. The next sections are mostly about the tools. You will be asked to enter code into the IDE editor, run the compiler and look at the output. The hope is that in a very short time, using the

---
² Limericks are traditional Irish poems. They consist of five lines, with the first, second and fifth lines rhyming and the third and fourth lines rhyming. Limericks also tend to be somewhat baudy.
tools will become automatic. When you start your own programs, you want to concentrate on your code and not be worried about how to invoke the compiler. Writers need to concentrate on what they are trying to say, not on how to use a word processor or whether they need to use a colon or semi-colon. As a programmer, you need to be so comfortable with your development environment that you don’t consciously think about compiling code, you just do it.
Relationship of Class to Object

We’ve talked about objects. Objects are computer models of things. The objects that we are creating are computer models of are pieces of plastic made by Lego™. We will describe them using the Java language.

As mentioned before, Java is a computer language that comes in two forms: source code and byte code. Source code is easy for people to understand while byte code is easy for the computer to understand (easy being a relative term here). Source code is stored in text files. Text is simply all the characters that appear on a keyboard. It does not have formatting, like bold or italics, and it does not have different fonts or font sizes. It is a very simple kind of file.

The basic format of a Java file is simple. At the root level, only four kinds of statements are allowed: package, import, class and interface. An example is at left.

Briefly, the package statement identifies the group of things that an individual object belongs to. For example, Motor and Sensor belong to the josx.platform.rcx package because motors and sensors are used to build RCX-based robots. For simple programs, no package statement is included. This results in the object being in the default package. This is OK for small programs but not for classes that you intend to use in other programs, e.g. the objects associated with the RCX.

The import statement allows a class to use other classes that are not in the same package. The example specifies an individual object, Motor, to be imported. Alternatively, replacing Motor with *, e.g. josx.platform.rcx.*, includes all the objects in the package.

The class and interface statements are different than the package and import statements. Note that the don’t end with a semi-colon, “;”. Instead they are composed of three parts: the keyword, either class or interface, a name, in this case TestBot or MySensor and a block which is delimited by matching curly-braces, “{" and “}”. Usually you only one class or interface, but not both, statement

package MyRobot;
import josx.platform.rcx.Motor;
class TestBot {}       // either this
interface MySensor {}  // or this
per file (later on we will see more than one class per file). The name of the class or interface must exactly match the name of file less the .java suffix. That is, the class TestBot must be in a file named TestBot.java and capitalization does matter. Everything else inside the java source must be in a block associated with a class or interface.
Exercises

1. Create a class diagram for sensors, the buttons, and the LCD panel. Create a class diagram for a mobile robot, including both fields and methods. List both methods and fields for each object. Explain why you included some things and left out others.

2. Create a class hierarchy for some set of objects. For example, living things -> plants -> trees -> evergreens. Make it at least four layers deep. At each layer, state one new method or field that is added.

3. Explain in your own words the difference between a class and an instance.

4. Write 3 haikus or limericks that deal with robots. Alternatively, write one algorithm in iambic pentameter (if you don’t know what iambic pentameter means look it up).

5. Compare and contrast programming with creative writing.
Introduction to Java: ZTRBot

Now that you are familiar with programming your robot in RIS and have completed some tasks, it’s time to start programming in Java. The plan here is simple: first build the motor controlling class and have the robot move around. Then add sensors so that the robot can react to its environment. If you recall the definition of a robot, these are the first two attributes that define what a robot is.

The first step is to make a Java file that models the Roverbot that you built and programmed with RIS. Right? Wrong! Instead we start with a class called Util, which stands for utility methods class. There are several reasons for doing this. First, you need to get used to the idea of “code once, use many”. Java is based upon the idea of re-using classes instead of writing the same thing many times. Util is a class that you will use everywhere. Second, Java programs are made up of many, separately compiled classes. The idea is to get used combining many different classes to make one program. Third, Util violates the concept of classes representing objects. There really isn’t any conceptual Util thing, just like there is no Math thing even though the Math class contains a lot of useful math methods. The point here is to get you to think about the rules of Java and why there might be exceptions. These points will become clearer after you work with this a bit.

Util.java

Util.java is where we put utility methods. Utility methods are used in many other classes to do, usually, simple functions. We start with a single method, pause, in the class and we may add more as time goes on. pause converts computer-time into real-time by stopping execution for a specified amount of time measured in milliseconds. For example, Util.pause(1000);, stops the program for 1 second. We will see what this means in just a moment.

```java
package MyRobot;

public class Util {
    public static void pause (int tim) {
        try {
            Thread.sleep(tim);
        } catch (Exception e) {
        }
    }
}
```
To create Util.java, right click on your rcx project. Then click add and then click new file. The screen at left shows these steps.

Finally, you should type in the file name in the name box. The location, by default, is your project directory. You can change this location but that is usually a bad idea for beginning programmers. The file name must correspond exactly to the class name. The important point about this is you cannot have spaces in the file name. After you type in the file name, click finish and JCreator returns to displaying its text editing box. Type in the code on the previous page, Util.java, and then save it.
Once you have created `Util` with the `pause` method, it is time to write test code for the class. The easiest way to do this is to create an entry point, a main method, and call the methods that you have created. Since we have created only one method in this class, `pause`, the test code is straightforward and shown at right. Note that this is not the only test code that can be written nor is it necessarily the best test code that could be written. It is simply good enough test code for our purposes.

The first line inside `main` declares an int called “cnt”. We use this as a counter variable, i.e. it is first assigned a value of 0, then 1, then 2, etc. The next line is a while statement. while statements have three parts: the word “while”, a logical expression, and a statement to execute if the condition evaluates to true. In this case, the condition is a literal true, so the while’s statement will execute forever. The while’s statement can be either a simple statement, a line ending with a semicolon, or a compound statement, a block containing several statements and delimited with braces. In this case, the while has a compound statement composed of three simple statements. The first statement displays the value of the counter on the RCX’s LCD. You can check out more information on the LCD class by looking in the Javadoc. Next is a call to the `beep` method in the Sound class. Again, you can find more information about this class in the Javadoc. Note that using visual or audio prompts is an important technique in debugging programs, especially in cases such as this where you cannot use a real debugger. The next statement increments, adds one to, the value of the counter. The `cnt` variable starts with a value of 0, so after the first increment it has a value of 1, after the second increment it has a value of 2, etc. Finally, the `pause` method is called with a value of 1000. This causes the RCX’s computer to wait for 1000 milliseconds or 1 second. What this test program does is to turn your RCX into a timer, albeit not a very good one.
Now that you have copied the test code into your Util class, it is time to compile and test it. To compile a file in JCreator, click on the build menu item and then compile file. In the output window at the bottom of the screen, you should get the message “process complete”. If not, you have probably made a typing error. Check carefully to make sure that your code is exactly like the examples and that your rcx project is set as the active project.

Once you have successfully compiled your code, you need to download it to your RCX for testing. Your rcx project is set up to do this by you giving JCreator the execute file command. Again, “process complete” should be displayed in the output window if all went well. There are several errors that are possible when you download a file. One is that you did not turn on the RCX. Another error is that you have not downloaded the LeJOS firmware to your RCX. These are the most common errors. If there are other RCXs around or you have bright lighting, you might need to stick the tower and RCX into a box and cover it. Finally, once you are successful in downloading the program, press the green run button on the RCX.

After you have checked and are sure your pause method is working, it is probably best to remove or comment out (using the “/**” and “*/” comment delimiters) your main method in order to conserve RAM on your RCX.
ZTRBot.java

ZTRBot is the basic class for the rest of the book. The name refers to the fact that we will be using a zero turning radius, ZTR, robot for all the programming exercises. The definition of ZTR is a robot that can turn within its own radius. Examples of ZTR vehicles are tanks, bulldozers and self-propelled cranes. Most vehicles, including cars and trucks, are not ZTR. The reason for using ZTR robots is that steering is much simpler, hence the programming is also much simpler.

package MyRobot;
import josx.platform.rcx.Motor;
class ZTRBot {
    public static void main(String[] args) {
    }
}

Before we start to actually code ZTRBot, we need to format the source file. First, create a new file in your RCX project named ZTRBot. Then type in the skeleton file as shown at right. The first line is a package statement. Just like Util, all of this code refers to MyRobot. The next line (blank lines don’t count; they just make code easier to read) is an import statement. ZTRBot is going to want to use the Motor that LeJOS defined. Import is a convenience statement; it allows us to refer to the Motor class with just the word Motor instead of its full name, josx.platform.rcx.Motor. These first two statements are simple statements; each contains one command and ends with a semicolon.

The class has three parts: the word “class”, the name of the class, in this case ZTRBot which must be the same as the name of the file and cannot have spaces or other special characters in it, and the block that defines the class, which is enclosed, delimited, by a starting curly brace and ended with a closing curly brace. All code for a class must be inside the class’s block. A block is a compound statement; it contains other statements both simple, ending with a semicolon, and compound, enclosed with curly braces.

A class block contains two kinds of declarations: methods and fields. As we said before, methods are verbs and fields are nouns or adjectives. Recall that classes are templates, e.g. dog, while instances are things of the class, e.g. my dog Shelby.
We are going to take an incremental approach to writing ZTRBot. That is, we will write some code, compile it, download it to the RCX and run it there. We will then add some functionality and repeat the process. The rationale for this approach is that it is easier to find errors, bugs, in the new pieces that you add, if you know that most of your program is already runs. To start, we have only one method, main. This method is special; it is the entry point for a class. If a class has a main method defined this way then it is a program and can be executed; if not, it is just a class and must be used by other classes that are programs.

Copy the code shown into ZTRBot.java in the IDE text editor and compile it. You do this by first making sure that your RCX project is the active one. Your package view in the IDE should look something like this with RCX in bold. Note that there are two files in the RCX project, Util.java and ZTRBot.java. If RCX is not bold, right click on it and set it to be the active project.

When RCX is the active project, right click on it and then click on the properties menu item. The box at left should appear. The run combo box shows all the classes in the project that are programs. At present, only ZTRBot is a program so it should appear in the box. This is the program that will download to the RCX when you run it.

Finally, to compile the program, click on the build menu item and then click on compile project. This will check to see which files in the project need to be compiled and then compile them. Messages from the compilation will appear in the output window as we saw before when doing the test compiles. When you get a clean compile, you can download your program to the RCX by clicking on execute project. This downloads the file that you specified in project properties as well as all classes.
that it needs to run. Alternatively, compile file compiles the currently selected file, the file you are editing in the text editor, and execute file downloads the current file and its supporting classes. The advantage to using compile project is it ensures that all the files in your project are up to date. The disadvantage is that it requires all the files in your project to be up to date. At this point, you should compile and execute ZTRBot. It should compile and download cleanly. However, when you run it on the RCX, it should do nothing since that is what the program does at this point.
The next step is to determine what things we want our robot to do. An easy place to turn to for inspiration is the RIS. In RIS code, the Roverbot is a ZTR robot. If you look in the Roverbot section, you will find six simple movement big blocks: forward, backward, turn right, turn left, spin right and spin left. The forward big block is shown, expanded at right.

The challenge that we face is to translate RIS code into Java, making changes as needed. There are two steps to this process: convert the RIS code into an algorithm and then translate the algorithm into Java. The algorithm in this case is fairly simple. Turn motors A and C on in the forward direction, wait for a while, in this case 1 second, and then turn both motors off.

In Java, we now need to create a method that implements the forward algorithm. As it turns out, the example code from the Javadoc for Motor, shown at left, contains the code we need, albeit in with a few bells and whistles that we don’t need. The code from the example sets the power level for each motor; since we will not be changing the power level frequently these commands are not needed in our basic forward method. The forward commands will be needed in our forward method so they transfer directly. The sleep command tells the computer to wait 1000 milliseconds, 1 second, so that the motors can run for that long. We need to change that to a variable so that we can change the amount of time to actually go forward. And, we will change it to the pause method that we created in the Util class. Finally, the stop commands will be replaced with flt commands so that it corresponds to the RIS code. Recall that stop applies brakes while coast simply removes power (think about riding a bike to understand the difference).

```java
Motor.A.setPower(1);
Motor.C.setPower(7);
Motor.A.forward();
Motor.C.forward();
Thread.sleep(1000);
Motor.A.stop();
Motor.C.stop();
```
The final result of this is the updated version of ZTRBot shown at right. We have added a method called goForward. This method makes the robot go forward. The convention for naming methods is to string words together that describe what the method does with the first word all in lower case and subsequent words having their first letter capitalized. Classes have the same naming convention except that all words have their first letter capitalized. When defining a method, modifiers come before the name, in this case public, static and void. public means that we can use the method in other classes. Remember that we declared pause public in the Util class and here we use it. static means that this is a class method and we will discuss that shortly. void means that the method does not return a value; that also will become clearer shortly. After goForward there are parentheses. This indicates that goForward is a method; if there are no parentheses then we would be declaring a field. Inside the parentheses is a parameter list which may be empty. In this case, it is not empty. Here, a variable call “tim”, for time, is declared. This variable is similar to a variable in algebra. Here is it declared as an “int” or integer. The variable is then used when we call pause. The value that tim is assigned when goForward is called is passed to pause.

