Increasing Accessibility to a First-Year Engineering Course in Mobile Autonomous Robotics

John C. Gallagher [1,2], Richard F. Drushel [3], Duane Bolick [1]

Department of Computer Science and Engineering [1]
Department of Electrical Engineering [2]
Wright State University
{dbolick,jgallagh}@cs.wright.edu, rfd@po.cwru.edu

Department of Biology [3]
Case Western Reserve University
rfd@po.cwru.edu

Abstract

Introductory classes in the design and programming of mobile autonomous robots offer both potential and matriculated engineering students entertaining and engaging educational experiences that give them early experience with the kinds of open ended design problems they will face in their professional careers. By their nature, however, these classes often require some prior computer programming experience – which raises the threshold of entry to the very early career students who might most benefit from the extra motivation and depth provided by dealing with open-ended problems. In previous work we discussed minimizing dollar cost and maximizing physical access to a robot by creating a WWW/web cam based infrastructure and supporting open sourced robot simulation software. In this work, we will focus on additional work that addresses more fundamental pedagogical issues including ease of collaboration among geographically dispersed students and the design of educational materials more suitable for maintaining low threshold, high ceiling educational experiences for the students.

1. Previous Work –WWW Autonomous Robotics

Formal knowledge based classroom instruction is necessary for the education of engineers. However, it also requires practicum components in which students can experience both the joys and frustrations of actual design, implementation, and testing in an environment rich with possibilities and with the guidance of experienced mentors. Generally, design practica occur toward the end of a student's undergraduate career. This is for good reason – many interesting problems require mastery of a significant body of knowledge to be approachable. On the other hand, many students receive enormous benefit from engaging in these design practica early in their undergraduate years. Not only do they get an early look at what real engineers do early on, they also are shown quite clearly why all the knowledge presented in the other courses is so valuable in practice. Many students who might otherwise drop out of, or never enter, engineering programs can thus be motivated to not just stay – but seriously apply themselves – in their traditional classes.

In previous work [1 - 3], we addressed issues related to offering an autonomous robotics course over the World Wide Web (WWW). By constructing a centralized environment containing a web connected mobile robot and providing 24/7 access to it via the Internet, we were able to leverage a single, expensive, robot to serve the needs of geographically dispersed students. By providing an open sourced Java based robot simulation environment, we were able to relieve potential bottlenecks caused by many students testing early versions of their robot controllers on a single robot. Also, we were able to provide these simulators, which run on nearly any modern microcomputer, free of charge. This further relaxed financial burdens in running the course and increased access to many under represented demographics. Most of our past work has focused on the technical nuts and bolts of getting the system running reliably and in maintaining sufficient fidelity between the simulation code and the actual robot. The point then was to increase physical access to facilities to many demographics that otherwise not be able to participate. In this paper, we will focus more on subsequent efforts to tune the pedagogy of the course material to lower the knowledge threshold of entry into the course and to increase access to early program undergraduates and advanced high school students. The paper will begin with a brief description of the course, the attendant infrastructure as it existed, and a few items that were less than well handled. It will then discuss specific changes that have been made to that infrastructure to deal with those issues and better support our proposed pedagogy. Following will be a discussion of specific educational materials developed to lower the knowledge threshold for participation. The paper will conclude with a brief discussion of future plans and open issues.

2. Previous Work – Access Issues

In our original class, students developed robot controllers to solve a series of increasingly difficult problems on a mobile robot simulator that we designed and implemented using Java. When finished, they upload their controllers to a real robot in our lab and observed the results via a WWW web cam. Students kept an engineering journal and were graded on the quality of their design/implement/test processes. Except for the remote and geographically distributed nature of the course, it was not in principle much different from more traditional robotics practica offered in person [4 - 7]. Though careful study of the differences between the two methods of offering the material is still underway, two issues with the online environment quickly became apparent. The first was that the activation energy just for a student to get his/her personal environment set up was so high that actual interesting work was delayed far too long. In traditional offerings, a TA or instructor can pre-configure machines and infrastructure so that a student can get to the meat of the study problems immediately. In a distributed and/or online environment – such support is not generally available as students are generally far removed from the instructor. Though our software was fairly easy to install for the experienced student, many in demographics of interest had difficulty. This can not be ignored. The second issue had to do with quality of Internet mediated student-to-student and mentor-to-student interactions. Generally, students and/or mentors would discuss solutions and tests with one another via standard text chat tools. Most students, perhaps already comfortable with such communication due to the growing ubiquity of cell phone based text messaging, were quite satisfied with that

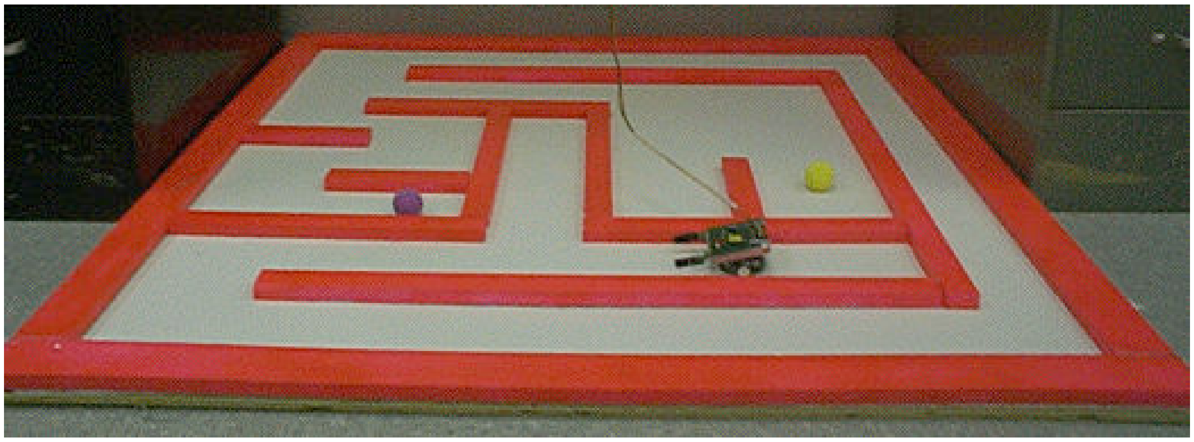


Figure 1: An Internet Connected Khepera Robot
This snapshot was taken from a WWW browser using the student accessible webcam. Users can pan, zoom, and tilt the camera remotely to get more detailed views as needed.

