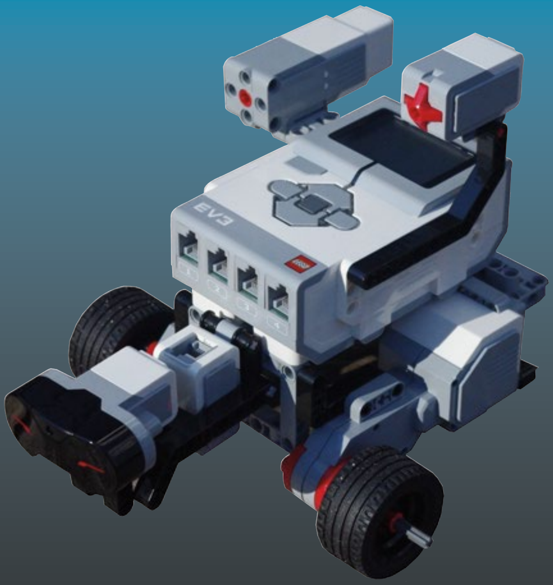
TECHNOLOGY IN ACTION™



# Beginning Robotics Programming in Java with LEGO Mindstorms

Wei Lu

Apress®

# Beginning Robotics Programming in Java with LEGO Mindstorms

Wei Lu

Apress®

***Beginning Robotics Programming in Java with LEGO Mindstorms***

Wei Lu
Keene, New Hampshire, USA

ISBN-13 (pbk): 978-1-4842-2004-7 ISBN-13 (electronic): 978-1-4842-2005-4
DOI 10.1007/978-1-4842-2005-4

Library of Congress Control Number: 2016959245

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text are available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code/`. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

*My wife Ling, for her endless love and support over the past five years when I worked on this book.*
*My daughter Julia and son Ryan, for giving their dad the best fun times when testing all of the robot's programs with them.*

# Contents at a Glance

# Contents

# About the Author

**Wei Lu** is an Associate Professor of Computer Science at Keene State College in New Hampshire. He received his Ph.D. in Electrical and Computer Engineering from the University of Victoria, Canada. Prior to joining Keene State College, he was a Senior Researcher with the German Research Centre for Artificial Intelligence (DFKI GmbH) and worked with Q1 Labs Inc. (Security Systems Division, IBM since October 2011) as a Secure Software Engineer. His general areas of research interests include Artificial Intelligence and Cyber Security. He has had more than 50 papers published by peer-reviewed journals, book chapters, and conference proceedings. In addition, he coauthored, *Network Intrusion Detection and Prevention: Concepts and Techniques* (Springer, 2010) and has served as a technical program committee member and a technical reviewer for more than 70 international conferences and journals.

# Acknowledgments

I would like to express my deepest gratitude to the following people:

My wife, for her endless love and support over the past five years when I worked on this book.

My daughter and son, for giving their dad the best fun times when testing all of the robot’s programs with them.

My parents, for giving me the encouragement to keep exploring new opportunities.

My colleagues Michael and Shari, for promoting Java robotics programming computer science education.

All my students, who participated the course, CS495 Artificial Intelligence and Robotics. This book would not be published without their motivation.

Nanyan Wang, for sharing his experience and insights in engineering and computer science and his technical review of this book.

The Apress team, for leading me through the entire jungle of authoring a book. Without their passion for publishing the best robotics programming book in Java for beginners, I would not have had an opportunity to write and publish this book.

# Introduction

There are many cognitive tasks that people can do easily and almost subconsciously, but that have proven extremely difficult to program on a computer. Artificial Intelligence (AI) is the process of developing computer systems that can carry out these tasks, and it is devoted to the computational study of intelligent behavior. Such intelligent behavior includes a wide range of phenomena, such as perception, problem solving, use of knowledge, planning, learning, and communication in order to take a complicated task and convert it into simpler steps that the robotics system can handle. Based on the Lego Mindstorms robotic system, this book develops a wide range of techniques in the Java programming language for modeling these phenomena, including state-space search, several knowledge representation schemes, and task-specific methods.

The book begins with an introduction to Lego Mindstorms EV3 and leJOS, an open source project created to develop the technological infrastructure, and a tiny Java virtual machine in which to implant software into Lego Mindstorms products using Java technology. It then continues with a discussion of problem- solving techniques, such as breadth-first search, depth-first search, heuristic search, hill-climbing search, and A star (A*) search, and finishes with robotics behavior programming in Java multithreading programming with a set of sensors.

A major goal of AI is to give computers the ability to think, or in other words, mimic human behavior. The problem with this mimicry is that, unfortunately, computers don't function in the same way as the human brain; that is, they require a series of well reasoned-out steps in order to find a solution. Therefore, one challenge in robotics programming is how to convert something complex into something simple that can be done by algorithms. This book bridges the gap between the theoretical AI algorithms and practical robotics systems by developing a set of algorithms and building them into the well-known Lego Mindstorms EV3 system in order to achieve an enhanced intelligence.

CHAPTER 1

# Introduction to Lego Mindstorms and leJOS

This chapter provides step-by-step guidance on how to set up the environment for Java robotics programming with Lego Mindstorms EV3, including a basic overview of Lego Mindstorms EV3 and an introduction to leJOS EV3. The chapter covers how to install the leJOS EV3 development system onto your computer, how to install leJOS EV3 firmware into the Lego EV3 brick, and how to install and apply the leJOS EV3 plug-in for Eclipse IDE. Finally, you will create your first Java robotics program called *HelloWorld*, upload the *HelloWorld* program from your computer into the Lego EV3 brick through a USB cable, and execute the program on Lego Mindstorms EV3.

## Introduction to Lego Mindstorms

Lego Mindstorms is an educational product from Lego designed to help you build robots easily. The product series has been evolving for decades, and Lego Mindstorms EV3 is the third generation. Figure 1-1 illustrates a typical Lego Mindstorms EV3 robotics set in which the EV3 brick is the brain of Lego Mindstorms EV3. It is an intelligent, programmable device that lets a Lego robot perform various intelligent operations.

**Electronic supplementary material** The online version of this chapter (doi:10.1007/978-1-4842-2005-4_1) contains supplementary material, which is available to authorized users.

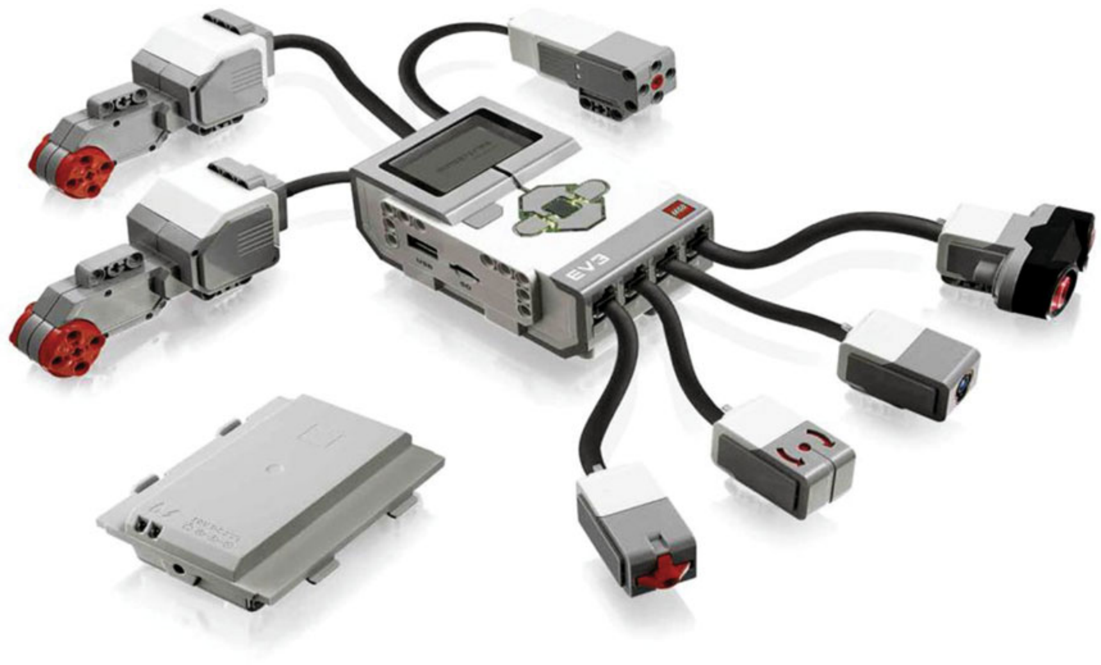


***Figure 1-1.*** *Lego Mindstorms EV3*

As illustrated in Figure 1-1, typical components of Lego Mindstorms EV3 include motor ports, sensor ports, a PC connection port, speakers, and EV3 buttons. Further details about the parts that comprise Lego Mindstorms EV3 can be found at: `http://www.lego.com/en-us/mindstorms/downloads`.

The motor ports included in Lego Mindstorms EV3 have four output ports for attaching motors: Ports A, B, C, and D.

Sensor Ports have four input ports for attaching sensors: Ports 1, 2, 3, and 4.

A mini-USB PC connection port is used to connect a USB cable to your local computer and download programs to the EV3 brick (or to upload data from the robot to your local computer). You can also use the wireless Bluetooth connection for uploading and downloading programs.

The speaker included in Lego Mindstorms EV3 makes programs with real sounds possible, and you can listen to them when you run programs.

You apply a dark gray button at the center of the EV3 brick for powering on, entering commands, or running programs. The brick also has a light gray button on the upper-left side, which is used for reversing actions, aborting a program, and shutting down the EV3. The other four light gray buttons on the EV3 brick are used for moving left, right, up, and down while walking through the EV3 menu.

Typical technical specifications for Lego Mindstorms EV3 are listed below. Additional detail on the Lego EV3 specifications can be found at: `http://www.lego.com/en-us/mindstorms/downloads`.

- A single main processor controls the robot:
  - 32-bit ARM9 processor running at 300 MHz
  - Ability to access 64 MB of RAM
  - Uses 16 MB of flash memory

- The operating system is Linux based
- Runs on 6 AA batteries or the rechargeable battery pack, which is slightly larger:
    - Although 6 AA batteries are theoretically equate to 9 volts, you are more likely to experience about 7-8 Volts, depending on the charge in the batteries
- Contains four motor/servo ports:
    - Three motors (two large motors and one medium-sized motor) come with each Lego Mindstorms EV3 kit
- Contains four sensor ports:
    - Comes with a variety of sensors:

      - Touch

      - Color

      - Ultrasonic

      - Gyro
- Communication:
    - Comes with Bluetooth onboard
    - A program can be loaded onto the EV3 brick using Bluetooth at a slow speed or with a USB cable at a much faster speed
    - Can be programmed to allow for communications between two (or more) EV3 bricks while the program is executing
    - Third-party software can be used to communicate between a PC and an EV3 brick while a program is executing
    - Includes a built-in 178 × 128-pixel LCD graphical display

## Introduction to leJOS

"leJOS" means Lego for Java Operating System, which is an open source language created for developing software for Lego Mindstorms products using Java technology. The leJOS project offers support for Lego Mindstorms EV3, and previous versions including Lego Mindstorms NXT and Lego Mindstorms RCX. The leJOS project delivers the following solutions for Lego Mindstorms:

### Lego Mindstorms EV3

- JVM for EV3 Brick
- leJOS API for EV3 brick
- leJOS PC Communications
- leJOS Tools

## Lego Mindstorms NXT

- JVM for NXT Brick
- leJOS API for NXT brick
- leJOS PC Communications
- leJOS JavaME Communications
- leJOS Tools

## Lego Mindstorms RCX

- JVM for RCX Brick
- leJOS API for RCX brick
- leJOS PC Communications
- leJOS Tools

In this book, we are focused on the most recent Lego Mindstorms product: that is, Lego Mindstorms EV3. Typical official packages provided by leJOS for the EV3 brick are illustrated in Table 1-1. These packages allow you to manage the EV3 brick, sensors, and actuators, as well as some other pieces of EV3 hardware.