The main method is testing code for the ZTRBot class. First, we must make an instance of our robot. “new ZTRBot()” does that and we save the value in the variable bot. The call to goForward is made in main. Here goForward is called with a parameter of 1000. Putting 1000 inside the parentheses assigns the value 1000 to the first variable in the parameter list, in this case tim. If you recall, time is measured in milliseconds so calling goForward with a parameter of 1000 causes the robot to go forward for 1 second. As we add more functionality to ZTRBot by adding methods, e.g. goBackward, we will call them in main to test them.

```
package MyRobot;
import josx.platform.rcx.Motor;

class ZTRBot {
    public void goForward(int tim) {
        Motor.A.forward();
        Motor.C.forward();
        Util.pause(tim);
        Motor.A.flt();
        Motor.C.flt();
    }

    public static void main(String[] args) {
        ZTRBot bot = new ZTRBot();
        bot.goForward(1000);
    }
}
```
Now is the time to discuss what static means. Recall that we said that a class is a Java description of some thing. That is true so long as you think of a thing in very broad terms. For example, the Util class that we started with does not correspond to any kind of real thing. Its “thing-ness” is just that it is a convenient place to put useful, general-purpose methods. ZTRBot, on the other hand, does describe a real thing, in this case a kind of robot. A specific robot, the Roverbot you should have built, is an instance of a ZTRBot.

The Motor class in LeJOS has three class, static, variables, A, B and C, which hold instances of Motor. These variables correspond to the three output port on the RCX that, not surprisingly, are labeled A, B, and C. The designers of LeJOS decided to create these three instances and then not allow programmers, you, to create any more instances of Motor. The reason is that memory, RAM, on the RCX is quite limited and there will never be more than three output ports on the RCX (a new version of the RCX with more output ports would need a new package of classes in josx.platform). All of the methods in Motor are instance methods; each functions in the context of a specific motor. For example, forward starts a motor turning in one direction; backward starts it in the opposite direction. Motor.A is the motor attached to output port A. Motor.A.forward() starts the motor turning in the forward direction. At this point we could get philosophical and talk about the essence of “forward-ness” and “backward-ness”. Let’s not. Instead, arbitrarily assign one direction of movement of the robot to be forward and the opposite direction to be backward. Then, when you program the robot to go forward, make sure that the wires connecting the motors to the output ports are attached so that the robot does the right thing.

To recap, static methods are universal. You can use them anywhere simply by typing the class name, a period, and the method name, e.g. Util.pause(1000). Instance methods require you to have created an instance of a class and then call the method in the context of the instance, e.g. bot.goForward(), where bot has been initialized to be a ZTRBot.

If you have not already done so, compile your ZTRBot class and download it to your robot. Run the program a few times by pressing the run button. It is a pretty boring program but at least it makes your robot move.
The next step is adding more movement commands. At right is ZTRBot with four new methods added: coast, turnRight, turnLeft and goBackward. Each method was created by copying and pasting goForward and then changing the method slightly. We have also replaced the calls to Motor.A.flt and Motor.C.flt with a call to the new method coast. Note that we do not an instance variable in front of coast. The reason is that coast is being called by other methods where the instance has already been specified so that instance carries over to coast. The rule is that you can call instance methods from other instance methods in the same class but not between classes.

At this point, you need to finish creating ZTRBot. There are four states for each Motor, A and C. The states are forward, backward, brake and coast. Using basic math and the fact that there are two motors, that means that there are 16 possible methods for you to write (if don’t know where 16 comes from, talk to a math teacher to figure it out). Some of these methods are not very useful but most of them are. So, go ahead and write the code, compile it and then run it to see how well you did. Note that when I test this kind of code, I hold up the robot and watch the wheels move and compare it to how I think they should move. If, for example, I think that the robot should go forward and, instead, it spins, I change the wiring of the robot so that the robot does what I intended. This technique is helpful whenever I start testing a new program.

```java
package MyRobot;
import josx.platform.rcx.Motor;

class ZTRBot {
    public void coast() {
        Motor.A.flt();
        Motor.C.flt();
    }

    public void turnRight(int tim) {
        Motor.A.forward();
        Motor.C.stop();
        Util.pause(tim);
        coast();
    }

    public void turnLeft(int tim) {
        Motor.A.stop();
        Motor.C.forward();
        Util.pause(tim);
        coast();
    }

    public void goBackward(int tim) {
        Motor.A.backward();
        Motor.C.backward();
        Util.pause(tim);
        coast();
    }

    public void goForward(int tim) {
        Motor.A.forward();
        Motor.C.forward();
        Util.pause(tim);
        coast();
    }

    public static void main(String[] args) {
        ZTRBot bot = new ZTRBot();
        bot.goForward(1000);
        bot.turnRight(1000);
        bot.turnLeft(1000);
        bot.goBackward(1000);
    }
}
```
At this point, you should have a working ZTRBot. ZTRBot is the base class that we are going to extend to create the rest of the bots in this book. Extending a class goes back to the idea of creating new classes from existing classes by adding something new to a class and giving it a new name. The power in this is that we have access to all of the base class’s methods and fields but cannot make changes to the base class that might cause errors. In addition to adding methods and fields, when you extend a class you can override methods in the base class. In this case, we have overridden the main method something new. Notice also that there is no import statement for the Motor. That is because at this level we do not want to directly manipulate the motors on the robot. ZTRBot is an API, a set of methods, that allows programmers to manipulate robots without having to think about all the little steps involved. If the API is good enough, we will use it over and over and never touch the motors directly again. This is going to be really important when we move from RCX based robots to more powerful ones. Then, we will just need to recode our ZTRBot class and have all extended classes just work on our new robots.

To test this extended class, create a new Java file in rcx and name it TestBot. Then copy the code into your file and compile it. There are several errors that you might get. If ZTRBot has not been compiled successfully or you leave out the package statement, you will get a message saying “cannot access ZTRBot”. If you leave out a semicolon, you will get a message “semicolon expected”. Etc. There is usually a line number for where the error occurred. If you get errors, go to that line and, obviously enough, fix the error. To be clear, there is no shame in getting errors. It is all part of the learning process. When I first started programming, I made almost every error possible but I only made the same mistake once. Learning from your mistakes is the hallmark of a good programmer.

package MyRobot;

class TestBot extends ZTRBot {
    public static void main(String args) {
        TestBot myBot = new TestBot();
        myBot.goForward(1000);
    }
}

3 Creating extended classes with the thought that underlying classes may be changed is called layering. Not importing classes in extended classes that were imported in base classes is called data hiding. Data hiding is intended to keep programmers thinking about what is important and ignoring that which is not.
The code at right is an updated version of TestBot. Here we add a for loop to the code. In its most common form, the for loop has three sections inside the parentheses. The first part creates and initializes a variable. I generally use idx for the name of this variable but the name is totally arbitrary. The variable is defined as an int, integer, and given the value of 0. The second part is the test. If the test evaluates to true, the statement or block of code after the parentheses is executed. If the test evaluates to false then the next statement is executed, in this case the coast statement. If the test evaluated to true, then after the block of code is executed, idx is incremented (++ means add one to the value contained in the variable) and the test in the second part is done again. This looping continues until the test evaluates to false which will be after 5 iterations, i.e. 5 times through the loop).

### Using Sensors

As stated earlier, there are three things that define a robot: the ability to autonomously effect the environment, the ability to autonomously sense the environment and the ability to communicate. ZTRBot satisfies the first criterion; it can move around without direct human control. Now it is time to start working on the second criterion, sensors.

### Polling Maze Runner
The code at left is for a maze running robot. This robot extends ZTRBot so all of ZTRBot’s methods are available. The class has a main method so it is a program. The main method is conceptually divided into two parts: initialization and execution. This is generally how all programs are set up.

The initialization section tells the program what kind of sensors your robot has attached to the sensor ports and how you want sensor readings reported. Then it turns on the sensors. Next, it creates an instance of the robot. That finishes the initialization part of the program.

The execution section is where the program actually runs. This is where we take our algorithm and translate it into Java. In this case, we are using a polling algorithm. The algorithm is really simple, three step procedure. First, ask if the left touch sensor is pressed, i.e. we have run into a wall on the left. If so, back up and turn to the right, away from the wall. Second, if left sensor is not pressed, ask if the right sensor is pressed. If the right sensor is pressed, back up and turn to the left. Finally, if neither sensor is pressed, go forward for a short time. Then, start the algorithm over and never stop.

To get into more depth, the initialization section is at left. Just like Motor, Sensor has three static variables, S1, S2 and S3, to hold Sensor instances that correspond to the three sensor ports on the RCX. Because there are a variety of
sensors that can be attached to these ports, the first step in using a port is to tell the computer which type of sensor is attached. We do that by calling the method `setTypeAndMode` with the right parameters. How do we know what the right parameters are? We look in the Javadoc.

The entry for this method is displayed at left. This method takes two parameters: type and mode. This is explained in the parameters section of the Javadoc entry. The first parameter is the kind of sensor attached, 1 stands for touch, 2 is for temperature, etc. The second parameter is the mode. Mode tells the computer what kind of sensor reading you want. If you want raw numbers from the sensors you enter 0. If you want it as a percent, the way that RIS works with light sensors, you enter 0x80. Etc. Note that mode numbers are written in hexadecimal, base 16, instead of the more common decimal, base 10. That is, 10 in decimal is 0x0A in hex and 0x10 in hex is 16 in decimal. The reason for using hex is that it makes it somewhat easier for the computer to understand the number.

The final line in the initialization section is constructing the robot’s instance. What we do here is create a computer model of the robot that corresponds to the physical robot that we have.

```java
while (true) {
    if (Sensor.S1.readBooleanValue()) {
        myBot.goBackward(500);
        myBot.turnRight(500);
    } else if (Sensor.S3.readBooleanValue()) {
        myBot.goBackward(500);
        myBot.turnLeft(500);
    } else {
        myBot.goForward(100);
    }
}

```

The execution section, at left, is where our program spends all of its time after it is initialized. while starts a loop similar to the for loop that we talked about earlier. But while is simpler than for. To repeat what was said earlier, a while loop is composed of the

```
while (true) {
    if (Sensor.S1.readBooleanValue()) {
        myBot.goBackward(500);
        myBot.turnRight(500);
    } else if (Sensor.S3.readBooleanValue()) {
        myBot.goBackward(500);
        myBot.turnLeft(500);
    } else {
        myBot.goForward(100);
    }
}
```
word “while” followed by a condition and a statement to execute if the condition is true. In this case, the condition is always true since we have a literal true as the condition. So, this is called an infinite loop because the condition will never be false and the statement will be executed forever, or until a statement inside the loop tells it to stop looping. Inside the loop, the first if statement asks the question, is sensor 1 pressed? If so, the robot backs up and turns right. If sensor 1 is not pressed then the else clause is executed. In what is called a nested if, the else clause is another if statement. This asks the question, is sensor 3 pressed? If sensor 3 is pressed then back up and turn left. If not, the else clause says to just go forward. Note that the go forward time is much shorter than the back and turn times. There is a reason for this that you should be able to determine if you experiment a little with this program. The reason is actually quite important and is why polling as a technique is not suitable in a lot of programming situations.
**Listener Maze Runner**

Polling is an older approach to real-time programming, which is what the programming of robots is. On the next page, we have the new, more advanced version of a maze runner. This version is an *event driven* or *interrupt driven* program. That is when something happens, an *event*, that our robot is interested in, a method, a *handler*, is called to deal with it. When the handler is done, the robot goes back to doing what it was doing before.

This leads into the topic of *threads*. A thread is an execution path. Think of it as defining which instruction gets executed now. In a multi-threaded environment, which Java is, several threads can exist at the same time. In a process called *time slicing*, the computer actually runs one thread for a short time then switches to another thread and another. The time is so short, milliseconds or less, that it appears that all the threads are running at the same time. In our polling version of maze runner, we only had one visible thread\(^4\), the thread that calls our main method. In our listener maze runner, we have two: the main thread and a sensor listener thread. The purpose of the main thread is the same as always; it runs our program. The purpose of the sensor listener thread is to keep checking to see if a sensor event has occurred. When the event occurs, all the handlers that have registered on that thread run.