mode of communication. However, students experienced difficulties in textually describing to other parties what their robots were doing in the world. It was possible to upload code to the real robot and have multiple people simultaneously watch its operation. However, this created an undesireabile resource bottleneck. Extensions that allowed collaborative viewing of the robot simulation environment across the Internet were sorely needed. The first issue limited access via a high knowledge barrier. The second issue limited access to collaborative debugging because of a hardware bottleneck. Before considering how these issues were solved, we will briefly describe the basic components of the simulation and remote control robot environment.

3. The Physical Robot and Its Environment

The environment we created in our lab consists of a single robot that operates within a 4x4 foot enclosure. The robotic hardware consists of a standard Khepera robot [8, 9] equipped with an auxiliary gripper arm module. Communication with the robot is facilitated through a wire tethered between the robot and a host machine's serial port. The robot is manipulated using software that writes/reads data to/from the robot via interpreted commands from the user. Within its enclosure, the robot may be confronted with a simple set of obstacles: reconfigurable walls (typically in the form of mazes and/or rooms), lights, and plastic soda pop bottle caps. Wall sections and lights are fastened to the floor of the enclosure, and therefore cannot be moved by the robot. Caps are used specifically as objects to be manipulated by the robot via its gripper attachment. It is this particular environment that is simulated by our software. Due to the environment's simplicity, the task of developing sensor and actuator models was significantly reduced. The color and reflective properties of the obstacles were specifically chosen so that sensor response would be similar at given distances from an obstacle regardless of its type. These properties along with the constant lighting in our lab provided the basis for the accurate yet efficient models eventually used within the simulator. The software architecture used to interface the hardware with student written controller is described more completely in [3]. Here it is sufficient to note that the students actually program a "virtual robot object" that is tightly

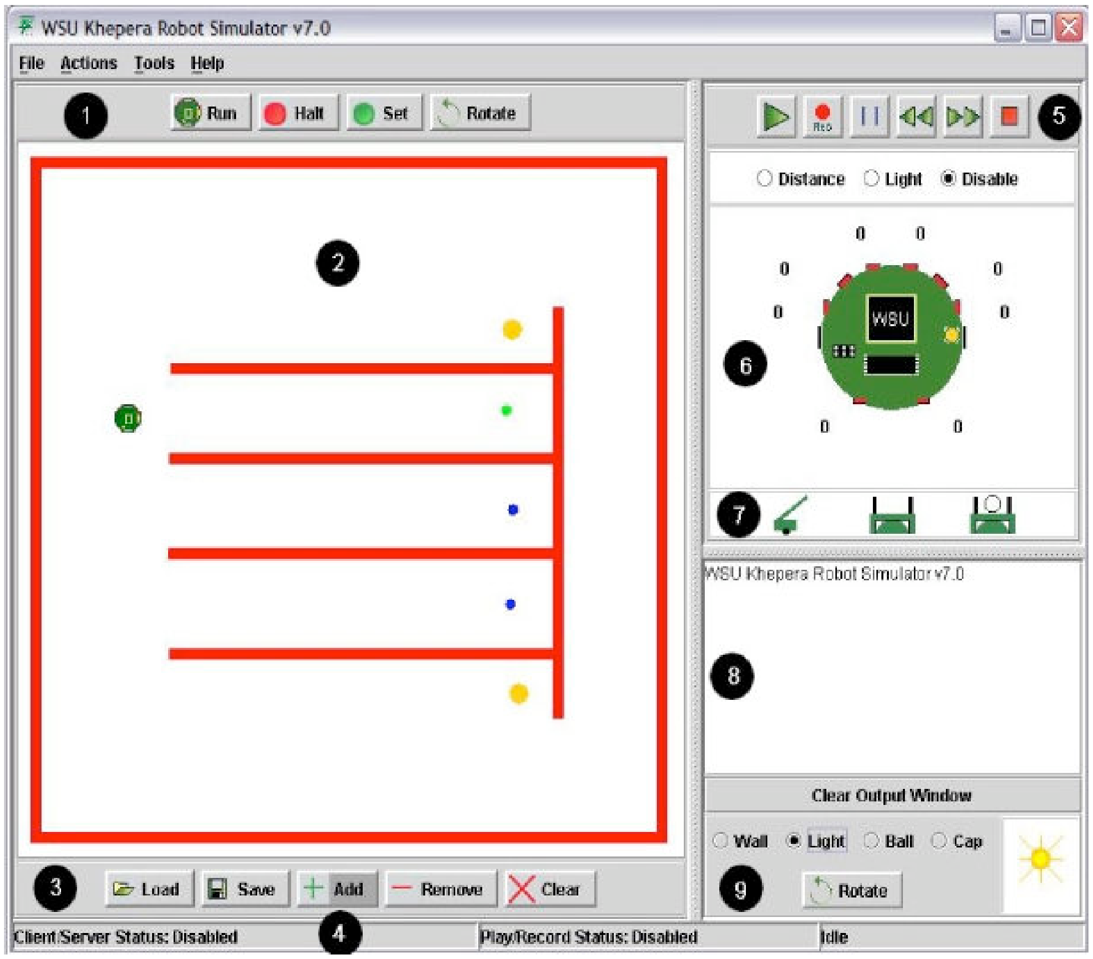


Figure 2: Ksim User Interface
This current simulator user interface. User interface elements include the Robot Control Panel[1]. The World Panel[2], World Control Panel[3], Client/Server Status Bar[4], Record and Playback Panel[5], Sensor Display Panel[6], Gripper Arm Status Panel[7], User Output Panel[8], Object Selection Panel[9]

coupled to the actual state of the real robot. This allows the students to have the illusion that they are controlling the raw hardware while providing a hidden supervisory layer that prevents accidental or purposeful damage to the robot hardware. Considering that the robot will be unattended for most of the time it is used, this is vital. Also, the use of the virtual robot object eases the transfer of controllers from simulation to real robot so long as the simulation, of course, provides an identical controller interface.