***Table 1-1.*** *EV3 brick packages*

| Package | Description |
|---|---|
| **lejos.hardware** | To support EV3 hardware |
| **lejos.hardware.ev3** | To access EV3 hardware |
| **lejos.hardware.lcd** | To access EV3 LCD |
| **lejos.hardware.motor** | To access EV3 motors |
| **lejos.hardware.port** | To access EV3 ports |
| **lejos.hardware.sensor** | To access all the sensors that are supported on the EV3 |
| **lejos.hardware.video** | To access video devices |

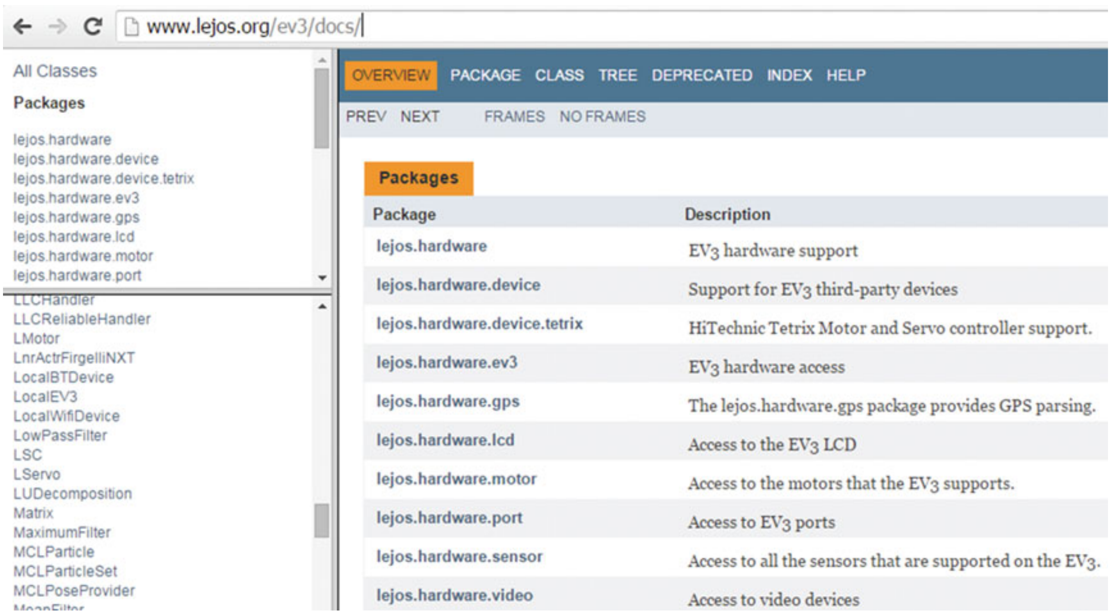
Table 1-2 lists packages that offer support for some robotics problems, such as localization and navigation.

***Table 1-2.*** *Robotics/AI packages*

| Package | Description |
|---|---|
| **lejos.robotics.localization** | Localization support |
| **lejos.robotics.mapping** | Support for maps |
| **lejos.robotics.navigation** | Navigation classes |
| **lejos.robotics.objectdetection** | Object detection classes |
| **ejos.robotics.subsumption** | Support for subsumption architecture |

All leJOS releases have documentation on the packages in the format of Javadoc. Details on the packages provided in leJOS EV3 can be found at: `http://www.lejos.org/ev3/docs/`.

Figure 1-2 shows the leJOS EV3 development documents.



www.lejos.org/ev3/docs/

All Classes

**Packages**

lejos.hardware
lejos.hardware.device
lejos.hardware.device.tetrix
lejos.hardware.ev3
lejos.hardware.gps
lejos.hardware.lcd
lejos.hardware.motor
lejos.hardware.port

LLCHandler
LLCReliableHandler
LMotor
LnrActrFirgelliNXT
LocalBTDevice
LocalEV3
LocalWifiDevice
LowPassFilter
LSC
LServo
LUDecomposition
Matrix
MaximumFilter
MCLParticle
MCLParticleSet
MCLPoseProvider

OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV NEXT FRAMES NO FRAMES

**Packages**

| Package | Description |
|---|---|
| lejos.hardware | EV3 hardware support |
| lejos.hardware.device | Support for EV3 third-party devices |
| lejos.hardware.device.tetrix | HiTechnic Tetrix Motor and Servo controller support. |
| lejos.hardware.ev3 | EV3 hardware access |
| lejos.hardware.gps | The lejos.hardware.gps package provides GPS parsing. |
| lejos.hardware.lcd | Access to the EV3 LCD |
| lejos.hardware.motor | Access to the motors that the EV3 supports. |
| lejos.hardware.port | Access to EV3 ports |
| lejos.hardware.sensor | Access to all the sensors that are supported on the EV3. |
| lejos.hardware.video | Access to video devices |

***Figure 1-2.*** *leJOS development documents*

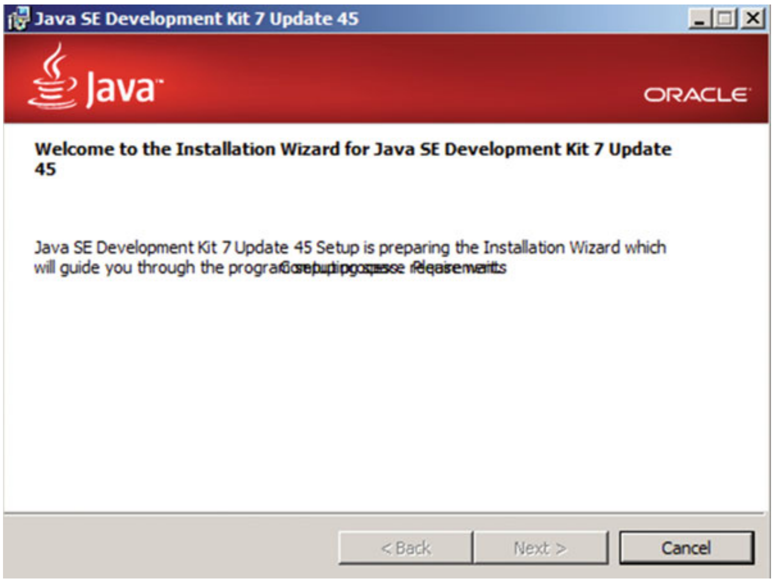
## JDK Installation

The leJOS project is based on Java technology, so you need to install the Java Development Kit (JDK) current release on your local computer. The JDK release can be found at: `http://www.oracle.com/technetwork/java/index.html`.

A Java Runtime Environment (JRE) is not sufficient, as it does not allow you to compile Java programs. leJOS EV3 only works with a 32-bit version of the JDK and JRE, so even if you have a 64-bit system, you should select a 32-bit version of the JDK. Also leJOS EV3 has been tested with JDK versions 1.7, and thus Java 7 is recommended in this book. As an example, the following steps show you how to install JDK using the Java JDK installer called `jdk-7u45-windows-i586.exe`.

## INSTALLING THE JDK

1. Double-click the file `jdk-7u45-windows-i586.exe`, and you will see the screen shown in Figure 1-3. Then, click the *Next* button.

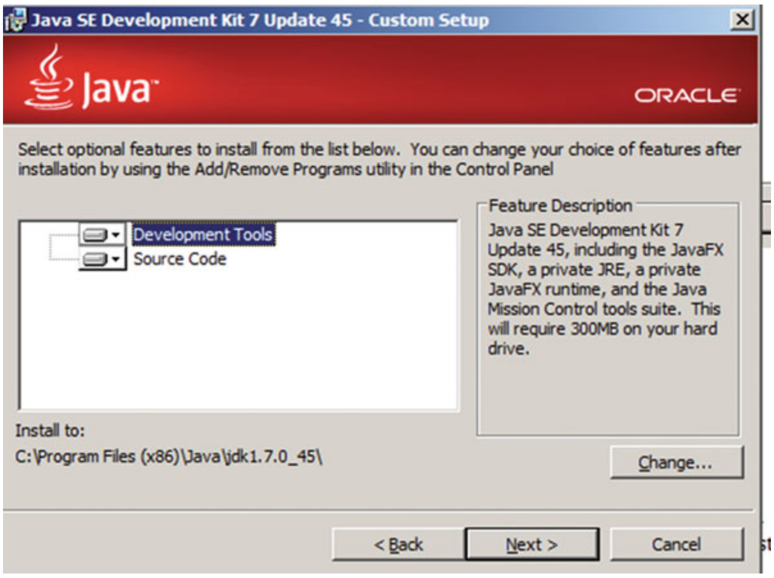

Java SE Development Kit 7 Update 45

Welcome to the Installation Wizard for Java SE Development Kit 7 Update 45

Java SE Development Kit 7 Update 45 Setup is preparing the Installation Wizard which will guide you through the prograComputing space requirements setup process. Please wait.

***Figure 1-3.*** *Step 1 of the JDK Installation*

2. Install the JAVA JDK to the path `C:\Program Files (x86)\Java\jdk1.7.0_45`, choose all components, and click the *Next* button, as illustrated in Figure 1-4. It will then install JDK components that you chose.


Java SE Development Kit 7 Update 45 - Custom Setup

Select optional features to install from the list below. You can change your choice of features after installation by using the Add/Remove Programs utility in the Control Panel

Development Tools
Source Code

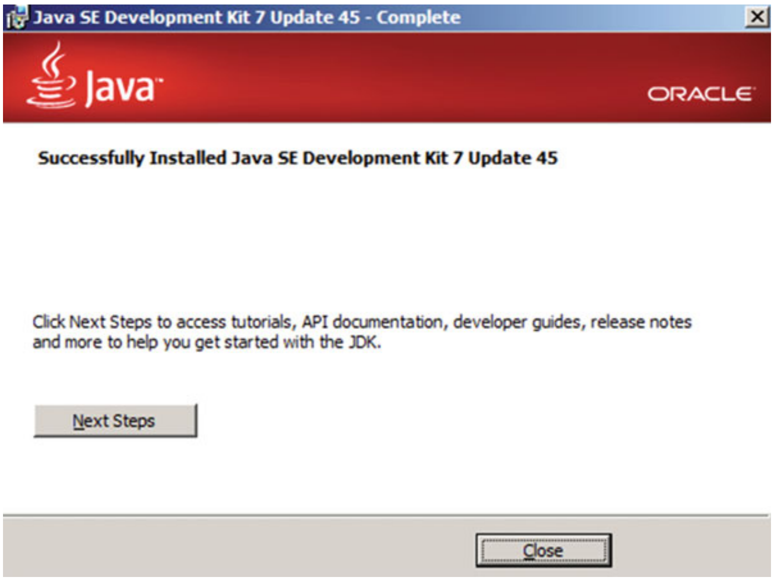
Feature Description
Java SE Development Kit 7 Update 45, including the JavaFX SDK, a private JRE, a private JavaFX runtime, and the Java Mission Control tools suite. This will require 300MB on your hard drive.