Conceptually, a multi-threaded program separates *synchronous* activities, things that need to be done on a regular basis, from *asynchronous*, things that happen randomly, ones. In this case, we repeatedly tell the robot to go forward for a short period of time on a regular basis on the main thread. The robot runs into walls at random time intervals and the sensor thread handles those events.

A program keeps running so long as at least one regular thread is running. That is, your main thread can explicitly start some other threads and the program is *kept alive* until the last thread terminates. Some threads, like the sensor thread, are called *daemon* threads and don’t keep the program alive and so you don’t have to explicitly terminate those threads before your program will exit.

\(^4\) Usually, there are many invisible threads. For example, if you have ever gotten the message “application not responding” that is because a timer thread is running that says every program must respond to keystrokes or mouse clicks within a defined period of time. If the program is not responding, it is probably broken and you, the user need to do something to fix it. At this point, you don’t need to be concerned with these invisible, *background*, threads.
Getting back to the example, this MazeRunner class, like most of our other classes, starts by extending ZTRBot. This makes our starting point a well-tested class and we can concentrate on the new features. It also implements the SensorListener interface. What does it mean to implement an interface? It is simple, really. When we extend a class, we do two things: we say that our new class has all the methods and fields of the old class and we say that all those methods in our new class do exactly the same thing as the methods in the old class unless we say otherwise. The first part, saying what is in a class is called *defining the interface*. The second part, saying what the methods do, is called *implementing the interface*. When we implement an interface, e.g. SensorListener, we say that our new class will have all the methods listed in SensorListener but we must define the methods ourselves, we don’t get default ones like we do when we extend a class. The Java language requires us extend one and only one class but allows us to implement as many interfaces as we like.

In this case, implementing the SensorListener interface means that we have a method in our new class, MazeRunner, that is named `stateChanged` and that method has three parameters: the sensor that had a state change, the old value of the sensor, and the new value of the sensor. If you look at the `stateChanged` method, it does pretty much what you expect. It checks to see which sensor had the state change and it checks to see what the change was. A new value of 1
indicates a press; a new value of 0 indicates a release. For a maze runner, a release is not interesting so we don’t write any code for that event.

Try running this maze runner and see what happens. Guess what? It doesn’t work. When you run it and press a sensor, it tries to go backward and tries to turn but it doesn’t get very far. Why?

It’s the threads. In our event handler, we have goBackward and turnRight with times of 500 milliseconds specified. In those two methods, we call pause to stop our thread from executing so that real-time can catch up with computer-time. So, what do you suppose happens when a thread decides it needs to stop running for awhile? Well, it yields its time slice and marks itself unrunable for the time it wants to be stopped. What does yield mean? It means other runnable threads can start running. Guess which thread gets to start? Our main thread starts running. And what does the main thread do? It immediately tells the robot to start going forward. So, the main thread does not wait for the handler to finish.

When you start working with threads, unexpected things happen. As a programmer, it is your job to figure out what is wrong and then fix it.
**Better Sensor Maze Runner**

Since interactions between threads is causing a problem, we need to come up with some way to keep the main thread from running while our event handler is working. The easiest way is create a boolean variable and call it “intFlg” which is shorthand for interrupt flag. This flag is set to true will the stateChanged method is active and used to keep the main loop, the goForward statement, from running while we do the back up and turn maneuver. Try taking it out and see what happens. The pause that is in stateChanged has a similar function. If the pause is not there, when the goForward in the main loop finishes up it turns off the back up before it should. The pause allows the main loop goForward to finish before the stateChanged goBackward starts. Note that this code is a real kludge⁵.

A far better way would be to use the Java

```java
package MyRobot;
import jox.platform.rcx.*;

class MazeRunner extends ZTRBot implements SensorListener {
    private boolean intFlg = false;
    
    public void stateChanged(Sensor sen, int oldVal, int newVal) {
        intFlg = true;
        Util.pause(30);
        jox.platform.rcx.LCD.showNumber(newVal);
        if (Sensor.S1 == sen && newVal == 1) {
            goBackward(500);
            turnRight(500);
        } else if (Sensor.S3 == sen && newVal == 1) {
            goBackward(500);
            turnLeft(500);
        }
        intFlg = false;
    }

    public static void main(String[] args) {
        Sensor.S1.setTypeAndMode (1, 0x20);
        Sensor.S1.activate();
        Sensor.S3.setTypeAndMode (1, 0x20);
        Sensor.S3.activate();
        MazeRunner myBot = new MazeRunner();
        Sensor.S1.addSensorListener(myBot);
        Sensor.S3.addSensorListener(myBot);
        
        while (true) {
            if (!myBot.intFlg) myBot.goForward(20);
        }
    }
}
```

⁵ A kludge is a piece of code that works but is ugly or inelegant.
Even Better Listener Maze Runner

My philosophy for writing production code, code that I am going to use a lot or sell, is that I write it from scratch three times. The first time is to see if I can accomplish my goal. The second time is to fix the bad engineering decisions I made the first time. The third writing is to “do it right”. Usually, programmers don’t have the luxury of writing a program three times. But this time we are at least going to write the program twice.

Initialization section

At left is the initialization section of MazeRunner. Quite frankly, it is kind of messy. There are several static calls to sensor methods and then we create our robot and make it listen to the sensors. This is a bad style because mixes lower level method calls with the high level construction of our robot. When you write code, you need to think about layers of execution and concept. The main method of a class should construct the main class, initialize it and then start the program running.

In the code at left, we have greatly improved the structure of the program by adding an initialization method, initSensors. Note that this method is an instance method; when we call it we want to specify which robot we are initializing. Also note the use of the keyword “this”. this means the object that was used to call the method. It is very handy because you don’t have to pass myBot into the initSensors method; it is already there.
The listener section is also a bit of a kludge. Using a flag to communicate between threads, in this case the listener thread and the main thread, is a bad idea. But we had to put it in because the pause in `goBackward` yields to the main thread.

Again, you should experiment with taking out these statements or changing the times so that you will start to understand the interactions that go on in event driven programming. Also be aware that Java uses the concept of listeners in virtually every program so this concept is very important.

```java
private boolean intFlg = false;
public void stateChanged(Sensor sen, int oldVal, int newVal) {
    intFlg = true;
    Util.pause(30);
    josx.platform.rcx.LCD.showNumber(newVal);
    if (Sensor.S1 == sen && newVal == 1) {
        goBackward(500);
        turnRight(500);
    } else if (Sensor.S3 == sen && newVal == 1) {
        goBackward(500);
        turnLeft(500);
    }
    intFlg = false;
}
```
Javadoc

Javadoc is the documentation for the Java class libraries and the name of the tool that creates the documentation. Pictured at right is the Javadoc for the LeJOS libraries (the SDK Javadoc is similar but much larger). It is displayed using a web browser. The screen is broken into three parts: the packages window, the classes window and the document window.

The packages window shows all the packages in the documentation set. There is a hierarchy that looks somewhat like a file tree because that is what it is. The roots java and javax (Java extension) packages are common to all versions of java. Clicking on a package displays the classes it contains in the classes window.
The classes window displays all the classes in the currently selected package. Packages are the set of classes, objects, which are related to a specific thing. You select a package to display by clicking on it in the packages window. In this case, the package joxp.platform.rcx is being displayed in the classes window. This package contains all of the classes related to the RCX. Clicking on a specific class displays it in the document window. In this case, the Motor class is being displayed. The only way to learn programming in Java is to explore the Javadoc.
The document window has several parts. The parts of the document are always the same format; the Javadoc tool compiles it that way. At the top is the navigation header. This is a set of hyperlinks that allow you move, navigate, through the Javadoc. This part of a Javadoc class description is always the same.

```
josxplatform.rcx

Class Motor
```

```
java.lang.Object
|
|-- josx.platform.rcx.Motor
```

```
public class Motor
extends Object


Example:

Motor.A.setPower(1);
Motor.C.setPower(7);
Motor.A.forward();
Motor.C.forward();
Thread.sleep (1000);
Motor.A.stop();
Motor.C.stop();
```

Just below the navigation header is the class name. This gives the name of the class and the package that it belongs to. This followed by the class hierarchy of the current class. We will cover what this means later. For now, think of it as the current class is just the same as the class listed above it, in this case Object, with a few additions or modifications. Note that class hierarchies and packages have nothing to do with each other.

Usually, following the class name section is a description of the class and an example fragment showing how to use the class. This if optional but when you write your Javadoc code, it is a good practice to follow.
When you scroll farther down in the example, the field and method summary sections are displayed. In this case, there are three fields. Each field corresponds to one of the three motor outputs on the RCX. The method summary is much longer with around 20 methods specified (this example shows only the first three but the others are in there). Clicking on the method or field name takes you to the detailed description of the method or field.  

<table>
<thead>
<tr>
<th>Field Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>static</strong> Motor A</td>
</tr>
<tr>
<td><strong>static</strong> Motor B</td>
</tr>
<tr>
<td><strong>static</strong> Motor C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void</strong> backward()</td>
</tr>
<tr>
<td>Causes motor to rotate backwards.</td>
</tr>
<tr>
<td><strong>static void</strong> controlMotor(char aMotor, int aMode, int aPower)</td>
</tr>
<tr>
<td><strong>Deprecated.</strong> I've decided to remove this method. If you really need it, check its implementation in classes/josx/platform/rcx/Motor.java.</td>
</tr>
<tr>
<td><strong>void</strong> flt()</td>
</tr>
<tr>
<td>Causes motor to float.</td>
</tr>
</tbody>
</table>

---

Note that one method has been **deprecated**. Java is a living language; over time it changes. Sometimes this means that certain methods or fields shouldn’t be used because better ways to do the same thing have been added. Because older programs may have the method or field, it can’t be removed entirely. But, it is deprecated in the Javadoc to say that it should not be used in new programs.
**backward**

public final void **backward**()

Causes motor to rotate backwards.

**setTypeAndMode**

public final void **setTypeAndMode**(int **aType**, int **aMode**)

Sets the sensor's mode and type. If this method isn't called, the default type is 3 (LIGHT) and the default mode is 0x80 (PERCENT).

**Parameters:**
- **aType** - 0 = RAW, 1 = TOUCH, 2 = TEMP, 3 = LIGHT, 4 = ROT.
- **aMode** - 0x00 = RAW, 0x20 = BOOL, 0x40 = EDGE, 0x60 = PULSE, 0x80 = PERCENT, 0xA0 = DEGC, 0xC0 = DECF, 0xE0 = ANGLE. Also, mode can be OR'd with slope (0.31).

**See Also:**
- **SensorConstants**

The detailed description displays the declaration of the method or field and a description of what it does. Some, e.g. backward from the class Motor, are quite simple. Others, e.g. setTypeAndMode from the class Sensor, are more complicated, displaying information about parameters for the method and having references to other, related classes.

To be clear, programmers add Javadoc code to their Java programs and then run the Javadoc tool to create documentation for the other people who will use these classes. You, as the programmer, will need to add Javadoc code to your programs in order to provide information to other programmers who will use your code. It is not magic and it is not hard but it is essential to being a good programmer.
Thus ends the brief introduction to Javadoc. In order to become a good programmer, you need to explore the Javadoc. There is a huge amount of useful classes in the JDK. No one can know them all and, even if someone could, each new release of the JDK adds hundreds of new classes. The only way to keep up is to refer to the Javadoc. The Javadoc for LeJOS is in the folder apidocs with the main file index.html. The JDK’s Javadoc is in the docs folder in the JDK’s folder, also accessed with index.html. Open it with your web browser using the file > file open menu item. If you cannot find the index file, you probably didn’t install the Javadoc. Go back to the install section and install it now. Note that the JDK’s Javadoc can easily take one half hour to install (which is probably why you didn’t install it to begin with).
Robotic Behaviors

What makes a robot a robot? It is the ability of the robot to independently accomplish complex tasks. Being able to independently do things is commonly thought of as a sign of intelligence. My definition of a robot has four parts:

1. A robot must be able to sense things in its environment.
2. A robot must be able to effect things in its environment based upon what it has sensed.
3. A robot must be able to communicate about what it has sensed and done.
4. A robot must be able to do all of these things without outside, e.g. human, intervention.

Developing intelligent machines, artificial intelligence, has long been a goal of computer scientists. Traditionally, artificial intelligence programs have been very large and extremely complex. That is, until Rodney Brooks of MIT’s Artificial Intelligence Laboratory started thinking about insects.

Insects have very little computing capacity in their brains. Yet, a fly can find food, avoid predators and reproduce, all of which are complex behaviors. How does something with so little intelligence accomplish such complex tasks?

Basically, an insect simply reacts to its environment. The insect has a set of hard-wired behaviors. A particular environmental stimulus or event trips a behavior. The stimuli can be virtually anything. If something is moving quickly, it jumps. If the humidity goes up, it seeks a mate. If its sugar level gets too low, it seeks food, etc. Note that these behaviors have different priority levels. Finding food is a fairly high priority but avoiding a fly swatter is much higher. So, if the fly is trying to find food but it sees a fly swatter coming, it will immediately stop seeking food and flee.