4. The Simulation and New User Interface Elements

The details of the basic simulation are likewise discussed in [2] and [3]. A great deal of effort was put into making the simulation highly portable to all common microcomputers and in making it possible to transfer robot control code unchanged from the simulation to the actual robot. For purposes of this paper, however, we need only focus on the user interface – as this is where changes required to fix the collaboration bottleneck were needed. Figure 2 shows the current user interface. Labeled interface components have the following functions:

Interface Element	Purpose/Use
Robot Control Panel	Allows a user to position the robot and start and stop control code execution.
World Panel	A bird's eye view of the simulated maze environment. This is updated real time as the simulation runs.
World Control Panel	Allows a user to load, save, and edit world configurations. Wall, cap, and light positions may be edited.
Client/Server Status Bar	Displays the current status of the robot movie recording system and Internet display sharing.
Sensor Display Panel	Displays the values being returned by the eight IR distance sensors or the eight light sensors. This display is updated in real time as the robot runs.
Gripper Arm Panel	Displays the status of the gripper arm.
User Output Panel	Space for users to display text messages from their programs.
Record and Playback Panel	Allows users to operate a virtual VCR to record movies of robot operation.
Object Selection Panel	Allows users to select non-robot objects to place into the world.

The most newest and most relevant elements are the *Record and Playback Panel* and the *Client/Server* status bar. These two features were added in direct response to student needs for better collaboration from within the simulation environment. A virtual VCR was integrated into the simulation that allows students to record the actions of their robot. These "movie" files, which are ASCII text recordings of all relevant robot parameters taken at user specified time intervals, can be played back later from within the simulation interface. They can also be easily emailed to other students for playback in their simulators. Also, since the recordings are in ASCII text, they can be imported into other tools (text editors, spreadsheets, custom user code, etc.) to support advanced analysis of sensor inputs and/or motor output timings. This new facility allows students to maintain and share recordings of particular test runs for group debugging and analysis. Also added to the simulation was the ability for any particular simulation to go into a "server" mode where it can stream all robot parameters live to any other simulator on an Internet connected machine. To accomplish this, a user would simply use the menu bar interface to tell the simulator to go into server mode. He or she can then tell or email other remotely located users the IP address of his/her machine – and they can connect their simulations to the server to display what the server robot is doing live. This facility allows large groups of students to be watching the same robot activity live and is meant to support group analysis and debugging. Both of these features are available in the currently distributed simulator and have been extensively tested for functionality.

5. Easing the Pain by Lowering Threshold to Participate

Because of the inherently distributed nature of the class, there are just some things we won't be able to do collaboratively. Among these is the initial set up of each student's individual computing environment and coming to grips with just enough Java to be functional. We are currently developing a textbook that both eases the burden of this setup and provides clear explanations of requisite Java techniques. We have adopted a conversational, tutorial style that we feel well simulates the style of assistance that an "in person" coach or mentor would provide. Example pages of this text are provided as an appendix to the paper for reader consideration. Readers are welcome to use the appendix to help download and configure the simulation for use on their personal machines at the conference or later. Comments are appreciated and welcome. A full version of the text will be available for distribution in summer 2005.

6. Conclusions and Future Work

To date, we have expended a significant amount of effort in developing infrastructure in support of an online robotics practicum. We believe that as more engineering departments consider placing entire degree programs online, the issues involved with providing meaningful practica experiences through that medium will become more important. Also, engineering outreach is both to under represented demographics and pre-college students can be highly enhanced by allowing communities of interest to form autonomously and asynchronously. WWW practica experiences can well provide those opportunities at low financial cost. It is unclear, however, how the change in delivery method changes the pedagogy of these courses. Using first generation code we have identified and solved a few obvious problems, which have been discussed in this paper. What remains is to conduct careful studies of learning efficacy in teaching the same style of practicum course in both online and traditional environments. It is hoped that, armed with appropriate support structure, we can uncover and correct for what are perhaps more subtle differences that we have not yet seen or anticipated. These studies will be conducted over the next three years – with the first full offering of the course using these materials being coincident with the 2005 ASEE National Conference. The authors will be conducting the course from the conference with geographically dispersed students.

In addition to what we feel to be much needed studies of pedagogy, we intend to continue expanding and improving the simulator package and the robot/user code remote interface. Among issues to be addressed are increasing the accessibility of the interface to those with motor and/or perceptual disabilities. We have leveraged technology to help overcome barriers of distance. There is no reason that we cannot similarly use technology to allow access to valuable engineering experiences to those who might find it difficult to physically manipulate real robots. Also, we intend to expand the simulator to function with more than just the Khepera robot. The Khepera is capable, but relatively expensive. However, to ease adaptation and adoption at other institutions, we wish to maximize the choices of underlying hardware platforms. Course materials, as well as all software, are available at: http://ehrg.cs.wright.edu/ksim/ksim.html. All materials are open source and freely available for non-profit educational activities.

Acknowledgements
This work was supported by the National Science Foundation under grants DUE 0341263 and DUE 0341150. Additional support has been provided by Wright State University and the Ohio Board or Regents and the Howard Hughes Medical Institute.