Install to:
C:\Program Files (x86)\Java\jdk1.7.0_45\

Change... < Back Next > Cancel

*Figure 1-4.* *Step 2 of the JDK installation*

3. Click the *Close* button, as shown in Figure 1-5. The JAVA JDK is then successfully installed on your computer at: `C:\Program Files (x86)\Java\jdk1.7.0_45`.


Java SE Development Kit 7 Update 45 - Complete

**Successfully Installed Java SE Development Kit 7 Update 45**

Click Next Steps to access tutorials, API documentation, developer guides, release notes and more to help you get started with the JDK.

Next Steps

Close

*Figure 1-5.* *Step 3 of the JDK installation*

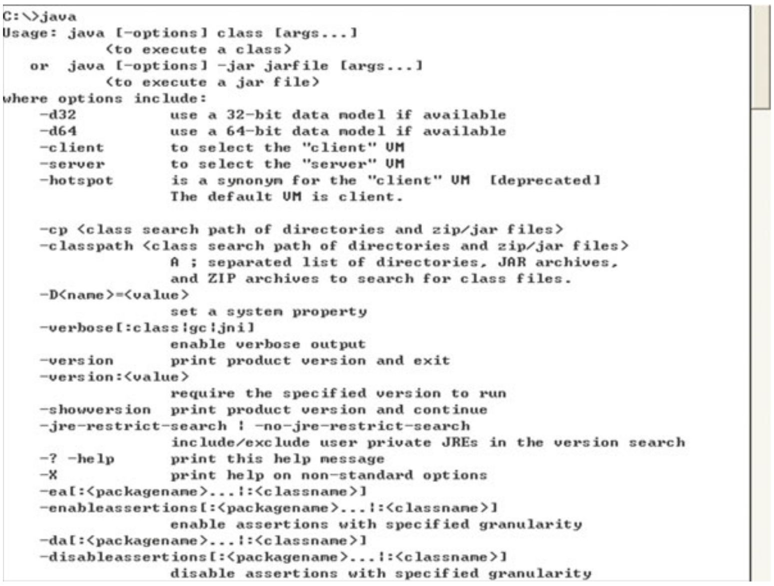
4. Once you have installed the J2SE SDK on your computer, it is necessary to check that you can compile and execute any java program.

## Testing the JDK Installation

Open a Shell console on your computer, and type the command `Java`:

- `Java`*:* Java command used to execute Java programs
- `Javac`*:* Java command used to compile Java programs

The reason to perform the first test is because you need to check that your operating system recognizes the command `java`, which is used to execute Java programs. If the shell console returns the options to use the command, as shown in Figure 1-6, then the test is a success.

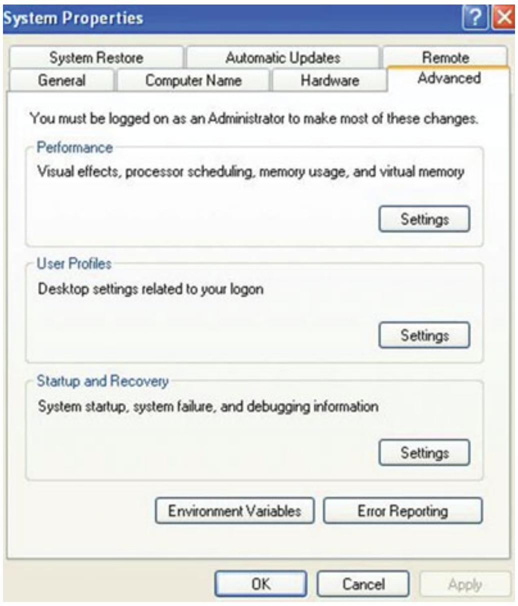


```
C:\>java
Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)
where options include:
    -d32          use a 32-bit data model if available
    -d64          use a 64-bit data model if available
    -client       to select the "client" VM
    -server       to select the "server" VM
    -hotspot      is a synonym for the "client" VM  [deprecated]
                  The default VM is client.

    -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar files>
                  A ; separated list of directories, JAR archives,
                  and ZIP archives to search for class files.
    -D<name>=<value>
                  set a system property
    -verbose[:class|gc|jni]
                  enable verbose output
    -version      print product version and exit
    -version:<value>
                  require the specified version to run
    -showversion  print product version and continue
    -jre-restrict-search | -no-jre-restrict-search
                  include/exclude user private JREs in the version search
    -? -help      print this help message
    -X            print help on non-standard options
    -ea[:<packagename>...|:<classname>]
    -enableassertions[:<packagename>...|:<classname>]
                  enable assertions with specified granularity
    -da[:<packagename>...|:<classname>]
    -disableassertions[:<packagename>...|:<classname>]
                  disable assertions with specified granularity
```

***Figure 1-6.*** *Test running the java command*

The second test is necessary to know if your operating system recognizes the command `javac`, which is used to compile your programs. Type `javac` on your keyboard and check the message. If your system doesn't recognize the command, then you have to update environment variables in your computer system. Right-click on your "My Computer" icon and select *properties*. Click on the tab *advanced options*. When you click this command, you will see a new window where you can update the variable path. This path is used to execute commands directly from a Shell keyboard.

On the tab *Advanced options* tab, click on the *Environment variables* button, as shown in Figure 1-7. The `path` variable is located in the *System variables* area. Find the variable `path`, and click the *Update* button. Path variables may have many statements because they are used by many applications.


System Properties

System Restore | Automatic Updates | Remote
General | Computer Name | Hardware | Advanced

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

***Figure 1-7.*** *A screenshot of setting up environment variables*

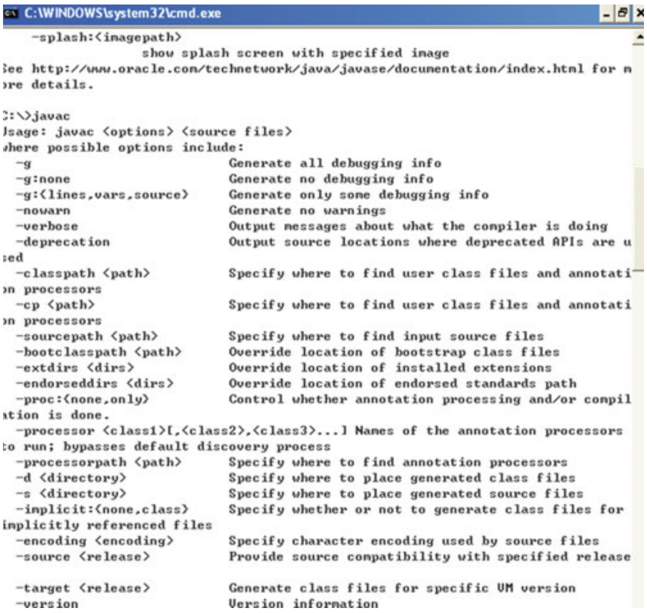
To update the `path` variable, find the path on your computer where the J2SE SDK is located. In this case, the path is:

```
C:\Program Files (x86)\Java\jdk1.7.0_45\bin
```

Once you know the path, add it at the end of the content of the *System variable* path.

You also need to create a new system variable called `JAVA_HOME`, and set its value to `C:\Program Files (x86)\Java\jdk1.7.0_45`. Once you have made the changes, reboot the DoS command window, and check the command `javac` again.

If you see the options for the command `javac`, as shown in Figure 1-8, then the test is a success — Congratulations! You have finished the JDK installation. Now you can use your computer to develop Java programs, and you have the basic java tools installed and configured.



```
    -splash:<imagepath>
                  show splash screen with specified image
See http://www.oracle.com/technetwork/java/javase/documentation/index.html for m
ore details.

C:\>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are u
sed
  -classpath <path>          Specify where to find user class files and annotati
on processors
  -cp <path>                 Specify where to find user class files and annotati
on processors
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -endorseddirs <dirs>       Override location of endorsed standards path
  -proc:{none,only}          Control whether annotation processing and/or compil
ation is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors
to run; bypasses default discovery process
  -processorpath <path>      Specify where to find annotation processors
  -d <directory>             Specify where to place generated class files
  -s <directory>             Specify where to place generated source files
  -implicit:{none,class}     Specify whether or not to generate class files for
implicitly referenced files
  -encoding <encoding>       Specify character encoding used by source files
  -source <release>          Provide source compatibility with specified release

  -target <release>          Generate class files for specific VM version
  -version                   Version information
```

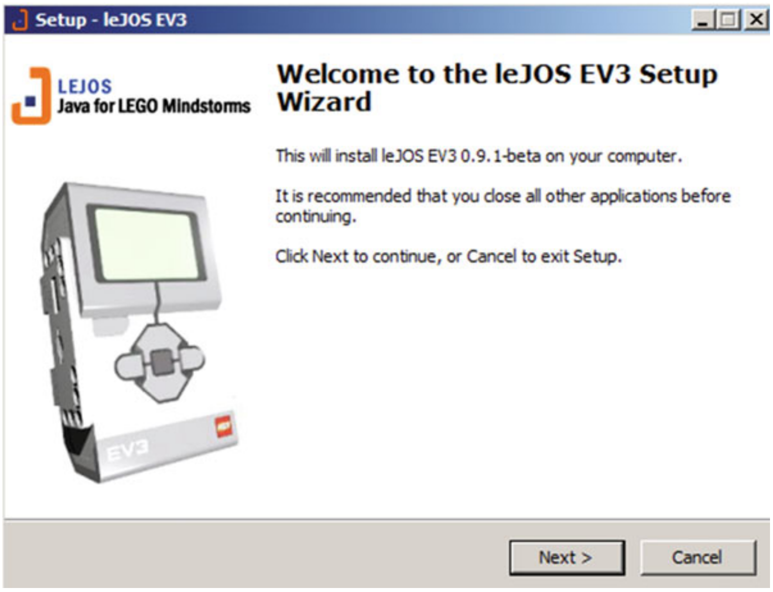
***Figure 1-8.*** *Test running the javac command*

## Installation of leJOS and Its Firmware on Lego EV3

leJOS can be installed in the leading operating systems such as Windows, Linux, and Mac OS. Since you have Java installed and working, it's time to install the leJOS system on your PC and its firmware on the Lego Mindstorms EV3. To do this, you will need an empty SD card with a maximum of 32GB capacity. Also, the SD card needs to be formatted with a FAT32 partition. The easiest way to install leJOS is to download the installer at: `http://www.lejos.org`.

### INSTALLING LEJOS

1. Visit `https://sourceforge.net/projects/ev3.lejos.p/files/0.9.1-beta/`, and then choose `Download leJOS_EV3_0.9.1-beta_win32_setup.exe (41.8MB)`. You can then download and save `leJOS_EV3_0.9.1-beta_win32_setup.exe` on your computer.
2. Double-click `leJOS_EV3_0.9.1-beta_win32_setup.exe`, and you will see a leJOS EV3 Setup Wizard, as shown in Figure 1-9.