The idea of insect behavior leads to a fundamentally simple way to control robots. Build a set of behaviors and associate each behavior with a trigger event. Assign each event-behavior pair a priority and then sit back and watch your robot go.
Our simple maze runner program has two separate behaviors, which we list at right. The default behavior starts when you press the run button and it is to just go forward. When the robot runs into a wall, the default behavior is interrupted by the hit wall behavior. After the hit wall behavior completes, the default behavior starts running again.

<table>
<thead>
<tr>
<th>Default</th>
<th>Go forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit wall</td>
<td>Back up, turn left or right</td>
</tr>
</tbody>
</table>

By creating a large set of event-behavior pairs and ordering their priorities, a robot can perform very complex tasks.
**Finite State Machines**

*Finite state machines* are an old paradigm, way of looking at things, for describing how autonomous systems work. The concept of finite state machines has four parts:

1. A set, which must be finite (as opposed to infinite), of states and associated behaviors.
2. A set of state transitions which define the allowed movements between states.
3. A set of rules that define when state transitions occur.
4. A set of input events that trigger state changes.

One variant of finite state machines has been incorporated into UML, the same UML that includes class diagrams. Like class diagrams, finite state machines add visual aids to bypass the thought to word to others’ words to others’ thoughts translations that we mentioned earlier. This list of finite machine concepts is not easy to understand in the abstract. An example, of the maze runner from above, will make the idea of finite state machines much clearer.

Maze runner has two states: blocked and unblocked. The unblocked state has the associated behavior of go forward while the blocked state has the behavior of avoid wall. When the behavior is complete, a state transition occurs. The outside event of a sensor being pressed triggers the state change from unblocked to blocked. When the avoid wall behavior completes, an internal event, the avoid wall movement completing, triggers the change back from the blocked to the unblocked state. Finally, when the go forward behavior completes, the unblocked state loops back into itself.

Finite state machine analysis can be much more complex than this simple example. But the rules and construction of the graph remain the same. Creating a finite state machine diagram makes the process of creating a set of behaviors for a robot much simpler. Remember this technique and compare it to the “be the robot” exercise as we continue with the discussion of behaviors in LeJOS.
Behaviors in LeJOS

A behavior is a Java object. You create a new behavior by creating a class that implements the `josx.robotics.Behavior` interface (UML at right). The details of how behaviors work in LeJOS are slightly different than the preceding description implies. Instead of directly calling an instance of a behavior when a stimulus event occurs, the behavior is asked if it wants to take control. If it does then the suppress method of the currently running behavior is called followed by the action method of the new behavior.

Behaviors work in conjunction with `josx.robotics.Arbitrator`. Arbitration is the process of deciding which behavior should be running and that is what an Arbitrator does. The UML for Arbitrator is at right and you can see it is quite simple. When you create a new Arbitrator, you pass the constructor an array of Behaviors. The array is in order of priority with the highest priority behavior last is the list. After you create the new instance of Arbitrator, you call its start method to start arbitration. Note that the start method does not create a new thread for the arbitration to run on so the start method will never return.

The source code for both Behavior and Arbitrator are in `\lejos\classes\josx\robotics`. The code is straightforward and well commented. This is a good example of how you should code.
MazeRunner

MazeRunner is a program that we have looked at extensively. In the next few pages, we will analyze how to implement it using behaviors in Java. Note that there are three separate classes that will be detailed in the following sections. You can copy each class into a separate file, named appropriately, and compile them. Add them to your RCX project to ensure that they will be compiled using the LeJOS libraries. Note that two of the classes extend ZTRBot, which you should have already created. If you have not created ZTRBot, created it in a package other than MyRobots or haven’t added all of the methods that are used, you will get compiler errors. You must have a working ZTRBot with all the required methods to build MazeRunner.

MazeRunner comes in three separate files. The first is the MazeRunner class. It starts by importing all of the classes of josx.robotics so that it has access to Behavior and Arbitrator. Note that it does not declare any methods or fields other than main so it is simply a program.

main starts by creating two behaviors, one of class HitWall and one of class GoForward. Note that both of these classes implement the Behavior interface so that an instance of either object can be assigned to a variable of type Behavior.

The next statement creates an array of behaviors. The array does not specify a size. Instead, it initializes the array to the two behaviors that we created, move and wall. The ability to put instances of different objects into a single array is called polymorphism. This is only allowed because these classes implement the Behavior interface. That is, in addition to anything else that they might be, these classes are also behaviors.

```java
package MyRobots;
import josx.robotics.*;

public class MazeRunner {
    public static void main(String[] args) {
        Behavior wall = new HitWall();
        Behavior move = new GoForward();
        Behavior[] ba = {move, wall};
        Arbitrator arb = new Arbitrator(ba);
        arb.start();
    }
}
```
The next statement creates an Arbitrator instance with the array of behaviors that was just created. Note that order of parameters is important. The rightmost behavior, wall, has the highest priority. The leftmost behavior, move, is the lowest priority behavior and should be a default behavior, i.e. a behavior that is always ready to run.

Finally, the last statement starts the Arbitrator running. This method never returns. It continually checks each behavior in priority order to see if it is ready to run. When a higher priority behavior is ready, it stops a lower priority behavior and starts the higher priority behavior. If no behavior is ready to run, it keeps looping until any behavior is ready.
GoForward is a very simple behavior. It starts with a package statement so that it belongs to our MyRobot set of classes. It then imports josx.robotics.Behavior so that Behavior is defined. Next, the class is defined as extending ZTRBot and implementing Behavior. Extending ZTRBot gives us all the movement methods for our robot. Implementing an interface simply means that the class defines the methods listed in the interface. For Behavior, the required methods are takeControl, suppress and action. GoForward defines those three methods and nothing more, although it could have other methods or fields if needed.

takeControl is called by the arbitrator to ask if the behavior wants to run. Since this is the default behavior, it always is ready to run so it simply returns true.

action is called by the arbitrator when it determines that a behavior should run. This is a very simple behavior; it turns on both motors in the forward direction and then waits 1000 milliseconds (1 second). Assuming no other behavior interrupts it, this allows the robot to go forward for 1 second before any other command is given to it. However, if a higher priority behavior says that it is ready to run, this behavior will be suppressed and the higher priority behavior will be activated.

suppress is called by the arbitrator for the currently running behavior when the arbitrator finds a higher priority behavior that wants to run. In this case, we stop the robot.

The combination of action and suppress highlight the fact that arbitration runs on two separate threads. The main thread is constantly checking to see if higher priority behaviors are ready to run while a second thread is used to run the suppress and action methods of the behavior instances. The second thread is what actually controls the actions of the robot.
HitWall belongs to the same package as the other classes. It imports Behavior, just as GoForward does, and also imports Sensor so that it can read the values of the sensors and so interact with the environment. It extends ZTRBot so that it has access to the methods used to move the robot and implements Behavior just like GoForward.

**takeControl** is more complex than in GoForward since this behavior will not be used as a default behavior. That is, the only time that we want this behavior to run is when a sensor is pressed. Here when the arbitrator asks, it returns true if sensor 1 or sensor 3, or both, is pressed. The arbitrator then calls its action method.

**action** starts by asking if sensor 1 was pressed. If so, it backs up and turns right (this is away from the wall if the robot is correctly constructed). If not then it assumes that sensor 3 was pressed and so backs up and turns left (again, away from the wall). We use the “if … else” statement since takeControl returning true guarantees at least one sensor is pressed and if sensor 1 is not pressed then sensor 3 must be pressed. We don’t include any statement to make the robot start going forward again. That is the role of the default behavior, GoForward, which starts running again as soon as this behavior, HitWall, completes its action method.

**supress** is the same as in GoForward and, in fact, this is a pretty generic suppression method.
**Exercises**

1. Write a line follower using behaviors using one light sensor. If you have two light sensors, write another version that uses both. Which version is better and why?

2. Draw the finite state machine that describes your line follower.

3. Write a maze runner using behaviors that stops when it encounters a black line (or white line if the surface you are using is black). Draw the finite state machine for this robot.
Navigation

Where am I? That is a pretty important piece of information for people and for robots. While people can generally look around and figure out where they are, robots can’t. And for an autonomous, un-tethered robot to be able to do anything useful, it needs to know where it is. This leads to the topic of navigation.

History of Navigation

The history of navigation goes back a long time. In ancient times, mariners would take to the seas. At first, they stayed within sight of land so that they were sure that they could get home when they wanted to. But after a while, curiosity got the better of some of the seamen and they sailed beyond the horizon.\(^7\) This was the major impetus for developing the science of astronavigation, i.e. navigating by the stars, and the branch of mathematics called trigonometry. Robotic navigation is much simpler than astronavigation, robots move on a flat surface while ships move on a spherical Earth, but trigonometry is used in both. And while you might have misgivings about using trig in the abstract, robots are not abstract. Moving a robot from point A to point B, and then getting it back again, is a simple application of ancient mathematics.

Coordinate Systems

A coordinate system is simply a way of giving places names. To be upfront about this, there is a huge body of complex mathematics that deals with coordinate systems and their implications. Forget about all of that and don’t worry about all of that. Math phobia is common and totally unwarranted when it comes to this kind of trig. Remember that our goal is to have our little robot do something as it rolls around on the floor for a while and then get back to where it started.

\(^7\) It is rumored that about the same time, the old saying “curiosity killed the cat” came into vogue. It is not clear whether the two events are related.
Coordinate systems come in two flavors, rectangular (as known as Cartesian) and polar. An example of a rectangular coordinate system is a chessboard. A chessboard has 64 squares arranged in 8 rows of 8 squares. When some people play chess, they want to record each move in the game (ours not to reason why, ...) for posterity. In order to do so, each square has to have a name. One convention for naming the squares is for each column to have letter, a-h, and each row a number, 1-8. Looking at the picture, the bottom left square is a1, the one above it is a2, and the one at the upper right is h8. This is a coordinate system but not a very useful one for navigation.

The reason that it is not useful for navigation is that it does not have a common metric. Metric means measurement unit and the chess coordinate system, while it does uniquely name every point, does not have the concept of distance. The question how far is a2 from c4 is not a valid question for this coordinate system because there is no metric.

So, let’s create a coordinate system for the chessboard that does have a metric. The simplest such coordinate system numbers each row and column, starting at 1. The lower left square is column 1, row 1 and we write this simply as (1, 1). The square directly above (1, 1) is column 1, row 2 or (1, 2). The square to the right of (1, 1) is (2, 1). This highlights the fact that the names we have created for the squares are ordered pairs of numbers. That is while (1, 2) and (2, 1) have the same two numbers in their names, the two squares are distinct. While this coordinate system has a common metric, the concept is distance is too fuzzy for our robots. For example, (1, 1) is 2 squares distance from both (1, 3) and (3, 3). But if you take a ruler and measure the distance between (1, 1) and (1, 3) and the distance between (1, 1) and (3, 3), you will probably conclude that they are different distances apart.
Finally, let’s create a coordinate system for the chessboard that not only has a metric but also has distances that are useful for robots. We’ll use ordered pairs again, but this time instead of naming the squares, we will name the corners of the squares that form the chessboard. Square (1,1) has four corners: lower left is (0, 0), lower right is (1, 0), upper left is (0, 1) and upper right is (1, 1). Square (1, 2) also has four corners, the lower two that it shares with square (1, 1), hence we have already named those corners, and the upper two which are (0, 2) and (1, 2). Measuring distances between corners, points, is easier and more precise because the distance inside a point is small compared to the distance between points. This is a grid coordinate system and it excels at allowing us to precisely locate robots on a flat surface and, thus, to make maps of a room or area. In a grid coordinate system, the ordered pairs are usually referred to as XY pairs. In many rooms, perhaps the one you are in, the floor is made out of tiles. In the United States, flooring tiles are usually 10 or 12 inches on a side and square. This makes an excellent place to test navigation robots because tiles, in addition to forming a grid, also tend to be very smooth, which as you will see, is very important for simple robots.