Bibliography

1. Gallagher, J.C. and Perretta, S. "WWW Autonomous Robotics: Enabling Wide Area Access to a Computer Engineering Practicum", The Proceedings of the 33rd Technical Symposium on Computer Science Education. ACM Press (2002).

2. Perretta, S. and Gallagher, J.C. "A General Purpose Java Mobile Robot Simulator for Artificial Intelligence Research and Education", Proceedings of the 13th Midwest Artificial Intelligence and Cognitive Science Conference (2002).

3. Perretta, S. and Gallagher, J.C., "A Portable Mobile Robot Simulator for a World Wide Web Robotics Practicum", in *Proc. of the 2003 American Society for Engineering Education Annual Conference and Exposition*. ASEE Press.

4. Beer, R.D., Chiel, H.J., and Drushel, R.F. "Using Autonomous Robotics to Teach Science and Engineering", Communications of the ACM (June 1999). ACM Press.

5. CWRU Autonomous Robotics Course. Online. http://www.eecs.cwru.edu/courses/lego375/

6. Martin, F.M. A Toolkit for Learning: Technology of the MIT LEGO Robot Design Competition.

7. MIT 6.270 Autonomous Robot Design Competition. Online. http://www.mit.edu:8001/activities/6.270/home.html

8. K-Team (Khepera Info). Online. http://www.k-team.com/

9. Mondada, F., Franzi , E. and Ienne, P. "Mobile Robot Miniaturization: a Tool for Investigation in Control Algorithms", ISER'93, Kyoto, Japan, October (1993).

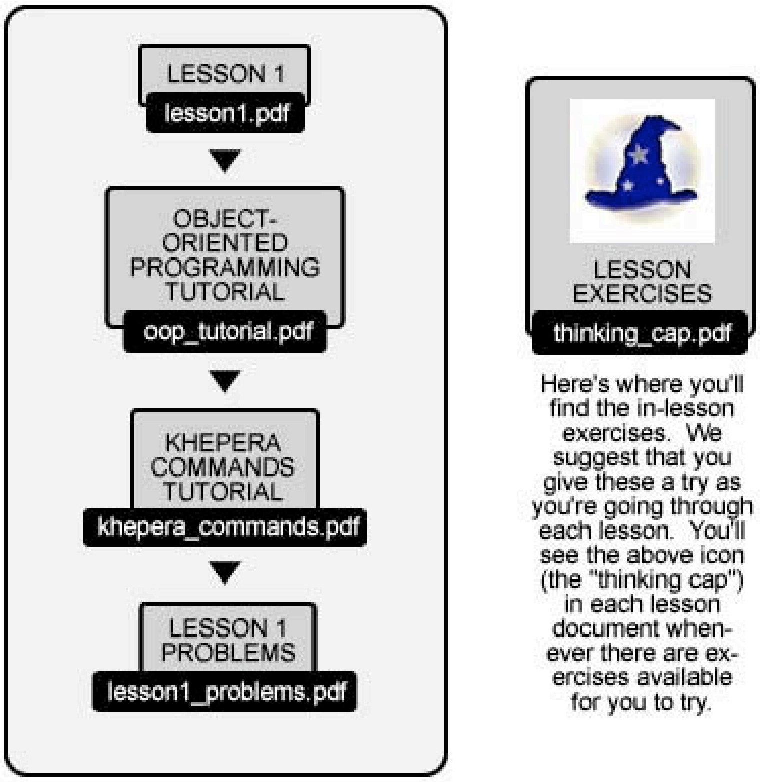
JOHN C. GALLAGHER
John Gallagher is dually appointed as an assistant professor in the both the Department of Computer Science and Engineering and the Department of Electrical Engineering at Wright State University in Dayton, Ohio. His research interests include analog neuromorphic computation, evolutionary algorithms, autonomous robotics, and engineering education.

RICHARD F. DRUSHEL
Dr. Richard F. Drushel is a Full-Time Lecturer in the Department of Biology, Case Western Reserve University, Cleveland, Ohio. He co-invented and has co-taught for 19 semesters a highly-successful LEGO- and microcontroller-based autonomous robotics course for undergraduates, as well as several summer courses for educators and secondary-school students. His research interests include 3-D kinematic modelling of soft-tissue structures in the feeding of marine molluscs, and the use of computers and robotics in education.

DUANE BOLICK
Duane Bolick is a Masters Student in the Department of Computer Science and Engineering at Wright State University. His interests are in autonomous robotics and robotics based engineering education.

Getting Started

Here's the path we suggest you follow as you go through the curriculum:



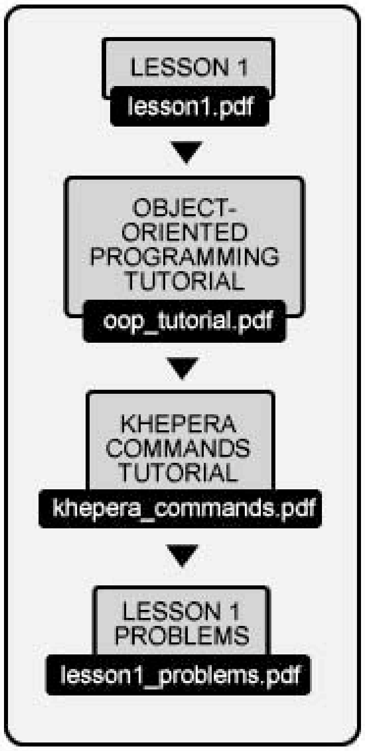
Here's where you'll find the in-lesson exercises. We suggest that you give these a try as you're going through each lesson. You'll see the above icon (the "thinking cap") in each lesson document whenever there are exercises available for you to try.