Welcome to the leJOS EV3 Setup Wizard

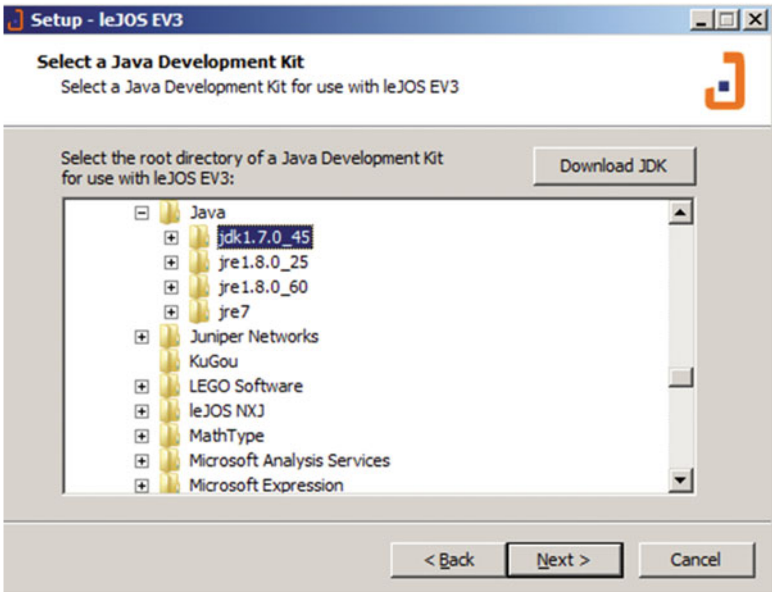
This will install leJOS EV3 0.9.1-beta on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

*Figure 1-9.* *leJOS setup wizard*

3. Click the *Next* button, and you will see the screen shown in Figure 1-10.



Select a Java Development Kit

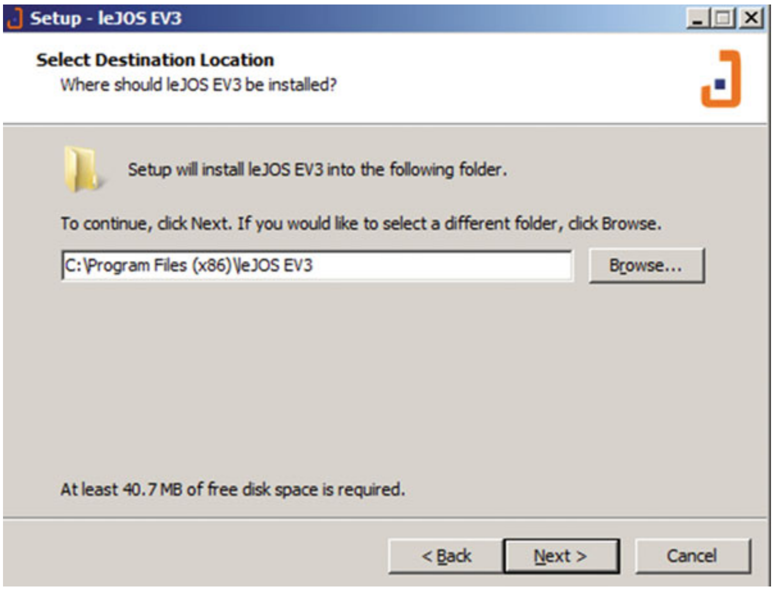
Select a Java Development Kit for use with leJOS EV3

Select the root directory of a Java Development Kit for use with leJOS EV3:

```
⊟ Java
    ⊞ jdk1.7.0_45
    ⊞ jre1.8.0_25
    ⊞ jre1.8.0_60
    ⊞ jre7
⊞ Juniper Networks
  KuGou
⊞ LEGO Software
⊞ leJOS NXJ
⊞ MathType
⊞ Microsoft Analysis Services
⊞ Microsoft Expression
```

*Figure 1-10.* *Choosing the right JDK for use with leJOS EV3*

4. Choose the 32-bit JDK that you installed, for example, `jdk1.7.0_45`, and then click the *Next* button. You will see the screen shown in Figure 1-11.



Setup - leJOS EV3

Select Destination Location

Where should leJOS EV3 be installed?

Setup will install leJOS EV3 into the following folder.

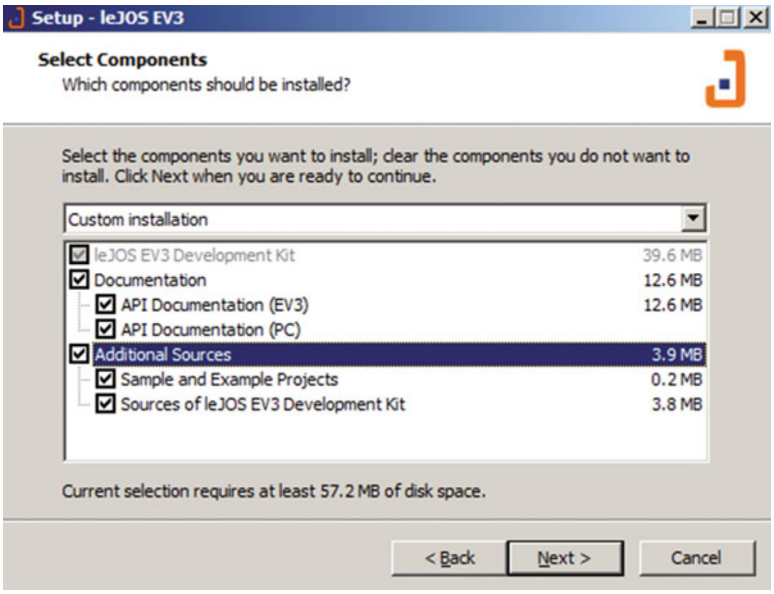
To continue, click Next. If you would like to select a different folder, click Browse.

C:\Program Files (x86)\leJOS EV3

At least 40.7 MB of free disk space is required.

***Figure 1-11.*** *Choosing a folder to install leJOS EV3 program*

5. Make sure that you have installed the JDK and set the path and `JAVA_HOME` to the installation directory of your JDK. You can click *browse* to select the path to which you want to install. In this example, I chose the destination folder `C:\Program Files (x86)\leJOS EV3`. After you set up the installation folder, click *Next* button and you will see the screen shown in Figure 1-12.


Setup - leJOS EV3

**Select Components**

Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

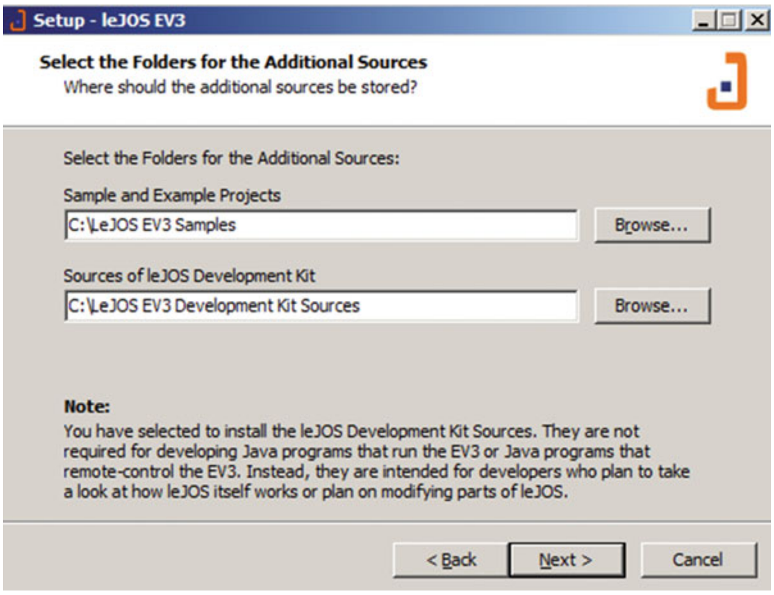
Custom installation

| | |
|---|---|
| ☑ leJOS EV3 Development Kit | 39.6 MB |
| ☑ Documentation | 12.6 MB |
| ☑ API Documentation (EV3) | 12.6 MB |
| ☑ API Documentation (PC) | |
| ☑ Additional Sources | 3.9 MB |
| ☑ Sample and Example Projects | 0.2 MB |
| ☑ Sources of leJOS EV3 Development Kit | 3.8 MB |

Current selection requires at least 57.2 MB of disk space.

***Figure 1-12.*** *Choosing ALL components of leJOS*

6. Check to choose all of the components you wish to install, and then click *Next* button. You will see the screen shown in Figure 1-13.


Setup - leJOS EV3

**Select the Folders for the Additional Sources**

Where should the additional sources be stored?

Select the Folders for the Additional Sources:

Sample and Example Projects

C:\LeJOS EV3 Samples

Sources of leJOS Development Kit

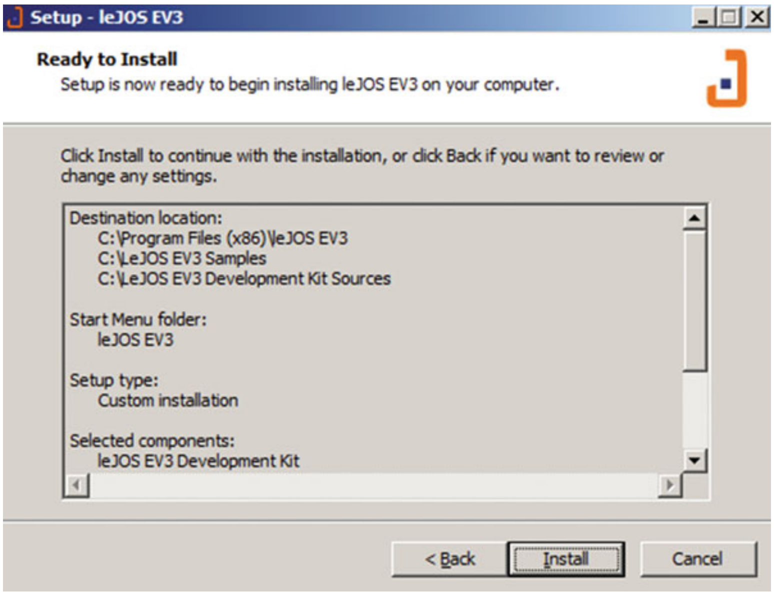
C:\LeJOS EV3 Development Kit Sources

**Note:**

You have selected to install the leJOS Development Kit Sources. They are not required for developing Java programs that run the EV3 or Java programs that remote-control the EV3. Instead, they are intended for developers who plan to take a look at how leJOS itself works or plan on modifying parts of leJOS.

***Figure 1-13.*** *Selecting folders where to install the sample leJOS projects*

7. Choose to install the Sample Projects and the Development Kit into root folder C, click *Next*, and then you can use the default setting to create a Start Menu folder called `leJOS EV3`. After that, click *Next* and you will see the screen shown in Figure 1-14.



Setup is now ready to begin installing leJOS EV3 on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

Destination location:
C:\Program Files (x86)\leJOS EV3
C:\LeJOS EV3 Samples
C:\LeJOS EV3 Development Kit Sources

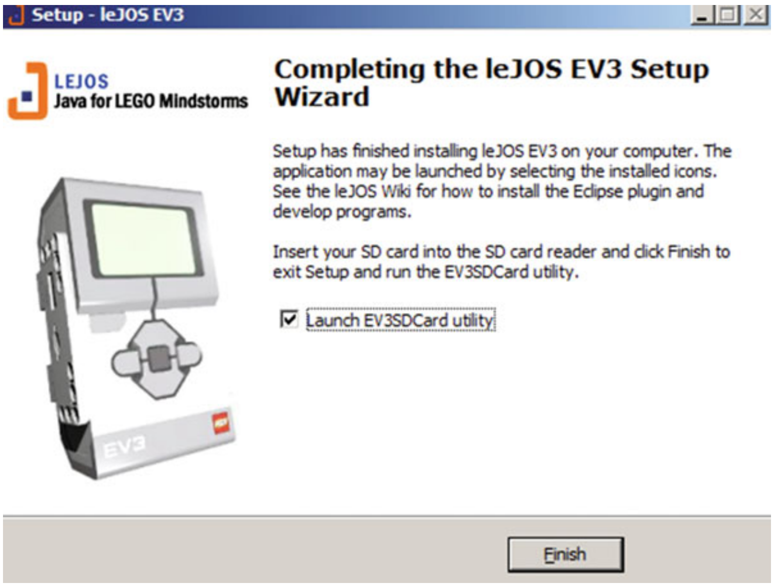
Start Menu folder:
leJOS EV3

Setup type:
Custom installation

Selected components:
leJOS EV3 Development Kit

***Figure 1-14.*** *General settings for the leJOS installation*

8. Double-check to see if all of the settings are OK, and then click Install. You will see the installation progress bar, and eventually you will see the screen shown in Figure 1-15.