Unfortunately, a grid coordinate system is terrible for robotic navigation. The reason is that we created the grid, not the robot. The robot doesn’t know anything about its ordered pair position as it moves across the floor. It can’t see it, unless of course you have both painted the grid on the floor and attached sensors so the robot can see the painted grid and that usually isn’t the case. But the robot need not be completely clueless about its position. The robot can be programmed to keep track of two things: how much it has rotated from its initial direction and how far it has traveled from its initial point. In other words, it can store its present location in terms of the distance and direction from its starting location. And it store multiple distance and direction pairs as it moves and turns.
Using distance and direction, we can create a coordinate system that is much more robot friendly. This type of coordinate system is a polar coordinate system. This type of coordinate system is very different from a grid system. This coordinate system moves with the robot. Wherever the robot is, its coordinates are always (0, 0). Which seems really odd and very confusing. But it is neither. Think about it from the robot’s perspective. The robot wants to go to several places and return to its starting point. For example, the robot wants to travel along the lines in our grid while sketching a square. To make this trip it will go 4 square lengths, turn 90 degrees, go 4 more square lengths, turn another 90°, travel another 4 square lengths, turn 90° and travel another 4 square lengths to return to its starting position. Note that distance and direction form an ordered pair, e.g. (90°, 4), that is similar to the ordered pairs of our rectangular coordinate system. The name for these ordered pairs is a heading. The trip just described has the robot navigating on four different, but not distinct, headings: (0°, 4), (90°, 4), (90°, 4), and (90°, 4). Headings are closely related to distance vectors. Distance vectors are ordered pairs of angles and distances, just like headings, but instead of having degrees expressed relative to the current orientation of the robot, distance vectors express it in terms of the starting orientation of the robot (or some other arbitrary direction, e.g. magnetic North). The list of headings (0°, 4), (90°, 4), (90°, 4), and (90°, 4) are identical to the distance vectors (0°, 4), (90°, 4), (180°, 4), and (270°, 4) where 0° is defined to be the direction the robot is originally pointed. The grid at right has polar coordinates superimposed to show the trip our robot makes. Remember that polar coordinates move with the robot. A new polar coordinate system is created each time so that the robot can change its heading.
The branch of mathematics called trigonometry was developed to solve the problem of navigation. Trig was developed to allow trading ships and explorers to make it to their destination and, more importantly, make it back home as well, on oceans that had no landmarks. We can use the trig functions sine and cosine to convert our distance vectors in our polar coordinate system to XY pairs on our grid coordinate system. The equations are trivial. The change in our Y position, ΔY (delta Y), (up and down on the grid) is simply the length of the distance vector times the cosine of the angle. And the change in our X position, ΔX (left and right on the grid) is simply the length of the distance vector times the sine of the angle. If we start the robot at (0,0) then it returns to (0,0). If we want to keep track of where our robot is located on the grid, we just add the ΔX and ΔY to our position each time we change headings. That way, our robots always knows its XY coordinates each though it “thinks” in polar coordinates.

Going the other way, from rectangular to polar coordinates is a bit trickier. To calculate the length of the distance vector you use the Pythagorean theorem, \(a^2 + b^2 = c^2\) where \(a\) is ΔX, \(b\) is ΔY and \(c\) is the length of the distance vector. The angle of the distance vector is the arcsine of \(a / c\) or \(ΔX\) divided by the length.

That is all that there is to navigation. The process is conceptually very simple, e.g. go a little ways, calculate where you are and repeat. The various navigator classes that we will be using, TimeNavigator, RotationNavigator and CompassNavigator, handle all the trig complexity. However, it is important for you to understand the basics of navigation because these classes sometimes don’t work in the real world (or don’t work as well as we would like).
NavBot.java – A Navigation Example

The code at left demonstrates one of the navigator classes, RotationNavigator. This code runs on a ZTR robot, e.g. Roverbot or NavBot, that has rotation sensors attached to both motors. wheelDiameter is the diameter of the drive wheels (assuming all drive wheels are the same size). driveLength is the distance between the wheels on opposite sides. gearRatio is the gear ratio between the wheels and the rotation sensors. Note that all three parameters may require adjusting due to factors such as wheel slippage and skid turning. For example, drive length is used to calculate how far a robot turns. I measured the drive length on a Roverbot to be 14 cm. However, in practice, I had to reduce this to 8 cm in order to have the robot turn 90° when I told it to turn 90°. Notice that I am expressing all lengths in metric units. The units you use does not really matter to the navigators. The only thing that is important is to be consistent, i.e. always use inches, cm or feet. If you mix units, don’t expect the robot to go where you expect. Also note that Lego stamps the diameter of the tires in mm on the side of the tire. This makes for one less thing to measure but be sure you divide this number by 10 if you use cm for everything else.

---

8 If you don’t have rotation sensors, you can use TimingNavigator instead. TimingNavigator is much less accurate and stable than RotationNavigator but it is much better than nothing. Note that the parameters are different for the constructor so change them accordingly.
Programming a Robot to go here, there and back

MapBot.java - the main program

```
package MyRobot;

import josx.platform.rcx.*;
import josx.robotics.*;

class MapBot {
    public static void main(String[] args) {
        float driveLength = 8.000f;
        float[] xpos = new float[10];
        float[] ypos = new float[10];

        RotationNavigator nav = new RotationNavigator(4.3f, driveLength, 1);
        Behavior b1 = new Move(nav);
        Behavior b2 = new Bump(nav, xpos, ypos);
        Behavior b3 = new Home(nav, xpos, ypos);
        Behavior[] bArray = {b1, b2, b3};
        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }
}
```

Like NavBot, the program MapBot.java is designed to run on a ZTR robot. Notice that we do not extend ZTRBot. Control of the motors is done by the navigator. This robot uses RotationNavigator and so has rotation sensors attached to sensors 1 and 3 and a touch sensor attached to sensor 2. The program combines behavior based programming with a RotationNavigator. It has three behaviors: Move, Bump and Home. The three behaviors will be described in detail in the next sections. To summarize, Move, the lowest priority behavior, simply starts the robot heading towards a randomly selected point. Bump is activated when the robot runs into something and causes the robot to backup and change direction. Home is activated when the robot has bumped 10 times and tells the robot to go to (0, 0).

The MapBot class is very simple with no fields and only one method, the entry point main. main starts by creating a RotationNavigator instance. Then it creates the three different behaviors, passing the navigator instance to each. It uses the same instance of the navigator in each behavior to ensure that there is one common place that tracks all the movements of the robot. After creating the three behaviors, it creates an array of the behaviors in priority order, lowest priority first, creates the Arbitrator instance and starts it.
Move – the lowest priority behavior

```java
package MyRobot;
import josx.robotics.*;
import josx.platform.rcx.*;

public class Move implements Behavior {
    private final int MAXDISTANCE = 70;
    private boolean active;
    private Navigator nav;

    public Move(Navigator nav) {
        this.nav = nav;
        active = false;
    }

    public boolean takeControl() {
        return true;
    }

    public void suppress() {
        active = false;
        nav.stop();
    }

    public void action() {
        active = true;
        while (active) {
            float x = (int) (Math.random() * MAXDISTANCE);
            float y = (int) (Math.random() * MAXDISTANCE);
            nav.gotoPoint(x,y);
        }
    }
}
```

Move.java is the lowest priority, default, behavior in MapBot. It contains a constructor that saves the navigator instance so that it can use it whenever it is activated. It also has the three methods, takeControl, suppress and action, that are required by the Behavior interface. Observe that takeControl always returns true, indicating it a default behavior. action uses a pseudo-random number generator (Math.random returns a random number between 0 and 1) to randomly pick X and Y coordinates that range from 0 to 70 cm (roughly 0 to 2 feet). It then calls gotoPoint to physically move the robot to the point (gotoPoint is a blocking function; it does not return until the robot makes it to the point specified). It then loops, picking new random numbers, until its suppress method is called. The suppress method does two things. First, it sets the active flag to false; this causes the gotoPoint loop to stop looping. Second, it calls nav.stop to stop the robot from moving and to update the current position of the robot in the navigator instance.

That is all there is to the move behavior.
Bump – the middle priority behavior

Bump.java is more complex than Move.java. It implements both the Behavior and SensorListener interfaces. That means that it defines the three Behavior methods, takeControl, suppress, and action, and the SensorListener method, stateChanged.

The suppress and action methods are very simple. suppress simply stops the robot and updates its recorded location by call nav.stop. action backs the robot up 20 cm. takeControl returns bumpFlg

stateChanged is called whenever the sensor’s value changes. For some types of sensors, like rotation sensors, this is not very useful since the value is constantly changing. For other types, like touch sensors, it is very useful since the event does not happen often. The listener is used here because the touch sensor may be pressed and released in the time period between the arbitrator calling takeControl. If that happens, the robot would miss the fact that it ran into something.
Home.java – the highest priority behavior

Home.java is the highest priority behavior in MapBot. The purpose of Home is to have the robot go back to its starting point after randomly moving about. The constructor is passed two arrays: xpos and ypos. Arrays are passed by reference. That means that the array that was created by main is the same array used here and in Move. So any changes made to elements of the array in Move or MapBot would be reflected here as well. Non-array parameters are passed by value. That means that changes in the method are not reflected in the calling program so changing the value of nav would not effect main or any other method.

Home takes control when the xpos array is filled. Presumably, the last element of the xpos, x position, array will not have the value of 0 when the robot bumps into something. So that when the robot has bumped into things, in this example, 10 times, the array will be filled with non-zero location coordinates and the robot will return to its starting point. After it returns to the starting point, it makes several beeps to provide auditory feedback to the user.

```
package MyRobot;

import josx.robotics.*;
import josx.platform.rcx.*;
import josx.util.*;

public class Home implements Behavior {
    private Navigator nav;
    float xpos[];
    float ypos[];

    public Home(Navigator nav, float[] xpos, float[] ypos) {
        this.nav = nav;
        this.xpos = xpos;
        this.ypos = ypos;
    }

    public boolean takeControl() {
        return xpos[xpos.length-1] == 0.0;
    }

    public void suppress() {
        nav.stop();
    }

    public void action() {
        Sound.beepSequence();
        nav.gotoPoint(0,0);
        Sound.twoBeeps();
        Util.pause(5000);
        Sound.beep();
    }
}
```
**Wrapping up Navigation**

Navigation is crucial to robotics. In this section, we have used the RotationNavigator to keep track of the robot as it moves about. You can easily substitute the TimerNavigator or the CompassNavigator and see how well your robot keeps on track. The next couple of sections build upon the navigator. In the communications section, we will keep track of where objects are located and pass the information on to a host PC and another robot. In the introduction to swing, we will collect the map data and display it graphically on a host PC.

**Exercises**

1. Navigation has a lot to do with precise movements, as does line dancing. Go on-line and research line dancing. Implement 6 dance steps in a new robot, DanceBot, and program it to do a line dance in time with the music. Optionally, have the robot play a tune itself while dancing.

2. Create a MapBot that will randomly find 10 fixed points, e.g. walls, in a room, return to its starting location and then display the coordinates of the points when the “view” button is pressed.
Communications

Communications is a key element in robotics. What good is our sending a billion dollar robot to Mars if we never find out what it learned? What good is our little mapping robot if we never find out where it went? The answer is not much good. Sending data from the RCX to your PC and sending data from RCX to RCX is key. You are already communicating between your PC and RCX every time you download a program. Wouldn’t it be cool to write your own programs to do the same thing? It’s amazingly easy to do.

Java programs use *streams* to communicate. Stream is a conceptual word and defining it is difficult. One definition is a stream is a connection between two independent entities used to transfer data or information. In my mind, that does not create a mental image that helps me understand. So, think about sitting with another person and having a conversation. Your spoken words are a word stream. For most people, the mouth is an output device and the ear is an input device. Streams are based upon connections and have only one direction; mouths can’t hear and ears can’t speak. In a conversation between you and me there are two streams: 1) I talk to you and you listen to me and 2) you talk to me and I listen to you.

Java has an extensive set of packages that deal with communications. The designers of Java generalized the idea of streams to include things that you would not expect. Virtually all communications, with the major exception of graphic user interfaces, GUI, between a program and the real world is based upon streams. Examples include surfing the web, saving a file on a hard drive, typing in the console window, and using a modem. The most basic streams are InputStream and OutputStream.

While people use word streams to communicate, computers use *byte streams, a.k.a. data streams*. Remember that a byte is the smallest primitive data type in Java. As it happens, ASCII characters, letters and numerals, are stored in single bytes. So each ASCII character is sent as one byte in a byte stream. Integers take four bytes so they get sent through the stream as four consecutive bytes. So
if your program is on the receiving end of a data stream how can it tell the difference between an int, a byte, a float or whatever? Simple answer is that it can’t. A byte is a byte. What you have to do is create a protocol. A protocol is the rules for a conversation that the sender and receiver agree to follow. While you might not be familiar with the word, if you use the Internet you use protocols all the time (it’s the P in TCP, IP and HTTP). Protocols are designed to make communication easier. Think about the protocol for using a telephone. The rule of the telephone protocol is that you pick up the phone when it rings. Next, you say “hello” or a similar greeting. The caller responses and gets to the purpose of the call. When the caller is done, you have the opportunity to response. Finally, the caller ends the call by saying “goodbye” or something similar. That is the protocol for using a telephone. The introduction of caller id illustrates how disruptive not following an established protocol is. Answering “hello Bob” when Bob is calling and not used to the new greeting, disrupts the flow of the conversation. If you haven’t done this already, try it.
PC to RCX Communications

Robot communication is very similar to a conversation. To have an RCX communicate, talk, to another RCX or PC, you need to do three things: add an output stream to the sender’s program, add an input stream to the receiver’s program and establish a protocol for the communication. The next two sections present the code for the PC and RCX. The protocol is very simple. The RCX will listen until it hears something. It will then read and interpret the bytes as ints until the PC closes the stream. That is it for the protocol.