Once these are complete, you may proceed through the numbered lessons and the assigned problems for each lesson.

SYLLABUS

WEEK 1

Introduction and Background



These documents will be covered in the first week.

They all contain material necessary for understanding the rest of the lessons in this course.

WEEKS 2-7

Once we've covered the introductory material, we'll look at each of these topics, one per week, in this order.

TOPIC	FILE
WEEK 2: Sensors I	**sensors_I.pdf**
WEEK 3: Sensors II	**sensors_II.pdf**
WEEK 4: Gripper	**gripper.pdf**
WEEK 5: Behavior I	**behavior_I.pdf**
WEEK 6: Statefulness	**states.pdf**
WEEK 7: Behavior II	**behavior_II.pdf**

WEEKS 8-10

In the 8th week, we'll introduce your final project, and you'll spend weeks 8 and 9 working on it. In the 10th week, you'll present your final solution for the project in the form of a working controller source code file.

Assignment 1: Writing Robot Controllers in Java

First, let's talk about the basics...

"So, what is Java?"

Java is a high-level computer language, like C, Fortran, Perl, and many others. Compared to these "others," Java is most similar to C in its syntax. While knowing C or C++ will help you program in Java, it's not necessary to know either to learn how.

"OK, I know C/C++. This should be a snap!"

In this case, you're right - we won't be using any complicated syntax. Just variables, expressions, loops, conditionals, and functions. And those are pretty much all the same as C/C++.

You can probably get through this tutorial fairly quickly, but make sure you understand how to compile your controllers and how to test them.

"I've programmed before, but not in C or C++"

Most programming languages use the same concepts as we're going to, so your experience will help you out - we won't be using anything complicated (and if we do, we'll explain it first).

Like I told the C/C++ savvy crowd up there, we'll use the simplest of programming constructs. Just pay attention to how they're done in Java, and you'll be OK.

"I've never programmed before"

There's never a bad time to learn programming. We won't be going into tremendous detail here - You only need to worry about learning enough to make the robot do what you want it to do, and as we'll see, it won't be too bad.

Pay close attention to this tutorial, go through it a few times if you need to, and ask for help if necessary.

Without further delay, let's begin...

Answer: "Very Carefully"

First, here's what you'll need...

1 If you don't already have **Java 2 Standard Edition SDK, version 1.4 or higher,** you need it. If you're using Windows, Linux, or Solaris, you can get it for free from java.sun.com. We suggest getting the latest, non-beta version of the Software Developer's Kit (SDK). The SDK includes everything you need to compile and run Java. If you're running Mac OS X, the Java SDK is already installed.

2 You need a **text editor**. Any text editor will do (vi, emacs, Notepad, Wordpad, TextEdit, etc.) If you're looking for a text editor with neat features like syntax highlighting, we suggest jEdit, which you can download for free at www.jedit.org.

3 Once you've downloaded and unpacked/installed/whatever (depending on your OS and SDK version), you'll need to **add an entry to your PATH** environment variable that points to the /bin directory inside the main Java directory. For example, if installing the J2SE SDK puts a directory on your drive called "j2sdk1.4.2" look inside that directory to find the /bin directory where the Java tools, like the compiler, are located. You need to do this so that when you type:

```
% javac
```

or

```
C:\> javac
```

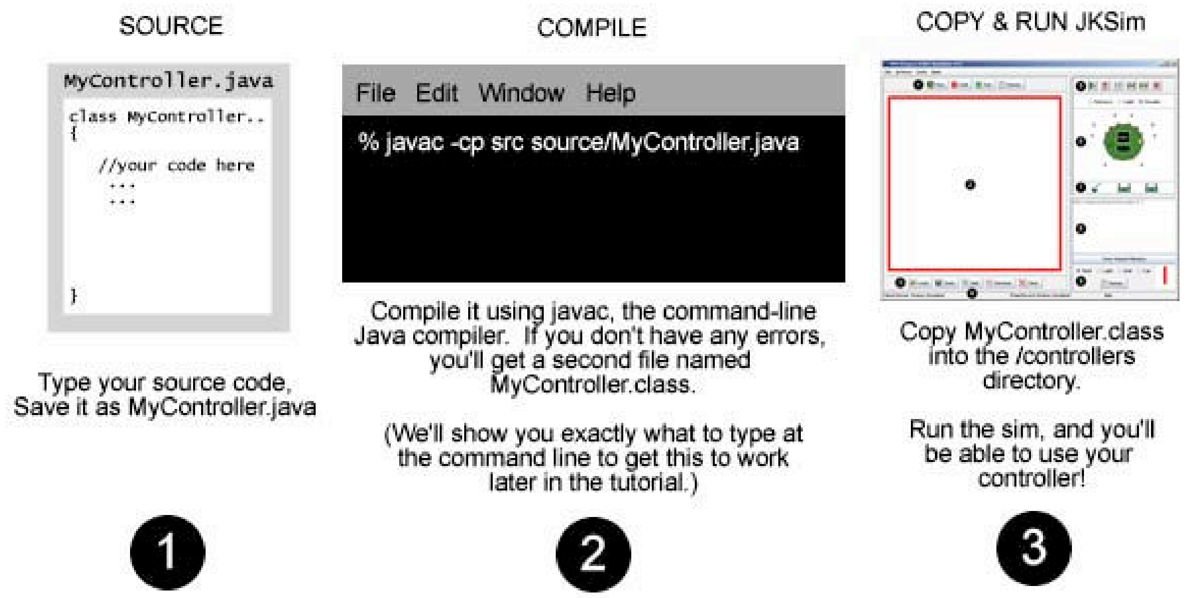
at the command line, your terminal will know where to find javac (the compiler).