Setup - leJOS EV3

LEJOS
Java for LEGO Mindstorms

Completing the leJOS EV3 Setup Wizard

Setup has finished installing leJOS EV3 on your computer. The application may be launched by selecting the installed icons. See the leJOS Wiki for how to install the Eclipse plugin and develop programs.

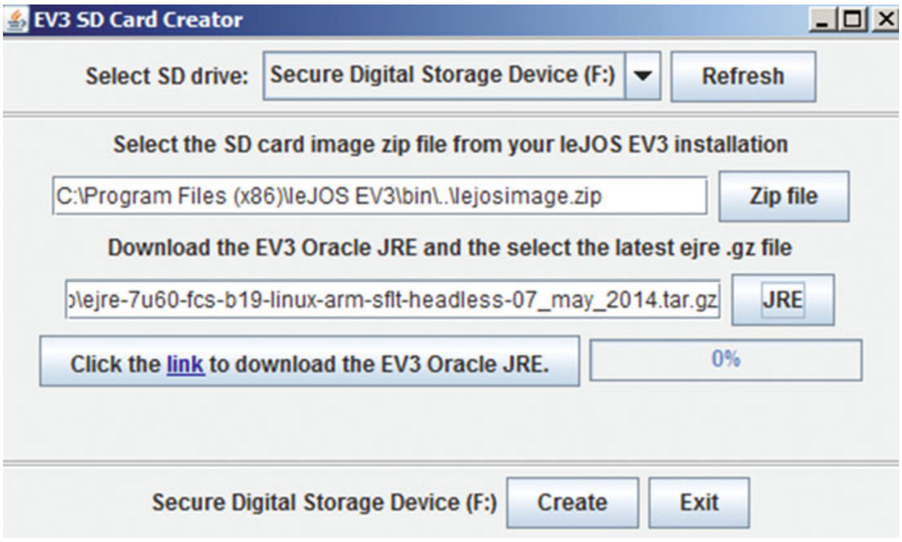
Insert your SD card into the SD card reader and click Finish to exit Setup and run the EV3SDCard utility.

☑ Launch EV3SDCard utility

Finish

*Figure 1-15.* *Finishing the leJOS installation*

9. Make sure that you have your SD card ready, and then click *Finish*. After that, a *EV3SDCard* utility program will start, as shown in Figure 1-16.


EV3 SD Card Creator

Select SD drive: Secure Digital Storage Device (F:) Refresh

Select the SD card image zip file from your leJOS EV3 installation

C:\Program Files (x86)\leJOS EV3\bin\..\lejosimage.zip Zip file

Download the EV3 Oracle JRE and the select the latest ejre .gz file

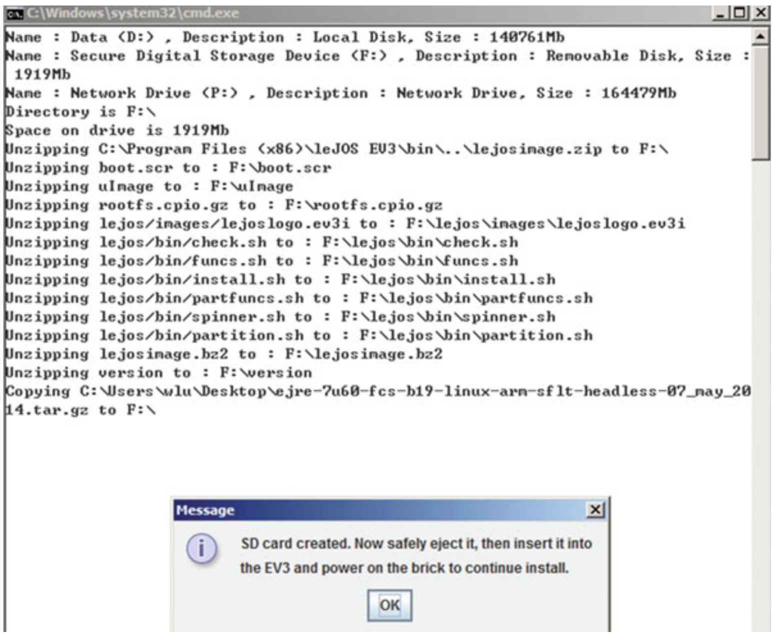
\ejre-7u60-fcs-b19-linux-arm-sflt-headless-07_may_2014.tar.gz JRE

Click the link to download the EV3 Oracle JRE. 0%

Secure Digital Storage Device (F:) Create Exit

*Figure 1-16.* *EV3 SD Card creator*

10. Choose the right SD Card drive, click the link to *Download the EV3 Oracle JRE,* and select the corresponding `.gz` file. Then click the *Create* button and you will see that the EV3 firmware is burned into the SD card, as shown in Figure 1-17.



```
C:\Windows\system32\cmd.exe
Name : Data (D:) , Description : Local Disk, Size : 140761Mb
Name : Secure Digital Storage Device (F:) , Description : Removable Disk, Size :
 1919Mb
Name : Network Drive (P:) , Description : Network Drive, Size : 164479Mb
Directory is F:\
Space on drive is 1919Mb
Unzipping C:\Program Files (x86)\leJOS EV3\bin\..\lejosimage.zip to F:\
Unzipping boot.scr to : F:\boot.scr
Unzipping uImage to : F:\uImage
Unzipping rootfs.cpio.gz to : F:\rootfs.cpio.gz
Unzipping lejos/images/lejoslogo.ev3i to : F:\lejos\images\lejoslogo.ev3i
Unzipping lejos/bin/check.sh to : F:\lejos\bin\check.sh
Unzipping lejos/bin/funcs.sh to : F:\lejos\bin\funcs.sh
Unzipping lejos/bin/install.sh to : F:\lejos\bin\install.sh
Unzipping lejos/bin/partfuncs.sh to : F:\lejos\bin\partfuncs.sh
Unzipping lejos/bin/spinner.sh to : F:\lejos\bin\spinner.sh
Unzipping lejos/bin/partition.sh to : F:\lejos\bin\partition.sh
Unzipping lejosimage.bz2 to : F:\lejosimage.bz2
Unzipping version to : F:\version
Copying C:\Users\wlu\Desktop\ejre-7u60-fcs-b19-linux-arm-sflt-headless-07_may_20
14.tar.gz to F:\
```

Message

SD card created. Now safely eject it, then insert it into the EV3 and power on the brick to continue install.

***Figure 1-17.*** *Installing the leJOS firmware into the SD card*

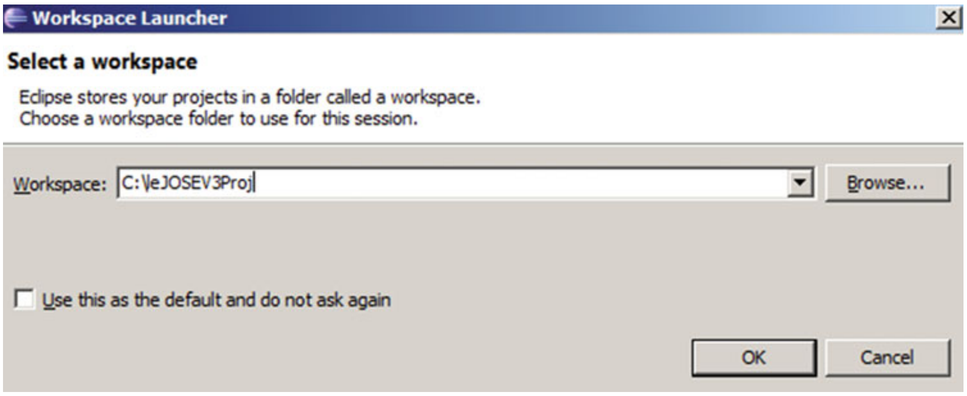
After the SD card is created, you then insert it into the EV3 brick, press the central dark gray button to start up the EV3, and finish the leJOS firmware installation on EV3 brick. A leJOS EV3 logo will be displayed on the LCD of the brick. After the installation is complete, a leJOS EV3 menu will be displayed with a default IP address on the top (`10.0.1.1`). At this point, you are ready to proceed to the next step and install the Eclipse plug-in for developing leJOS programs into the EV3 brick.

## Eclipse IDE and Eclipse Plug-In for LeJOS EV3

Of course it is possible to do Java programming by merely using a text editor and a command line. However, it's much easier for programmers to click on buttons to make things happen rather than typing in commands and optional parameters. Generally, standard text editors do not include many features that help you when editing code, and they do not tell you when you misspell the name of a class or miss inserting a bracket. An *Integrated Development Environment (IDE)* is a tool that allows you to enter, compile, and upload code to your EV3 using simple buttons, and it also monitors the code syntax by color coding your code so that you can identify keywords and variables easily. One of the best open source IDEs is Eclipse by IBM, which is free, powerful, and easy to use. This section will show you how to set up the Eclipse IDE for programming in Java on leJOS EV3.

The first step is to download Eclipse at: `http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/indigosr1`, and search for *Eclipse IDE for Java Developers*. Please note that you need to download the 32-bit version of the *Eclipse IDE*, even if you are using a 64-bit computer, because the leJOS EV3 plug-in will not work with a 64-bit Eclipse version. The download is a `.zip` file, and you must decompress the files into a directory. This will be the permanent location for Eclipse program. To run Eclipse, you simply double-click the executable file in the Eclipse directory. Similarly, in order to delete Eclipse, you merely delete the Eclipse directory from your computer.

The first time that you run Eclipse, it will ask you for a workspace location. In this lab, you can select the workspace as the folder that you used to install the leJOS, which in this case is `C:\leJOSEV3Proj`, as illustrated in Figure 1-18.