PCSend.java

The program at left is the PC part of communication example. The program uses RCXBean to manage the data stream with the RCX. It starts by creating a new RCXBean. Then it specifies the port the IR tower is on. This is all the setup the RCXBean needs to be ready to communicate with the RCX. To download the data, it loops through the two arrays of ints.

The code that uses the RCXBean is inside a try-catch block. This makes it possible to gracefully deal with the variety of unexpected that could occur, e.g. tower not plugged into PC, RCX is turned off during session, sun shines on IR port blinding it, etc. If an error occurs, the program prints an error message (something we can do on the PC but not on the RCX) and exits. A graceful exit beats the
alternative, which, since this is hardware that we are dealing with here, could include hanging the system and requiring a re-boot.

You can cut and paste this into the IDE. Note that you compile this with the standard JDK compiler, not the LeJOS compiler. You will need to add the LeJOS class files, classes.jar, comm.jar, and pcrxcomm.jar to your project classpath. In JCreator, you click on Project>Project Setting and click on the required libraries tab. Add a new set libraries, naming it LeJOS, and add the three archive, jar, files. You also need to have tower.dll available when PCSend runs. The easiest place to put it is in the same directory as your class files.

**RCXRecv.java**

RCXRecv is the final piece of our first attempt at PC RCX communications. On the PC, RCXBean did several things for our communication program, including hiding the underlying data stream. On the RCX side, there is no RCXBean, so we work with the streams directly. RCXPort is the physical infrared port on the RCX. It has an InputStream that we make into a DataInputStream (if you check the docs, InputStream only has methods for reading bytes while DataInputStream provides methods for reading all of the primitive types). Then we simply wait for ints to appear on the IR link and display them on the LCD screen. The read is inside a while true loop that never exits normally. Instead we use try-catch and have the PC close the data stream. Closing the data stream throws an exception.

```java
import josx.platform.rcx.*;
import josx.rcxcomm.*;
import java.io.*;

class RCXRecv {
    public static void main(String[] args) {
        recv();
    }

    private static void recv() {
        try {
            RCXPort port = new RCXPort();
            DataInputStream in = new DataInputStream(port.getInputStream());
            while (true) {
                int val = in.readInt();
                LCD.showNumber(val);
            }
        } catch (java.io.IOException e) {
            LCD.showNumber(8888);
        }
        try {
            Thread.sleep(2000);
        } catch (Exception e) {
            System.exit(0);
        }
    }
}
```

---

9 Check out the code for RCXBean in the LeJOS source.
IOEndOfFileException, which is a derived class of IOException, and the code in the catch block is executed, the number 8888 is displayed, and then the program pauses for 2 seconds and stops running by calling System.exit.

That is all there is to this program. Note that a lot of the messy details, and communications is extremely messy, is hidden from you by the RCXPort class and the hardware-based routines it uses. The examples of the details are beyond the scope of this book but you can get an idea of how many things can go wrong by looking the source for the communications classes that we have used.
**RCX to PC Communications**

The code for going the other way, from RCX to PC, is very similar to the previous example. At right is the method for the PC to receive data from the RCX. Note that it uses a while true loop that won’t exit normally and relies on the RCX to close the data stream to break out of the loop. To make PCSend into a receiver, add this routine to your PCSend class and replace send() with recv() in the main method (note the name of the class is now somewhat misleading but won’t make the program not work).

```java
/* PC code for receiving */
public static void recv() {
    RCXBean rcxb = new RCXBean();
    try {
        rcxb.setComPort("USB");
        while (true) {
            System.out.println("PCC_m: " + rcxb.receiveInt());
        }
    }
    catch (IOException e) {
        System.err.println(e);
    }
}
```

The code at left is the RCX side for sending data. Copy this method into your RCXRecv class and replace recv() with send(sa) in the main method. Note that the send method is passed an array of ints as a parameter. This is the first step in making send into method that can be used in other programs.

```java
/* RCX code for sending */
static int[] sa = {10, 20, 30, 10, 5};
public static void send(int[] outarr) {
    try {
        RCXPort port = new RCXPort();
        DataOutputStream out = new DataOutputStream(port.getOutputStream());
        for (int idx = 0; idx < outarr.length; idx++) {
            out.writeInt(outarr[idx]);
        }
        out.close();
    }
    catch (java.io.IOException e) {
        System.err.println(e);
    }
}
```

After you complete the cut and pastes into PCSend and RCXRecv, compile and test the two programs. Remember that you need to compile PCSend with the J2SDK compiler and RCXRecv with the LeJOS compiler.

At this point, the question arises of what additional code is needed to get two RCXs talking to each other. The answer is none. If you have two RCXs, try running RCXSsend on one and RCXRecv on the other.
Excercises

1. Create a PCBean that mirrors the functionality of RCXBean. What methods in RCXBean are not necessary in PCBean? Describe PCBean in a UML diagram.

2. Add communications capability to your DanceBot. Write a caller program to tell DanceBot what step to do next. Define a protocol that allows DanceBot to understand each command.

3. For MapBot, add an upload function. When the robot has finished its mapping, have it upload its collected data when a PC-based program asks for it.

4. Create a new NavBot with communications capability. When MapBot returns, have NavBot signal it to upload the coordinates. Then have NavBot proceed to approach but not touch each of the obstacles that MapBot found. Repeat but with NavBot going to the closest, non-visited point each time.
Graphical User Interfaces – GUI

Flavors of Java

Java was designed to be the language of the Internet. It uses a virtual machine architecture so that once a Java program is compiled, that program can run on any computer that has a Java virtual machine, JVM, of the same version, regardless of manufacturer or operating system. What this means to you as a programmer is that you no longer have to worry about changing your program when a new operating system comes out. Highly paid systems programmers at places like Javasoft do that work for you, leaving you time to improve your program.

You are already familiar with two different flavors of the JVM, the one on your PC that came in the J2SDK and the one on the RCX that came in LeJOS. A JVM provides services, threads, file access, communications, etc., that are typical of an operating system. These two JVMs provide very different levels of service. For example, the RCX JVM does not provide methods to access files because on the RCX there is no disk to store files on. Many other methods are missing because there is simply not enough memory on the RCX to hold the code for these methods. Likewise, classes like RCXPort are not in the standard Java because RCX IR ports are very uncommon on general-purpose computers.

The point of this discussion is to highlight the fact the capabilities of programs written in Java depend upon the virtual machine that the program is running on. It may surprise you to learn that many computers have two JVMs installed and that the one that comes with the J2SDK is not the more common one.

Virtually every web browser has an embedded JVM. Like the LeJOS JVM, this JVM has severely limited capabilities. By default, programs running on this JVM cannot save or delete files. Programs cannot open connections to other computers. They are not allowed to run other programs. In fact, programs running on a browser’s JVM are not even called programs, they are called
applets. The reason for this is clear; on the Internet there are bad people writing programs. These people delight in causing other people grief by deleting files, crashing computers or disrupting mail service. They write programs called viruses or worms that spread throughout the Internet causing problems on thousands, or millions, of computers. The developers of Java made it extremely difficult to write these bad programs and spread them through web browsers.

A real strength of Java is writing graphical programs that run on the web. That area is beyond the scope of this book because applets cannot communicate with the RCX.

**Swing and AWT**

Graphical user interfaces, GUIs, are the preferred way for modern programs to communicate with human users. The standards for the appearance of window-based programs date back around 20 years, to the CUA standard. CUA required programs to have a certain “look”. A program needed a solid rectangle for its main window. Across the top is a bar for a title and an “X” that the user can click on to close the window. Main frames also have a menu bar located just under the title bar. The menu bar starts with the words “File” and “Edit” on the left and has “Help” as the rightmost entry. Under “File”, clicking on “Open” opens a file chooser box. This standard continues on for many pages, describing what programs should, and do, look like. This standard has become so pervasive that most windowing operating systems implement these elements, e.g. frames, file choosers, so user programs can call them directly instead of having to start from scratch for every program.

Java has two different sets of classes to implement GUIs: AWT and Swing. AWT, Abstract Windowing Toolkit, was introduced with the first version of Java. AWT relies on the operating system to provide display elements, such as frames and file choosers, by simply wrapping the operating system call in Java code (this is the same as forward() and backward() wrapping calls to ROM in MyMotor.java). This made it easier for the system programmers who created Java to finish the JDK. It also makes the Java applications look like any other application on a particular computer. The problem with this is that different computers have a
different look for all the elements and a Java program that looks good on one computer looks terrible on another. For that reason, making a Java based program look the same on all computers, Swing was born.

Swing moves a lot of the functionality provided by operating systems into Java libraries. It largely replaces AWT for writing new code, although the parts of AWT that don’t make operating system calls, like picking colors, are still used. The naming convention for elements that have been replaced is to prepend a J, e.g. Button becomes JButton.

Just to be clear, Swing is huge with hundreds of classes and thousands of methods. Each item you see on the screen, e.g. buttons, scroll bars, text fields, etc., has its own class in the Swing library. In the next sections, we will create a Swing based application that uses many elements. Note that using AWT elements in a Swing application where equivalent Swing elements exist, e.g. Button (AWT) and JButton (Swing), should not be done.
As usual, we start with “Hello World”. At right is GUI version that displays “Hello World” in the title bar of a frame. Three of the four lines are obvious. The first line creates a new Swing frame object. The third line sets the size in pixel, 400 pixels wide by 300 pixels high. The fourth line makes the frame appear on the screen (by default frames are not displayed because a you may want to wait until you have completed adding text and graphics before displaying your window). The second line is a bit less clear.

You should recall creating classes, GoHome.java and CenterBump.java, which implemented the SensorListener and TimerListener interfaces. These classes had methods that were called when a specific event occurred, e.g. when the robot ran into something, the stateChanged method was called in CenterBump.

MyWindowAdapter is the same sort of thing. It extends WindowAdapter and overrides the WindowClosing method. By default, WindowClosing simply makes the window, in this case our JFrame, disappear. It does not cause a program to stop running, which is the right thing to do most of the time. However, here we do want to stop the program so we override the method by extending the class and replacing the one method.

This is a good point to highlight the difference between implementing interfaces and extending classes. An interface simply specifies the names of methods that an implementing class has to define. GoHome had to have a method named “timedOut” because

```java
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class HelloWorldG {
    static public void main(String[] args) {
        JFrame jf = new JFrame("Hello World");
        jf.addWindowListener(new MyWindowAdapter());
        jf.setSize(400, 300);
        jf.setVisible(true);
    }

    static class MyWindowAdapter extends WindowAdapter {
        public void WindowClosing(){
            System.exit(0);
        }
    }
}
```
the TimerListener interface required it. It would be an error if GoHome did not define it. On the other hand, extending another class means that all the methods in the extended class exist in your new class. You can override existing methods by defining new methods with the same name but you don’t have to. WindowAdapter has 10 methods that implement four different interfaces that have to do with windows changing. By extending this class and overriding one method, we don’t have to deal with implementing all the other methods associated with those interfaces but those methods will be there and, we assume, will do something reasonable if they are ever called.
**HelloWorldG2.java**

The previous version, HelloWorldG, opened a frame and displayed “Hello World” but did not do it in a graphical way. This new version, HelloWorldG2.java, HWG2 for short, does. We’ll see how by examining the differences between the versions.

First, HWG2 extends JPanel. JPanel is container. You can put things into it and you can draw on it. HWG2 extends it so that we have a place to draw.

After creating an instance of HWG2, we have an instance to draw on and then we add it to the content pane of our JFrame so that it will show up on our screen when we make the JFrame visible.

Finally, we override the paint method from JPanel. Every time an element in a GUI is to be displayed, its paint method is called. The parameter to the paint method is the graphics object that the instance will paint upon. You can picture this as a tablet or canvas. In this example, we set the color to red and then draw “Hello World” at location (50,50), i.e. 50 pixels from the top edge and 50 pixels from the left edge of our graphics object.
**DummyMapData.java**

The program at right is a first step to graphing the map data that our robot has found. This class starts by extending HelloWorldG2. Doing so gives us access to our JPanel for drawing and our MyWindowAdapter class for keeping track of our windows.