4 Finally, you need the **Wright State University Khepera Simulator application** (KSim). Go to ehrg.cs.wright.edu/ksim/downloads/downloads.html and download it. Once you've unpacked the archive, locate the file JKSim.jar in the main KSim directory - this is the Java executable to run the simulator (double-click on it, or run it from the command line). Also, locate the /controllers directory in the main simulator directory. This is where you'll put your robot controller .class files so the simulator can find them.

Once you've installed the SDK, the Khepera simulator, you've picked your favorite text editor, and your computer is able to find javac (the Java compiler), you're ready to continue...

And now, an overview of what you'll do to write a robot controller:

You'll write a robot controller source code file, you'll compile it using javac, and you'll copy the compiled .class file to the /controllers directory in the main KSim directory. Then you can use your controller to control the simulated robot.



```
class MyController..
{
   //your code here
    ...
    ...

}
```

```
% javac -cp src source/MyController.java
```

Type your source code, Save it as MyController.java

Compile it using javac, the command-line Java compiler. If you don't have any errors, you'll get a second file named MyController.class.

(We'll show you exactly what to type at the command line to get this to work later in the tutorial.)

Copy MyController.class into the /controllers directory.

Run the sim, and you'll be able to use your controller!

So, the next question is: "What's in a source code file?"

The robot controller source code file you'll be writing (and you'll be writing more than one throughout this course) contains Java code that will tell the robot how to behave - what to do in certain situations, how to get out of trouble... Generally, whatever behaviors are required by the problem at hand.

Because the controller is one part of a larger application (the KSim program), we'll provide you with the following things:

1) A template for writing the controller source code file - you'll mostly need to "fill in the blanks"

2) A list of things you can tell the robot to do, and how to ask it about what it senses - you'll use these commands (methods, really) when writing the controller source code file, and

3) A tutorial of how to use Java to do interesting things with these commands. (that's what this is...)

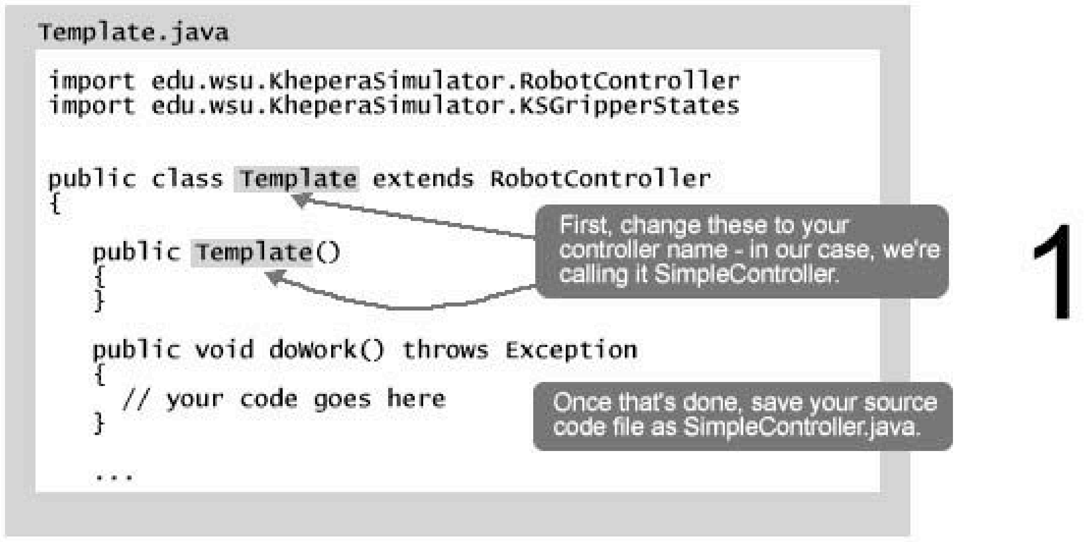
So let's take a look at this template...

The template...

We provide you with the skeleton of a robot controller source code file. It's called Template.java, and you'll find it in the /source directory of the main KSim directory. First thing to do when writing a controller is to open this file, and rename it to whatever name you choose for your controller.

For this tutorial, we'll be using the name SimpleController.

So here we go - open Template.java in your text editor...

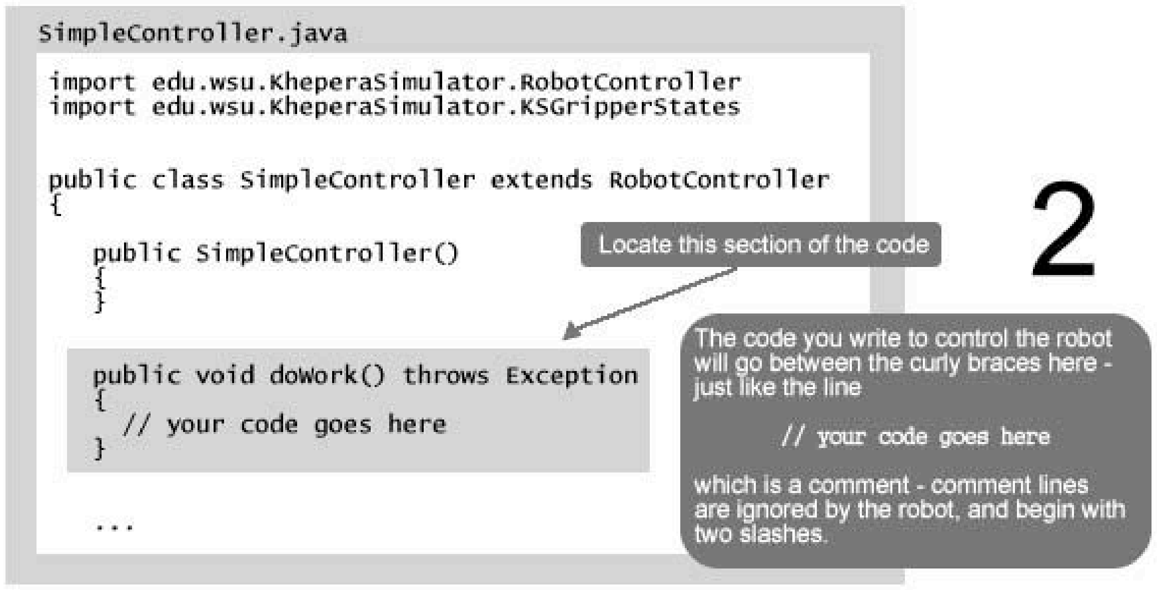


Template.java

```
import edu.wsu.KheperaSimulator.RobotController
import edu.wsu.KheperaSimulator.KSGripperStates


public class Template extends RobotController
{

   public Template()
   {
   }

   public void doWork() throws Exception
   {
     // your code goes here
   }

   ...
```