Workspace Launcher

Select a workspace

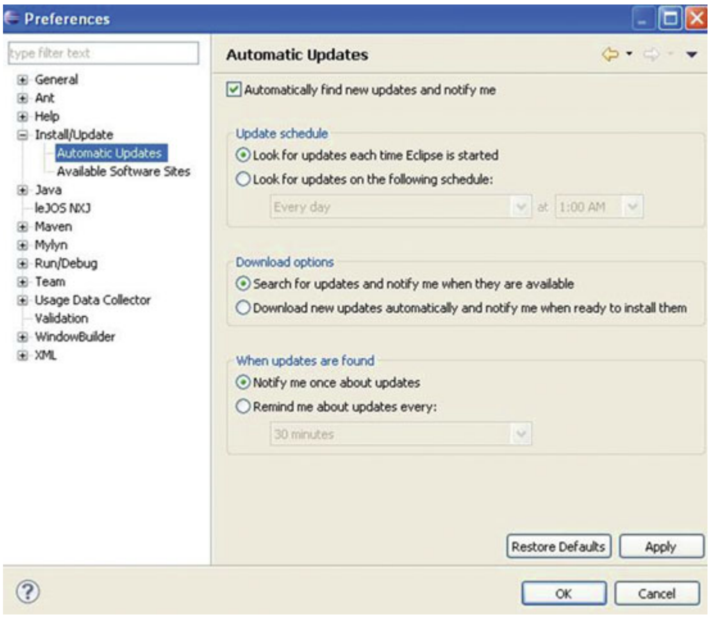
Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\leJOSEV3Proj

☐ Use this as the default and do not ask again

***Figure 1-18.*** *Setting up the workspace for Eclipse*

In case there are software patches or new features, you can set Eclipse up to search automatically for updates by clicking on `Window -> Preferences`, double-clicking `Install/Update` in list, and highlighting *Automatic Updates*. As shown in Figure 1-19, you place a check mark next to *Automatically find new updates and notify me* and then click OK. After that, Eclipse and its plug-ins will be updated automatically.


Preferences

Automatic Updates

☑ Automatically find new updates and notify me

Update schedule
[● Look for updates each time Eclipse is started]
○ Look for updates on the following schedule:
Every day at 1:00 AM

Download options
[● Search for updates and notify me when they are available]
○ Download new updates automatically and notify me when ready to install them

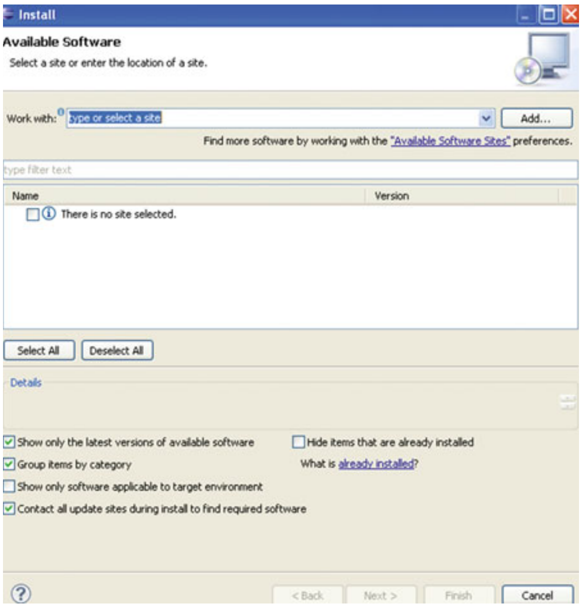
When updates are found
[● Notify me once about updates]
○ Remind me about updates every:
30 minutes

***Figure 1-19.*** *Setting up automatic updates in Eclipse*

## INSTALLING THE ECLIPSE IDE AND ECLIPSE PLU-GIN

Now that you have Eclipse installed, it's time to install the LeJOS EV3 plug-in. Follow these steps:

1. In Eclipse, select `Help > Install New Software`. You will see a dialog requesting that you input a URL, as shown in Figure 1-20.



Available Software
Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

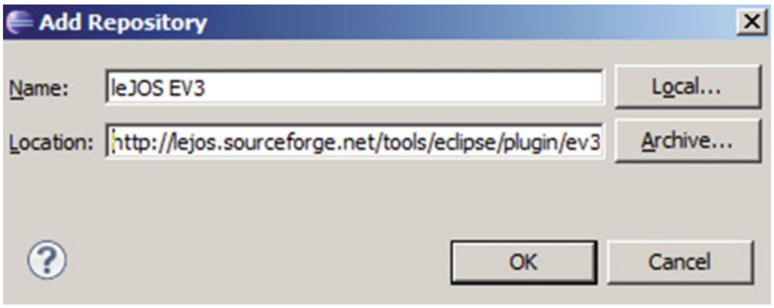
type filter text

| Name | Version |
|---|---|
| ☐ There is no site selected. | |

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

*Figure 1-20.* *Step 1 of installing the leJOS EV3 plug-in*

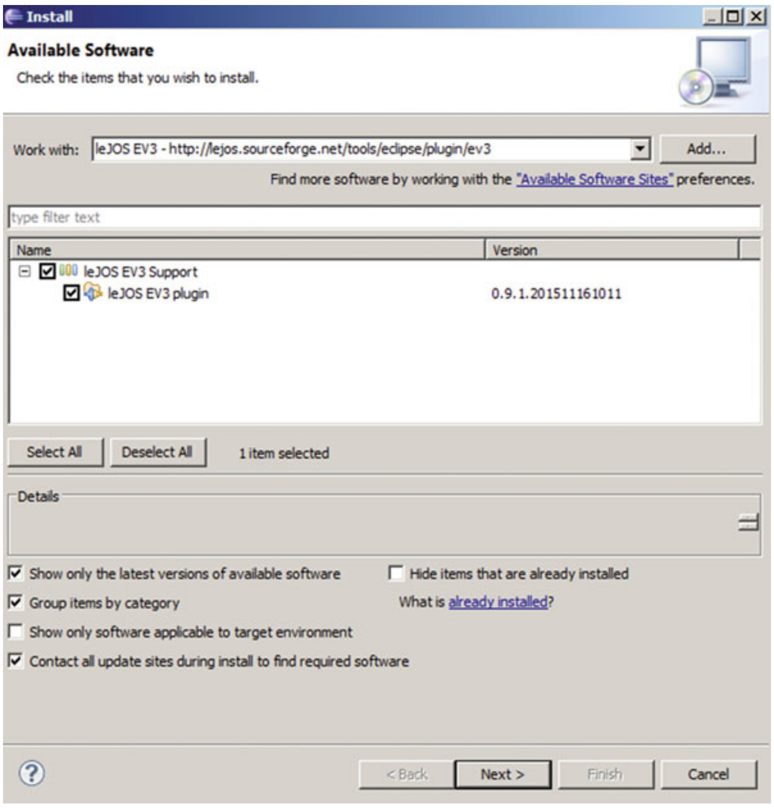
2. Click *Add*, and you will see another dialog box, as shown in Figure 1-21. Enter the name *leJOS EV3*, and for the location enter this: `http://lejos.sourceforge.net/tools/eclipse/plugin/ev3`.



Name: leJOS EV3
Location: http://lejos.sourceforge.net/tools/eclipse/plugin/ev3

*Figure 1-21.* *Step 2 of installing the leJOS EV3 plug-in*

3. Click OK. You should see a new item in the main dialog box, as shown in Figure 1-22. Place a check mark in the box next to the new item, and click the *Next* button.


Install

**Available Software**

Check the items that you wish to install.

Work with: leJOS EV3 - http://lejos.sourceforge.net/tools/eclipse/plugin/ev3

Find more software by working with the "Available Software Sites" preferences.

type filter text

| Name | Version |
|---|---|
| ☑ leJOS EV3 Support | |
| ☑ leJOS EV3 plugin | 0.9.1.201511161011 |

1 item selected

Details

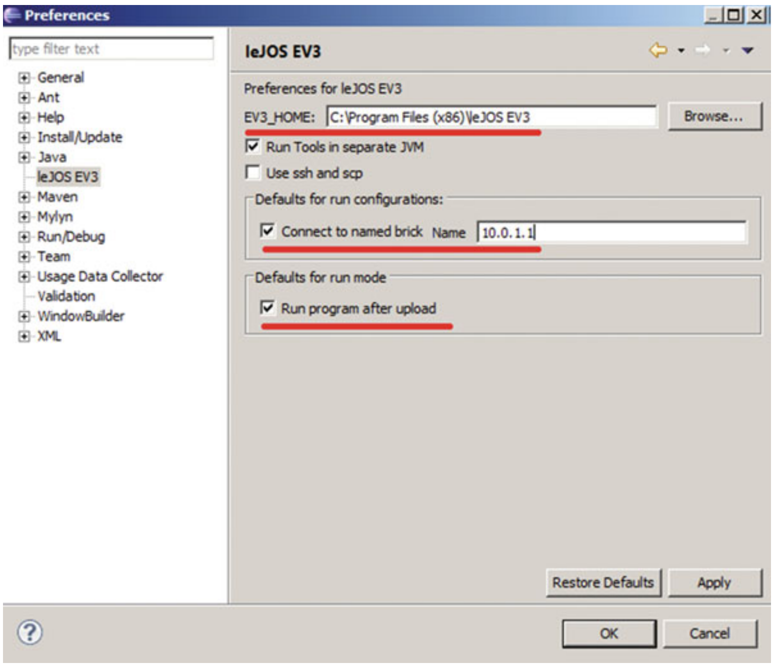
☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

***Figure 1-22.** Step 3 of installing the leJOS EV3 plug-in*

4. Read and accept the license agreement, and click *Next* button. The plug-in will install automatically.

5. When complete, you will be asked to restart Eclipse. Once it has restarted, you will see some subtle changes in Eclipse. The plug-in will add new leJOS menu items to a variety of places within Eclipse.

6. Eclipse will automatically look for the `EV3_HOME` environment variable to locate leJOS EV3. Check to make sure that the preferences are what you like. Select `Windows -> Preferences` and then leJOS EV3 from the list. If the *leJOS EV3* directory is not correct, either type in the location or browse to it. Make sure that you browse to the main directory and not one of its subdirectories. After that, you need to double-check if they are the same as the items illustrated in Figure 1-23.



Preferences for leJOS EV3

EV3_HOME: C:\Program Files (x86)\leJOS EV3

☑ Run Tools in separate JVM

☐ Use ssh and scp

Defaults for run configurations:

☑ Connect to named brick Name 10.0.1.1

Defaults for run mode

☑ Run program after upload

*Figure 1-23.* *Preferences for the leJOS EV3 plug-in*

## CREATING AND UPLOADING A PROGRAM: HELLOWORLD

Now you need to create a place to enter code. Eclipse keeps individual Java projects in its own project directories. For example, if you create a large, multiclass project dealing with mapping, you would create your own project within its own directory to store the class and data files.

In this section, you will create a project that you will use to store code.

1. Select `File > New > Project`.
2. In the next window, double-click leJOS EV3 to expand the folder options. You want to create a leJOS EV3 project, so select leJOS EV3 project and click the *Next* button, as shown in Figure 1-24.