We add one method to main, getData. In this dummy program, getData simply creates 10 points, XY pairs, by calling the random number generator, which returns a float with a range of 0.0 to 1.0, and multiplying by 100 to give us ints with a range of 0 to 100. It stores each XY pair in a Point object. These objects are then stored in a Vector. A Vector is like an array but more robust because it grows as you add things to it and the things you add don’t have to be of the same class.

Finally, the paint method has been changed. First, the size of the graphics object is found and axes are drawn to bisect it both horizontally and vertically. Then we use the iterator interface to retrieve the points in order. Note that we are not drawing points; we are drawing lines and a line is defined by its two end points. To initialize our line, we get an end point. But the end point of one line is the beginning point of the next. So we loop while we have another point, making the last end point our next beginning point, getting another end point and drawing the line that connects them.
MapData.java

This is the penultimate, next to last, program in this series. In this program, we combine receive data from the last chapter on communications with our map display program. We extends DummyDisplayMap so that we don’t have to replicate the paint method or the MyWindowAdapter class. Note that we are creating a new main method in each new class since our class name changes each time and we use the class name to create an instance of our class in main.

We have copied getData into this class definition. Java, as compared to C++, does not allow a class to extend more than one other class. There are reasons for this restriction that are beyond the scope of this book.

```java
import java.io.*;
import java.awt.*; import javax.swing.*; import josx.rcxcomm.RCXBean;

public class MapDisplay extends DummyDisplayMap {
    static public void main(String[] args) {
        JFrame jf = new JFrame();
        jf.addWindowListener(new MyWindowAdapter());
        getData();
        jf.setSize(400, 300);
        jf.getContentPane().add(new MapDisplay());
        jf.setVisible(true);
    }

    public static void getData() {
        RCXBean rcxb = new RCXBean();
        coorVec.clear();
        try {
            rcxb.setComPort("USB");
            while (true) {
                Point pt = new Point();
                pt.x = rcxb.receiveInt();
                if (pt.x == -1) break;
                pt.y = rcxb.receiveInt();
                coorVec.add(pt);
            }
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```
MapDisplayButton.java

MapDisplayButton is the end of our journey into Swing. This class adds a button so that you can upload data on command.

There are several new pieces in this class. First, a constructor has been added, the method MapDisplayButton. Note that this method does not have a type; since its name is the same as the class, it returns, without needing a return statement, an instance of an object of this class. The reason for having a constructor is that we are doing something that cannot be conveniently done in a static context, adding a listener to a button.

Creating a button is also new. It’s created at the class level so it is available to any method in the class. It is also an instance variable; every time a new MapDisplayButton instance is created a new JButton with the name of JB_recv is also created.

Creating the button is one thing; using it is another thing entirely. To be useful, a button needs a listener. SymAction is another listener class, just like SensorListener and MyWindowAdapter. Whenever a button is pressed, it generates an ActionEvent that is passed to every registered listener by calling the listener’s
actionPerformed method. In SymAction, we check to see if the source is the JB_recv button (since we only have one button, it is fairly certain that it is JB_recv). If it is, we get data from the RCX and then repaint our panel. repaint calls the paint methods for every component in our MapDisplayButton instance. That displays the newly received mapping data.
**Last Word on Swing, AWT and GUIs**

This is just the briefest possible introduction to Swing, AWT and GUIs. Graphic User Interfaces are the key to Java. If you are at all serious about programming, the next step is to get an IDE with a GUI builder. The major products in this category include Sun’s Forte, Borland’s JBuilder and IBM’s Visual Age. There are versions of these products that are free and other versions that cost thousands of dollars. And, you do get what you pay for.

You can build GUIs without a GUI builder. However, keeping track of the code is tedious, at best. GUI code gets very big and very ugly. By that I mean, each button you add to your application requires a minimum of 4-6 lines of code. Fifty buttons generate 200-300 lines of code with very slight, but significant, differences. Maintaining and updating such code is hard.

A GUI builder is based upon the principles of RAD, Rapid Application Development. You start with a blank screen and a palette of tools. You click on a tool and place the selected object, e.g. JFrame, JButton, JTextField, etc., where you want it. The GUI builder then generates the source code (a text file that you can edit), including listeners, for you to compile. If you want to move objects around or change their sizes, you do so using drag and drop techniques.

One technique that I have found useful is to never edit the files the GUI builder produces. Instead, I extend the class and put all of my routines, like the getData method in the last example, into the new class. The reason for this is clear the first time the GUI builder eats the code you haven’t backed up.

In closing, notice how each class built upon the last class. At each step, we made sure our code did what we expected it to. Then, we added new code knowing that our old code did what it was supposed to. This is a debugging technique that is very powerful. It is also in tune with a fundamental philosophy of Java; code once, use many times.
Exercises

1. Write a GUI program that extends the caller program you wrote for DanceBot in the previous chapter. Use buttons in your GUI to call the dance steps interactively.

2. Work in teams or with the whole class to develop a protocol to allow multiple robots to dance at the same time. Optionally, extend the protocol to allow different robots to perform different, hopefully complementary, steps simultaneously. Extend the GUI program to use a radio button group or combo box to select individual robots or the entire group.
The Java Language

The Java Language Reference (2nd ed.) is the defining document for the Java language. Most beginning programming students expect such a document to be totally beyond them. That expectation is wrong. The JLR is a mix of computer science jargon and some really clear English descriptions. It is not a book to read from cover to cover, in order. It is a reference book that you use to answer questions as you learn to program. The JRL has 18 chapters which we will describe very briefly in the following sections. Some we will basically skip while others we will examine in some detail. The 18 chapters are titled:

1. Introduction
2. Grammars
3. Lexical Structure
4. Types, Values and Variables
5. Conversions and Promotions
6. Names
7. Packages
8. Classes
9. Interfaces
10. Arrays
11. Exceptions
12. Execution
13. Binary Compatibility
14. Blocks and Statements
15. Expressions
16. Definite Assignment
17. Threads and Locks
18. Syntax

The JRL can be purchased in book form or can be downloaded from java.sun.com in either html or pdf format. The following sections are a brief introduction to the JRL. While there are many books on Java, including this one, remember that it is always best,
Introduction

The introduction provides sample programs, a note on the notation used in the rest of the book, a list of special classes and a list of reference works. These special classes include Object, String, Thread, Class, ClassLoader and some interfaces. Remember that a class is the Java representation of an object.

Grammars

As with English, grammar is the set of rules that determine the validity of an expression or sentence. For example, in a natural language there is a rule that a sentence must have a noun and a verb to be a valid sentence. Computer languages have similar rules. It is not necessary to understand how computer grammars work to be able to effectively program.

Lexical Structure

```
/* Hello World Example */
class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Lexical structure refers to the Java source code which is the text that you type in. The elements of lexical structure include comments, identifiers, keywords, literals, separators, operators and line terminators, white space.

Lexical analysis is the process of determining the lexical structure of the source code. What that means is figuring out what each character in the source code means. This is a two part process. First the source code is broken up into a list of tokens, where a token is simply one of the lexical elements, e.g. identifier, operator, etc. Each token is delimited, surrounded by, white space,
After each token is found, parsed, a lookup is performed to determine which kind of lexical element the token is.

Looking at the sample program at left, there are concrete examples of many lexical elements. The program starts with a comment on line 1. A comment is, essentially, a single token and anything inside of the comment delimiters, /* & */, is ignored by the compiler. Line 2 has three tokens: a keyword, class, an identifier, HelloWorld, and a separator, {. Line 3 has 11 tokens. Line 4 has 9 tokens while the last two have 1 token each. Lexical analysis is not something that you will be doing as a programmer but learning the rules is very important.

Unicode

Java uses Unicode for storing character data, including the source code. You are probably familiar with ASCII, American Standard Coding for Information Interchange, the traditional character set for small computers. Standard ASCII consists of 128 characters. This is all the characters on your computer keyboard, including upper case and lower case letters, numerals and punctuation characters plus a few more. Since computers are digital devices, there are no letters inside a computer; there are only numbers. ASCII is a mapping which makes equivalence between symbols and numeric values. For example, ‘A’ is 65, ‘B’ is 66, ‘a’ is 97, ‘2’ is 50, etc. As it turns out, for English ASCII works well since there are less than 100 distinct symbols used. However, computers are no longer English-only devices. Languages like French and German have additional characters like ‘ç’ and ‘ë’. Languages like Russian, Hebrew and Arabic have entirely different alphabets while Chinese does not have an alphabet at all. The additional characters make 128 a ridiculously small number. Unicode is a superset of ASCII and has room for 65536 distinct symbols. The first 128 characters of Unicode are the same as ASCII, while the remaining 65408 characters contain a variety of other symbols. In Java, most things use only the ASCII subset of Unicode. But comments, identifiers and string literals can contain any Unicode character that you choose.
Comments

Java, like C, has two types of comments, block and end of line. A block comment starts with a /* and ends with */. Everything in between the delimiters is ignored by the compiler. You can put any text you like inside a comment except */. That is, /* this is a comment */ is a valid comment but /* */ this is a comment inside of a comment */ */ is not valid. This concept is called nesting and in Java, comments do not nest. Block comments are also used for Javadoc, which is used to document source code. A later section will describe Javadoc in enough detail for it to be useful to you.

End of line comments are very simple. They start with a // and end at the end of the line.

Identifiers

Identifiers are the names that you create for classes, methods, properties, etc. In the Hello World example, the class name HelloWorld is an identifier. In Java, an identifier begins with a letter, either upper or lower case, and is followed by letters, digits, “_” or “$”. Note that operators and separators act as delimiters and so cannot be part of an identifier.

Keywords

Keywords are the identifiers that make up the Java language. Java has 48 reserved words, although it uses only 46 of them. It may seem surprising but Java does not include functions to read from the keyboard, write to the screen, manipulate files, do math functions or many other things. That functionality is included in the standard packages, which will be introduced later.

<table>
<thead>
<tr>
<th>class and method definition or declaration</th>
<th>abstract, class, const, extends, final, implements, import, interface, native, package, private, protected, public, static, strictfp, synchronized, throws, transient, void, volatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive variable types</td>
<td>boolean, byte, char, double, float, int, long, short</td>
</tr>
<tr>
<td>flow control</td>
<td>break, case, catch, continue, default, do, else, finally, for, goto, if, instanceof, return, switch, throw, try, while</td>
</tr>
<tr>
<td>object manipulation</td>
<td>new, super, this</td>
</tr>
</tbody>
</table>
The keywords are in this table, categorized by function. There are four basic categories: defining or declaring classes, methods and properties, specifying variable types, controlling the program flow and using objects. Note that primitive variable types are used in method declarations if a method returns a value. The two reserved words that are not used in Java are const and goto.

Operators

Operators are special characters that are usually used to manipulate primitive variables. There are 37 different operators in the Java language, many of which you will never use. The most commonly used operators are assignment, \( = \), equals, \( == \), the four math functions, \( + \), \( - \), \( * \), and \( / \), logical or, \( || \), and logical and, \( && \). For every simple operation, like \( + \) or \( - \), there is a corresponding combined operation and assignment, \( += \) or \( -= \).

Literals

Literals are values for primitive types and Strings. The best way to explain what literals are is to give examples. Integer literals are numbers without decimal points, e.g. 0, 33, -157, 1234321, etc. Real literals are numbers with decimal points and, optionally, exponents, e.g. 3.3, -47.0, 5E03, etc. Character literals are single characters surrounded by single quote marks, e.g. ‘a’, ‘D’, ‘?’ etc. There are two boolean literals, true and false. String literals, the only non-primitive type with literals, are zero or more characters surrounded by double quotes, e.g. “string”, “ another string “, “”, etc. Finally, there is one literal for reference types, null, and it is used to indicate that a reference variable is not referring to anything.

Note that this is an introduction to literals. There are many variants beyond what is described here so you should refer to the JRL for more information. However, this introduction should be adequate to get you started programming Java.
Separators

Separators are the final elements of lexical structure. There are nine separators in Java and six come in pairs, { and }, ( and ), and [ and ] while the other three are ; , , and . All of them act as delimiters. The squiggly brackets are used to delimit a block of code, creating a block statement. The parentheses delimit the parameters for a method. The square brackets delimit the index of an array. The semicolon delimits the end of a simple statement. The period connects instance variables with member methods or properties. The comma is used to delimit parameters in method calls. Examples of what this means will follow.

Types, Values and Variables

Variables are things with names, identifiers that are used to hold other things. The things that variables can hold come in two flavors, primitive values and references to objects. Java is a strongly typed language. That means that when you declare a variable you must also specify its type.