First, change these to your controller name - in our case, we're calling it SimpleController.
Once that's done, save your source code file as SimpleController.java.
1

Note: When you're creating your own controller, you'll save the file as *ControllerName.java*, where *ControllerName* is the name you chose - again, in this tutorial, the name we're using is SimpleController



SimpleController.java

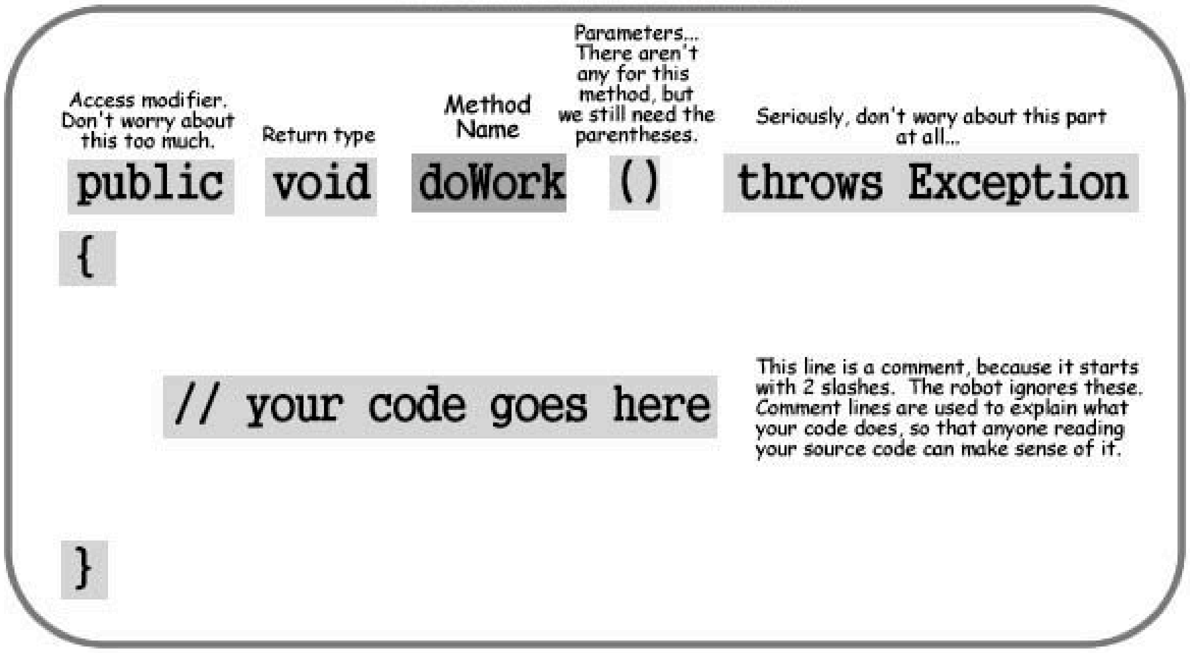
```
import edu.wsu.KheperaSimulator.RobotController
import edu.wsu.KheperaSimulator.KSGripperStates


public class SimpleController extends RobotController
{

   public SimpleController()
   {
   }

   public void doWork() throws Exception
   {
     // your code goes here
   }

   ...
```

Locate this section of the code
2
The code you write to control the robot will go between the curly braces here - just like the line
// your code goes here
which is a comment - comment lines are ignored by the robot, and begin with two slashes.

This section of code is a **method** (or function). Its name is **doWork**. Any commands between the curly braces get repeated to the robot every so often (we'll look at exactly *how* often in a moment, and *why* this repetition is useful later).

A method starts with a line that states its name, as well as some other information about it. This line is called the **method signature**. The method signature is then followed by a set of curly braces. All the statements that make up the method are contained between these braces. Let's look at the doWork method briefly. (we'll talk more about all of this when we start writing our own methods.)

We zoom in on the doWork method...



```
public void doWork () throws Exception
{

    // your code goes here

}
```

Access modifier. Don't worry about this too much.
Return type
Method Name
Parameters... There aren't any for this method, but we still need the parentheses.
Seriously, don't wory about this part at all...
This line is a comment, because it starts with 2 slashes. The robot ignores these. Comment lines are used to explain what your code does, so that anyone reading your source code can make sense of it.

The statements you write in the doWork method are repeated, by default, every 5 milliseconds. ***This is something that the Khepera Simulator application does for this method, and only this method.*** (that is, the methods you write won't automatically repeat, unless you make them do it) You can change this interval (as we'll see on the following page), if you want to.

Now - let's make our SimpleController do something!