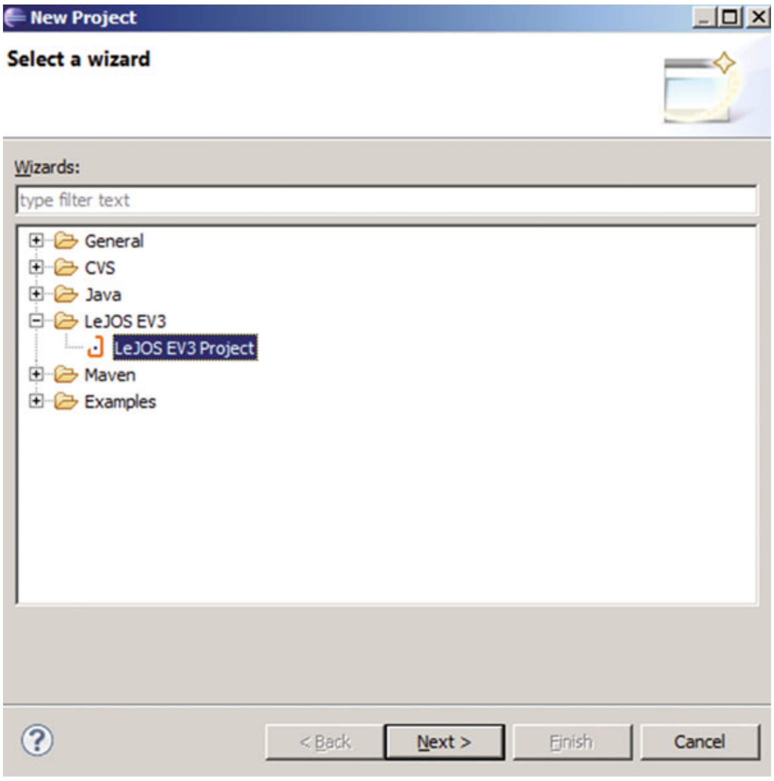
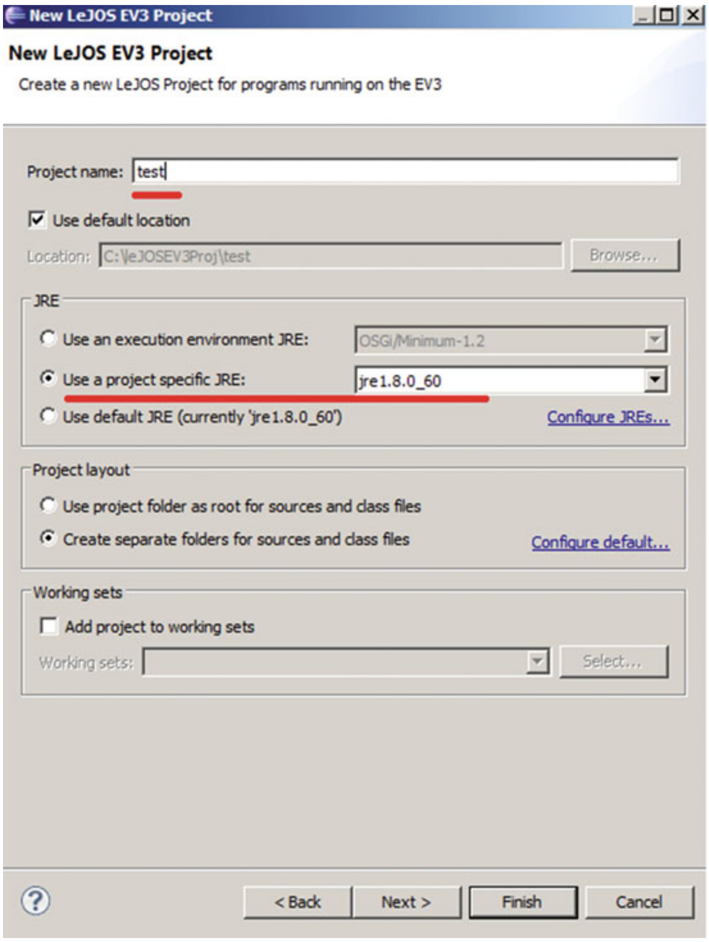


***Figure 1-24.*** *A New leJOS project*

3. For the project name, enter `test` and then click Finish, as shown in Figure 1-25.



Project name: test

[✓] Use default location

Location: C:\leJOSEV3Proj\test

JRE

Use an execution environment JRE: OSGi/Minimum-1.2

[✓ Use a project specific JRE:] jre1.8.0_60

Use default JRE (currently 'jre1.8.0_60')

Project layout

Use project folder as root for sources and class files

[✓ Create separate folders for sources and class files]

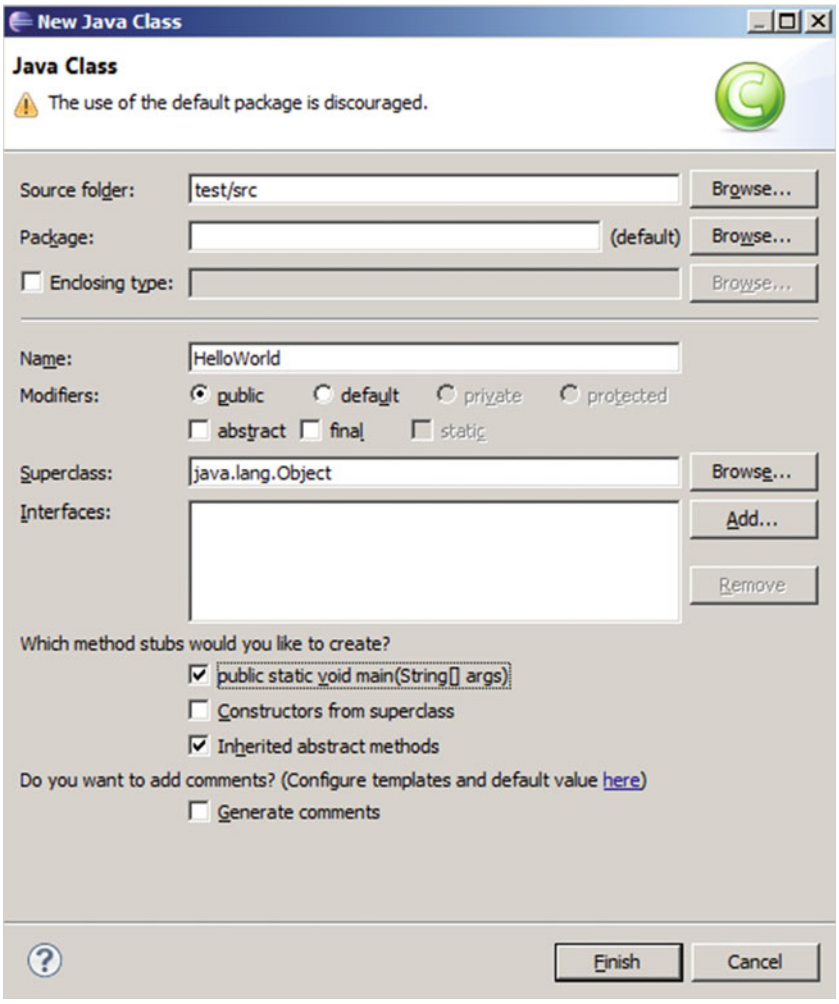
Working sets

[ ] Add project to working sets

Working sets:

***Figure 1-25.*** *Create a new leJOS EV3 project*

4. In order to add a new class file, select `File > New > Class`. Enter `HelloWorld` in the name field, as shown in Figure 1-26. Eclipse will also offer other options, such as automatically adding a `main()` method. Check this if you want Eclipse to do some of the typing for you.



The use of the default package is discouraged.

Source folder: test/src
Package: (default)
☐ Enclosing type:
Name: HelloWorld
Modifiers: [✓ public] default private protected
☐ abstract ☐ final ☐ static
Superclass: java.lang.Object
Interfaces:
Which method stubs would you like to create?
☑ public static void main(String[] args)
☐ Constructors from superclass
☑ Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
☐ Generate comments

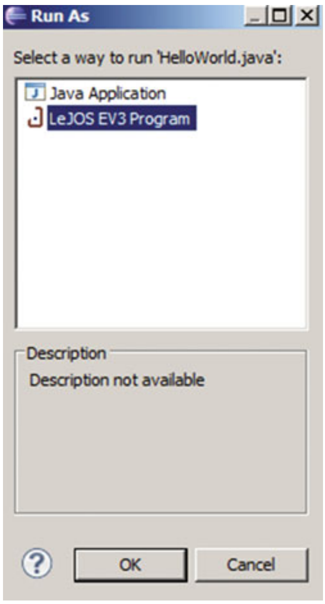
*Figure 1-26.* *Add a class in the new leJOS EV3 project*

5. Click the `Finish` button when you are done. You should see a new class file with some starter code.

Enter the `HelloWorld` code that follows into the file.

```
//***************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 Hello World.java
//An example to display HelloWorld on the LCD screen of EV3 brick
//***************************************************************

// import EV3 hardware packages for EV brick finding,
// activating keys and LCD
import lejos.hardware.ev3.EV3;
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.lcd.TextLCD;

public class HelloWorld {

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantized LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

// drawing text on the LCD screen based on
// coordinates
                lcddisplay.drawString("HelloWorld", 2, 4);

                // exit program after any button pressed
                buttons.waitForAnyPress();
        }

}
```

6. Click the *Save* button, turn on your EV3 brick, and then click the green *Run* button in the Eclipse toolbar. A pop-up window will appear the first time that you click the *Run* button for a class file. Select the leJOS EV3 program, as shown in Figure 1-27, and click OK. The program will begin uploading. You need to make sure that the EV3 brick is connected to your computer through the mini-USB port.


Run As

Select a way to run 'HelloWorld.java':

Java Application
LeJOS EV3 Program

Description
Description not available

***Figure 1-27.*** *Running a leJOS EV3 program*

Your EV3 will show `HelloWorld` at the center of the screen when the program is uploaded and automatically run, assuming that you are using the default settings. If you don't want the program run automatically every time it's uploaded, select `Windows > Preferences > leJOS EV3` and uncheck *Run Program after Upload*.

## Summary

This chapter got you started with installing JDK and leJOS system on your local computer and burning the leJOS EV3 firmware on the Lego Mindstorms EV3 brick using an external SD card. You also ran your first Java robotics program called `HelloWorld` using leJOS and your robot. leJOS can be installed in the leading operating systems such as Windows, Linux, and Mac OS. In particular, you learned the following in this chapter:

- How to install leJOS EV3 software into your Windows operating system with the leJOS installer.
- How to install the leJOS EV3 firmware into your Lego EV3 brick.
- How to install and use the leJOS plug-in for the Eclipse IDE.
- How to write source code in Eclipse and then upload and run the program on your EV3 brick.
- How to install and configure JDK on your local computer.

In the next chapter, you will learn about EV3 large motors and their corresponding motor classes provided in leJOS EV3. Then, based on that, you will study how to control basic movement using motors, how to interrupt rotation, how to regulate the motor speed, and how to trace a straight line.

CHAPTER 2

# Introduction to Motor Sensors

This chapter provides an introduction to EV3 large servo motors and their corresponding motor classes provided in leJOS EV3. In particular, the chapter includes six sample Java projects to cover the following topics:

1. Controlling basic movement using motors
2. Using a Tachometer for inertia testing
3. Controlling the accurate rotation of motors
4. Interrupting rotation
5. Regulating the motor speed
6. Tracing a straight line

## Basic Concepts of Java Programming

The leJOS EV3 infrastructure uses the Java programming language to develop and implement software systems for the Lego Mindstorms EV3 robot. Java is an object-oriented programming language that has been widely used in the software engineering industry. This book does not aim to tutor you on learning the Java programming language. Rather, it is assumed that you already have some basic Java programming experience.

Any Java program has a *main* part that is used to manage all operations. In Java, all files are classes, but there is only one Java class having the method `main`. In the example program `HelloWorld.java`, which you developed in Chapter 1, the public method `main` is illustrated in the following program:

```
// import EV3 hardware packages for EV brick finding,
// activating keys and LCD
import lejos.hardware.ev3.EV3;
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.lcd.TextLCD;

public class HelloWorld {

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();
// instantiated LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

// drawing text on the LCD screen based on
// coordinates
                lcddisplay.drawString("Hello World", 2, 4);

                // exit program after any button pressed
                buttons.waitForAnyPress();
        }

}
```

In the above program listing for `HelloWorld.java`, `lcddisplay.drawString` is used to print the text `Hello World` on the display screen of the Lego EV3 LCD, which is similar to the statement `System.out.println("Hello World")` that prints `Hello World` on the DoS command window.