Primitive Types

There are 8 primitive data types: boolean, int, short, long, byte, float, double and char. These 8 types break into 4 groups: logical, integer, real and character. Logical variables hold one of two values: true and false. Logical variables are used almost exclusively in program flow control statements. That is, if a condition is true your code does one thing and if it is false your code does something else. Integers are natural numbers and include the byte, short, int and long types. byte is the smallest, 1 byte, of the integer types and has a range of -128 to 127. short is next in size, 2 bytes, and has a range of roughly -32000 to +32000. int is the next in size, 4 bytes, and has a range of roughly -2 to +2 billion. long is the biggest integer type, 8 bytes, with a range of roughly ± 10^{20}. Integers are used for counters in loops and as properties in objects. Real numbers are numbers that have a decimal point. Real variables come in two sizes, float, 4 bytes, and double, 8 bytes. float is accurate to about 7 significant digits while double has roughly 16 significant
digits. Real variables are used primarily to hold data about the real world. Finally, char is used to hold a single character. In Java, char is a Unicode character which means that not only can it hold a single Latin character, it can also hold characters from a variety of non-English alphabets and Chinese and Japanese characters.

Reference Types

In previous sections, we discussed objects at length. Creating an instance of an object without having a place to hold it would be pointless. Reference type variables are used to hold references to created instances of objects.

Conversions and Promotions

This section describes what is and is not allowed when attempting to make assignments between variables of different types. In general, such assignments are not permitted. Within the integer and real groups, assignments between variables of different types are allowed but if the assignment is from a larger sized variable to a smaller sized one (narrowing), it may cause an error. For example, if you try to assign 1000000 to a short integer variable, that causes a silent error because variables of type short can only hold values from roughly -32000 to 32000. Likewise for reference types, between classes in the same family tree, it is possible to assign an instance of a derived class to a variable of the super class. For example, any object can be assigned to a variable of type Object, since Object is the starting point for all classes. And it is possible to assign a variable of type Object to a variable of its own class, provided you specify which class it is. But it is not possible to hold references to instances of unrelated classes in reference variables.

Names

A name is an identifier that has been declared to be of a specific type. A name has scope; a name is defined only within the block in which it is declared.
There are conventions for how to create names. Nothing enforces these rules but violating them indicates that you are not a “good” Java programmer.

1. Class names are concatenated nouns with the first letter of each capitalized and the rest lowercase, e.g. MyRobot.
2. Method names are verb phrases with the first word all lowercase and the rest initial uppercase, e.g. startMyRobot. Methods that change or test properties should use “set” or “is” as the first word of the phrase, e.g. setPowerLevel, isRunning.
3. Field names are concatenated nouns with the first word all lower case and the rest initial uppercase, e.g. powerLevel. Care should be taken to not obscure method names with field names.
4. Constants should noun phrases with each word separated by an underscore, “_”, and all uppercase, e.g. MAX_POWER.
5. Local variables should be short and meaningful. An acronym is good, e.g. mr for a variable of type MyRobot. Shortening of words in noun phrases is good, e.g. buf for buffer or out for output. One character names should be avoided.

Packages

A package is a set of classes that share a common theme. Most of Java’s functionality comes from the standard packages which are java, the core java package, and javax, the java extensions package. The Java language is tailored to different environments, PCs, microwave ovens, etc., by using different versions of the standard packages. LeJOS replaces the standard java package with its own for use on the RCX. The replacement package is significantly smaller with far fewer methods than the standard package because the RCX has extremely limited memory. Yet, it is still Java that is run on the RCX. Later, we will compare the java package from the JDK with the LeJOS java package.

Classes

A class is Java’s way of representing an object. All classes are derived, either directly or indirectly, from the class java.lang.Object. Classes have two kinds of members: methods and fields. Methods are functions or subroutines that are related to the object. Fields are attributes or properties of the object. Both methods and fields can be either instance or class. Classes can be derived from other classes, extending the class.
There are several reserved words that act as modifiers for class, method and/or field declarations. These reserved words are `abstract`, `extends`, `final`, `implements`, `native`, `private`, `protected`, `public`, `static`, `strictfp`, `synchronized`, `throws`, `transient`, `void`, and `volatile`.

`extends` and `implements` modify only class names. `extends` declares that a new class inherits all of the methods and fields of another class. In previous examples, the class `BassetHound` extends the class `Dog`, e.g. `class BassetHound extends Dog`. A class that extends another class almost always adds new methods or fields and/or overrides existing methods or fields. A class that extends another is called a `subclass` while the class that is extended is called the `super-class`. Note that any method that can be called from an instance of a super-class can be called from an instance of the subclass. `implements` is used to specify that a class has all the methods and fields that are specified in a special type of class called an interface. There are two major differences between extending and implementing. First, a new class can extend only one other class while it can implement as many as needed. Second, methods from a super-class already exist whereas an interface requires the programmer to write the definition of the methods.

`abstract` is a modifier for class that creates a hybrid between a super-class and an interface. An abstract class has some methods that are declared but not defined. The programmer must write the declarations for the abstract methods. But there usually are non-abstract, i.e. declared, methods in an abstract class so it is different from an interface.

`private`, `public` and `protected` are modifiers that refer to where classes, methods and fields can be used and where they cannot be used. Something declared public can be used anywhere. Something declared private can be used only within the class where it is declared and nowhere else, not even in subclasses. Something declared protected is between public and private; it can be used only within the class where it is declared and within any subclasses. The convention in Java is to make most fields private and most methods public.
static modifies classes, methods and fields. It is used to say that there is only incarnation of whichever type of thing it modifies. For example, Math is a static, Java class. Ponder what an instance of Math would be. All the methods in the class Math are static, e.g. Math.cosine, Math.log, etc. Likewise, Math.PI is the value of pi. These are all things that you only want one of. In LeJOS, Motor.A, Motor.B and Motor.C are all static variables and refer, respectively, the motors attached to ports A, B and C. static fields or methods are also called class fields or methods. This is in contrast to instance fields or methods, which is the default. In a program, there is exactly one incarnation of a class field or method whereas there is one incarnation per instance for non-static fields.

final says that whatever it modifies cannot be changed. A final method cannot be overridden. A final field cannot have its value changed. Math.PI is an example of a final field. Unless you change universes, it would be unwise to change the value of pi.

transient and volatile indicate a field that changes. A transient field is one that is not stored when its instance is stored. A volatile field is one that can have a new value assigned to it outside of your program. It is very unusual for a beginning programmer to encounter either one.

native indicated that a method is object code for the real machine your program is running on and not Java byte code. You cannot declare a method native and then write code in Java to define it. You must, instead, write the method in another language, like C++ and then perform magic to make it work in Java. Note that native code is used in only a very few places, like the JVM, and makes a program non-portable. A beginning programmer should never declare native methods.

strictfp requires classes or methods to use special libraries for real number manipulation. Ordinarily, floating point numbers, float and double, use the computer’s hardware math routines. This means that a JVM running on a MS Windows machine may produce slightly different results than a Sun or Macintosh for floating point numbers. The differences are usually very, very small and of no real consequence. However, there are some cases where the results must be identical. In that case, specifying strictfp causes the JVM to use software math routines instead of the computer’s hardware routines. The causes the program to produce identical results...
without regard to the computer you use at the cost of, usually, much longer execution time. A beginning programmer will never need to use strictfp.

synchronized indicates that a method must be executed without interruption. Java is a multi-threaded environment. At any time, the JVM can stop executing one thread and start another. Obviously, when the JVM does this, it saves enough information so that it can resume the stopped thread without any problem. Usually, this method is good enough. However, there are times when a method cannot be interrupted in this manner. In those cases, declaring a method synchronized guarantees that the body of the method will not be interrupted. Note that you should be very careful in declaring methods synchronized because synchronized methods interfere with the normal scheduling of the JVM, which may result in truly obscure and difficult to debug errors.

throws is used to declare which exceptions a method can throw. Any method that throws an exception must be placed inside a try {} catch () {} block.

void indicates that a method does not return anything.

**Interfaces**

Interfaces are a special kind of class. In Java, a class can be derived from only one parent class. However, a class can implement as many interfaces as needed. An interface is simply a specification that guarantees that a class contains methods and/or fields with specific names and signatures.

**Arrays**

An array is simply an ordered list of things. The things in an array can be any of one type, either primitive or reference. A specific element in the array is accessed by supplying an index integer.
Exceptions

An exception is an unusual or unexpected event that interferes with the normal flow of a program. Dividing by zero is a classic example. After dividing by zero, it is fairly clear that your program should not continue running as if nothing happened. But what should it do? Java is an unusual language in that it allows a program to detect and deal with almost any unexpected event instead of having the system simply kill the program.

Java divides these events into two classes: errors and exceptions. Errors are events that are totally unpredictable or could happen in some many places that constantly checking for them is impractical and how to recover from them is unclear. Running out of memory is an error. Events that are predictable are exceptions. Java uses exceptions for events such as end of file while reading in a file. Exceptions are handled using a try {} catch () {} set of blocks. Programs can generate their own exceptions using a throw statement. Exceptions and exception handling adds a get deal of flexibility to Java for dealing with unusual events.

Execution

Java is a very secure language. In order to achieve that security, there are very specific rules on executing Java programs on different. The JRL defines those rules. Most beginning programmers will not need to be concerned about these specifics.

Binary Compatibility

Binary compatibility is the ability of a single executable program to be run on a variety of different computers. Java, through the use of a virtual machine, maintains binary compatibility on all supported platforms. The JRL description of binary compatibility is extensive and not especially necessary for a novice programmer to understand.
**Blocks and Statements**

A block is one or more statements surrounded by curly braces, “{” and “}”. A variable is local to a block if it is declared inside the block. The scope of (it is accessible) a variable is the block that it is declared in and all sub-blocks of that block.

A statement causes something to happen and does not have a value. Statements are delimited by a trailing semicolon. Java has 13 distinct statements, most of which correspond to reserved words. The two that do not correspond to reserved words are the empty statement and the expression statement. An empty statement is simply a semicolon preceded by nothing but white space. An expression statement is simply an expression, which may be an assignment, a method call or an increment operator, followed by a semicolon. The remaining 11 statements are `if / if else`, `switch`, `while`, `do`, `for`, `break`, `continue`, `return`, `throw`, `synchronized`, and `try catch / try catch finally`. These statements all control the flow of execution of the program, that is, they determine which statement to execute next. We will not describe the syntax of these statements now. Instead, as we continue programming their usage will be clear.

**Expressions**

An expression is something that evaluates to a value. This includes numeric literals, e.g. 3, method calls, e.g. `mr.getPowerLevel()`, mathematical operators, e.g. 3 * 4, among other things. A variety of operations and expressions will be introduced through the programming exercises. Listing them here would create a long and boring list. You are encouraged to look through the JRL if you are interested.

**Definite Assignment**

Definite assignment is simply the requirement that every variable declared in Java must be assigned a value before it can be used. That seems like a reasonable requirement because there are exceeding few instances when your program would want to use a
variable before it had explicitly assigned a value to it. Almost always, using a variable before having assigned it a value is an error. Having programmed in languages that don’t have this requirement, I can assure you that it is an error that can be extremely difficult to find. The JRL devotes many pages to this topic because from a computer science perspective, it is very important and very hard to prove that a variable really has been assigned a value. For a beginning programmer, definite assignment is not an important consideration. If you do forget to initialize a variable, you will get a compiler error and be required to correct it before your program will successfully compile. The simplest thing to do is whenever you declare a variable, give it a 0 or null value, e.g. \texttt{int intvar = 0;} or \texttt{MyMotor mm = null;}. This is a good habit to get into because this kind of habit makes it harder for you to make a coding error. The best programmers have all developed good habits like this and so should you.

\textit{Threads and Locks}

A thread is separate execution path. When you start a Java program, one thread is created. Your program can start additional threads. For example, a robot may start one thread for each sensor it has. This is a very efficient method of programming a robot.

Threads all share the same memory space. For example, \texttt{Motor.A} is a static variable that controls the motor attached to port A on the RCX. All the threads in your program have the ability to change \texttt{Motor.A}. This could cause problems if, for instance, the thread for one sensor was turning the motor on forward and another sensor was turning it on backward.

Java provides a mechanism of locks based upon methods, \texttt{wait}, \texttt{notify} and \texttt{notifyAll}, built into \texttt{java.lang.Object}, the super-class of all classes. These locks allow one sensor’s thread to take ownership of the motor for a long enough time to complete its task and then yield to another sensor’s thread so the second thread can do what it needs to do.

Threads and locks are one of the most powerful and advanced features built into Java that really distinguishes Java from other languages. Understanding how this works is best done through code examples for the robots.
Syntax

The last chapter presents the BNF, Baccus Naur Form, of the Java grammar. To computer science students wanting to write a parser and lexer for Java, the chapter is quite interesting. For most programmers, you might learn something from reading it but you might not.