Remember that list of commands we mentioned earlier? Well, one of those commands is:

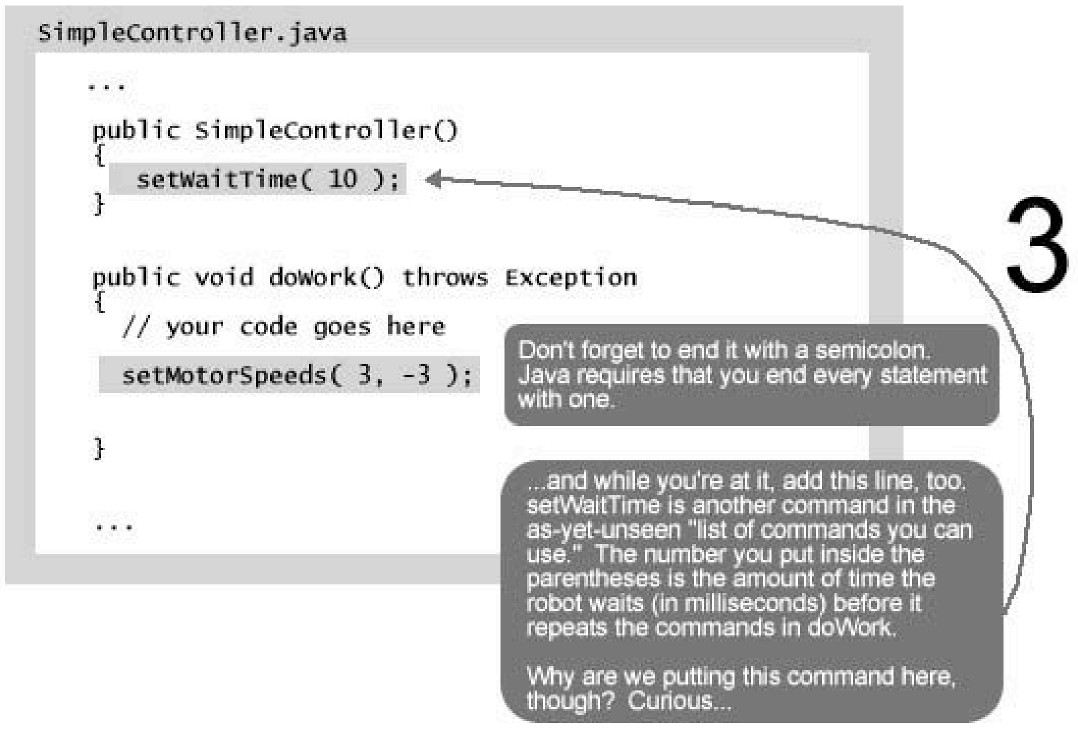
```
setMotorSpeeds( x, y );
```

The Khepera has 2 wheels, and a motor to drive each one. The way you'll control the Khepera's movement is to set the speeds of the motors. x and y are integers that represent a speed setting of the left and right motors, respectively. The settings range from 5 (forward fastest) to 0 (stopped) to -5 (backwards fastest). So we'll use the values 3 and -3 for x and y, giving us:

```
setMotorSpeeds( 3, -3 );
```

Let's put this command into our doWork method...

Type the line into your text editor, as shown below...



SimpleController.java

```
 ...

  public SimpleController()
  {
    setWaitTime( 10 );
  }

  public void doWork() throws Exception
  {
    // your code goes here

   setMotorSpeeds( 3, -3 );

  }

  ...
```

Don't forget to end it with a semicolon. Java requires that you end every statement with one.

...and while you're at it, add this line, too. setWaitTime is another command in the as-yet-unseen "list of commands you can use." The number you put inside the parentheses is the amount of time the robot waits (in milliseconds) before it repeats the commands in doWork.

Why are we putting this command here, though? Curious...

OK - here's why. That section of the code that starts with

```
public SimpleController()
```

sort of looks like a method, too, right? Well, it's a special kind of method, called a **constructor.** The code inside its curly braces gets run once, when the controller starts running. Usually, though, we don't put any commands to the robot, other than setWaitTime() here. (It's considered uncivilized...)

At this point, we have a controller that will do something. So let's save the source code file again in the /source directory of the main KSim directory. Make sure you've saved it as a file named **SimpleController.java**.

We're now ready to compile our robot controller source code file (SimpleController.java) using the Java compiler...

How to compile your source code...

1) Make sure your source code file exists in the /source directory of the main Khepera Simulator directory.
2) Start a command-prompt terminal (In Windows, Start > Run, then type *cmd* in the Run dialogue box)
3) Change your directory to the main directory of the Khepera Simulator Directory
4) Type the following line at the prompt:

```
javac -cp src source/SimpleController.java
```

(If you receive any errors, go back and make sure that you've followed the steps we've done so far.)

How to run a robot controller...

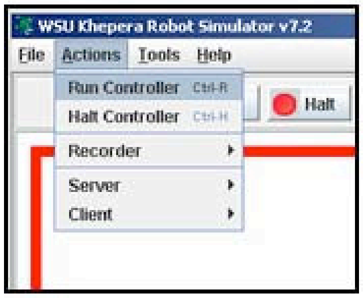
OK - we've compiled our source code file, SimpleController.java, and there should now exist in the same directory, another file called SimpleController.class

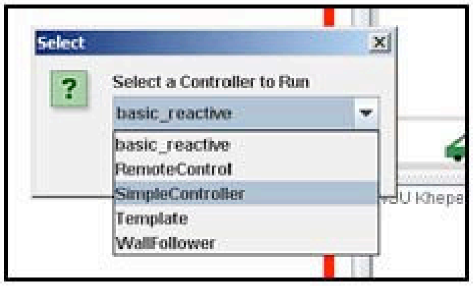
Copy SimpleController.class to the /controllers directory of the main Khepera Simulator Directory.

Start the Wright State University Khepera Simulator - you may double-click the icon of the JKSim.jar file if you are running Windows. Otherwise, enter the following command at the command line:

```
java -jar JKSim.jar
```

Select Actions > Run Controller from the menubar at the top of the window, and then choose our SimpleController from the dropdown menu of the Run Controller dialogue box that appears.





(before you click Run, any guesses as to what our controller does?)