In this example, `HelloWorld.java` is a simple project with a unique class. In practice, however, a Java project is usually complicated, so often you will have multiple classes, one of which will manipulate the rest of classes. In summary, any Java class includes an import area, the class encapsulation, and the `main` method. The import area declares what Java package you are going to use in a particular class. For example, in the `HelloWorld.java` program, you use EV3 Keys and LCD features and then it is required that you import the package `lejos.hardware.Keys` and `lejos.hardware.lcd.*`. In the `main` method, the `main` class needs a `main` method to execute it. In the `HelloWorld` class example, it has a `main` class used to execute some instructions; that is, `lcddisplay.drawString("Hello World", 2, 4)` to print `Hello World` at row 2 and column 4 on the LCD screen and `buttons.waitForAnyPress()` to exit program after pressing any buttons on the EV3 brick.

## Introducing Motors

*Motors* are the source of all movement in the Lego EV3 kit, and they play an essential role in all robotics projects in this book. Therefore, this chapter will familiarize you with the operations of the motor classes provided in leJOS EV3.

Using the motor algorithms implemented in the leJOS EV3, you can set up the EV3 robot rotating precisely to a specified number of degrees, say 240, without crossing over this target number. Moreover, you can set up a constant speed for a wheeled vehicle, and this can be achieved by tuning the power level in real time so that when the robot is moving up or down, it can maintain the same speed that you established. In addition, the motor classes enable the motors to accelerate to full speed by initially starting at a slow speed and constantly accelerating to reach the specified speed. As a result, you can command the motors to advance in a direction for thousands of rotations and then have them return to the original point at any time. Such features open up unlimited ideas for creative navigation and rotation of arms for EV3 robots.

To work with the three motors provided in the Lego EV3 robotics set, you have to import the `lejos.hardware.motor` package. This package comes with four fields: A, B, C, and D for the four motor ports, as well as a number of methods. For details on doing this, you can check out the Java API function of leJOS EV3 at: `http://www.lejos.org/ev3/docs/`. For example, you can set the speed of a motor hooked to port A by using the following statements:

```
EV3LargeRegulatedMotor LEFT_MOTOR =
new EV3LargeRegulatedMotor(MotorPort.A);
LEFT_MOTOR.setSpeed(720);
```

This code will set the speed of the `LEFT_MOTOR` so that it will roll 720 degrees per second (that is, two complete rotations per second). If you want to have this motor go forward, you say the following:

```
LEFT_MOTOR.forward();
```

Next, six example Java projects are provided to introduce the motor classes for controlling the motors. You need to compile, upload, and run all six example programs to complete all of the tasks in this chapter.

## Introducing the Motor Class

The motor class provides access to the EV3 large servo motors. When controlling the movement of EV3, a motor must be connected to one of the four EV3 motor ports. The class provides an instance for each port, namely, MotorPort.A, MotorPort.B, MotorPort.C, and MotorPort.D.

Each of these four objects is an instance of the class `EV3LargeRegulatedMotor`, which provides methods for controlling the motors. In this section, you are given a set of six programs, and by using them, you can perform experiments to understand how the EV3 large motors perform. These programs are simple enough, so you don't need much Java programming experience to write them. Nonetheless, they still allow you to gain a basic understanding on programming and controlling the motors' movements.

### Controlling Basic Movement Using Motors

This program uses the basic motor methods that control motor movement. Methods used in this program include those shown in Table 2-1:

***Table 2-1.*** *Basic Motor Methods*

| Class | Method | Function |
|---|---|---|
| EV3LargeRegulatedMotor | `forward()` | Motor rotating forward. |
| | `backward()` | Motor rotating backward. |
| | `stop()` | Motor stopping quickly. |
| Keys | `waitForAnyPress` | Wait till any key is pressed. |
| LCD | `drawString(String str, int x, int y)` | Print a text string on the LCD at coordinate row x and column y. |

The program should do the following:

1. Run motors A and C in the forward direction.
2. Display FORWARD in the top line.
3. Wait until a button is pressed.
4. Run motors A and C backward.
5. Display BACKWARD in next line.
6. Wait until a button is pressed.
7. Stop both motors A and C.

The following program implements the motor movements defined above:

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 p1.java
//Simple motor testing
//*****************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;

public class p1 {

        public static void main(String[] args) {

EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

                // instantiated LCD class for displaying
// and Keys class for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // move robot forward and display status on LCD
                // change directions when button is pressed
                LEFT_MOTOR.forward();
                RIGHT_MOTOR.forward();
                LCD.drawString("FORWARD", 0, 0);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // move robot backward and display status on LCD
                LEFT_MOTOR.backward();
                RIGHT_MOTOR.backward();
                LCD.drawString("BACKWARD", 0, 1);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // stop robot and display status on LCD
                LEFT_MOTOR.stop();
                RIGHT_MOTOR.stop();
                LCD.drawString("STOP", 0, 2);
                // exit program after any button pressed
                buttons.waitForAnyPress();

        }
}
```

## Using a Tachometer for Inertia Testing

The EV3 large servo motor has a built-in tachometer that keeps track of the current angle (in degrees) of the motor axle. The purpose of this program is to use the tachometer to find out how quickly the motor stops.

The program should do the following:

1. Set the motor speed to 720.
2. Run MotorPort.A forward.
3. Wait till the tachometer count reaches 720.
4. Stop the motor.
5. Display the tachometer reading on the LCD.
6. Wait until the motor has actually stopped.
7. Display the tachometer reading again on the LCD.
8. Wait for a button press to give you time to record the screen display.

Due to the inertia of the motors, you will find out that the motor does not stop immediately after you call the method `stop()`. New methods used in this program include those shown in Table 2-2:

***Table 2-2.*** *Methods Used in Tachometer*

| Class | Method | Function |
|---|---|---|
| EV3LargeRegulatedMotor | `getTachoCount ()` | Gets the motor angle in number of degrees. |
| | `resetTachoCount ()` | Resets the counter number to 0. |
| | `setSpeed(int speed)` | Sets up the rotation speed in number of degrees per second. |
| | `getRotationSpeed()` | Gets the actual speed of motor in number of degrees per second. |
| | `clear()` | Clears the LCD screen. |

```
//***************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 p2.java
//Motor inertia test
//***************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;

public class p2 {

        public static void main(String[] args) {

EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // set motor to move 720 degrees per second
                LEFT_MOTOR.setSpeed(720);

                // start forward movement
                LEFT_MOTOR.forward();

// a counter to count the number of degrees
// rotated
                int count = 0;

// continue moving until motor has rotated 720
// degrees
                while (count < 720)
                        count = LEFT_MOTOR.getTachoCount();

                // stop the motor
                LEFT_MOTOR.stop();

                // display the tachometer reading
                LCD.drawString("Tacho Read: " + count, 0, 0);

// wait for motor to actually stop and display
// tacho count.
// this number will be higher than previous due // to motor inertia
                while (LEFT_MOTOR.getRotationSpeed() > 0);
LCD.drawString("Tacho Read: " + LEFT_MOTOR.getTachoCount(), 0, 1);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                LCD.clear();
        }
}
```

After running the above program, you will find two numbers printed on the LCD screen. The two numbers are different. The first number displays what the tachometer reading was when the `stop()` method was called. You would expect this number to be 720, since the program waits for the tachometer to reach 720 before it calls method `stop()`.The second number is higher than the first, however. This is because the motor does not stop immediately when the `stop()` method is called, due to the motor's inertia.

## Controlling the Accurate Rotation of Motors

The *Motor Class* provides a regulator thread that runs all the time, which you can use to stop the motor at a specified angle. In this example, you will run a test to gauge the accuracy of the `rotate()` method. The methods used in this program include those shown in Table 2-3:

***Table 2-3.*** *Methods Used in the Motor Class Regulator Thread*

| Class | Method | Function |
|---|---|---|
| EV3LargeRegulatedMotor | `rotate(angle)` | Rotate through the number of degrees of the defined *angle*. |
| | `rotateTo(angle)` | Rotate to the specified *angle*. |

The program should do the following:

1. Set the speed to 720.
2. Rotate the motor one complete revolution.
3. Display the tachometer reading on the LCD, row 0.
4. Rotate the motor to angle 360.
5. Display the tachometer reading on the LCD, row 1.
6. Wait for a button press to give you time to read the LCD.
7. Clear the LCD.

The motor usually stops less than 1 degree from the specified angle according to the motor regulator. This is done by calculating how far the motor will continue to run after the brake has been applied. The brake is applied before reaching the specified angle, and then a minor adjustment is made to fine-tune the motor position until it is close enough.

```
//****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 p3.java
//This program demonstrated motor rotation control
//****************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
```

```
public class p3 {

        public static void main(String[] args) {

EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();
// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();
                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // set motor to move 720 degrees per second
                LEFT_MOTOR.setSpeed(720);

                // rotate the motor one full revolution
                LEFT_MOTOR.rotate(360);

                // display the tacho count in row 0
LCD.drawString("Tacho Read: " + LEFT_MOTOR.getTachoCount(), 0, 0);

                // rotate to the angle 360
                LEFT_MOTOR.rotateTo(360);

                // display the tacho count in row 1
LCD.drawString("Tacho Read: " + LEFT_MOTOR.getTachoCount(), 0, 1);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                LCD.clear();
        }
}
```

By running the above program on the Lego EV3, the first tachometer reading will probably be 360 and the second 359 (or 360 depending on the motors). These two numbers are within a degree of each other. The first step in the program is to rotate one full rotation. Then it should set the motor to position 360. Since one rotation is 360 degrees, it is then expected that these numbers will be close. Through demonstration of the program, you can observe that better precision can be achieved by moving to a certain position than by calling the `stop()` method. The motor class predicts when the motor should stop and applies the brake earlier to make up for the impact of inertia.

## Interrupting Rotation

Sometimes you will want the motor to stop and something else before it has reached a specified angle. In this example program, you will have code detect a button press to interrupt the rotation job if you press a button soon enough. The `rotate()` methods will not return until the motor has stopped at the target angle. However, the new methods you use in this program can return immediately. Some new methods used in this program include those shown in Table 2-4:

***Table 2-4.*** *Rotate Method Used in the Motor Class*

| Class | Method | Function |
|---|---|---|
| EV3LargeRegulatedMotor | `rotate(angle, immediateReturn)` | Rotate through the number of degrees of the defined *angle*, and on the mean time if the value of `immediateReturn` is true, the method returns immediately. |
| | `rotateTo(angle, immediateReturn)` | Rotate to the specified *angle*, and on the mean time if the value of `immediateReturn` is true, the method returns immediately. |
| | `(boolean) isMoving()` | Return true if the motor is always rotating. |
| | `int readButtons()` | Return the button id number if any button is pressed. |

The program should do the following:

1. Start a rotation of 7,200 degrees.
2. While the motor is rotating, display the Tachometer count at the position of row 0.
3. When a button is pressed, stop the motor.
4. After the motor has stopped, display the Tachometer count at the position of row 0.
5. Record the two numbers read from Tachometers, and then wait for a button press to exit the program.

When you press the button before the rotation is complete, the motor will stop without completing its rotation.

```
//*************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 p4.java
//interrupting motors using buttons
//*************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.Sound;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;

public class p4 {

        public static void main(String[] args) {

EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();
// instantiated LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();

                // sound two beeps before starting program
                Sound.twoBeeps();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// rotate 7200 degree during which the method has // returned value all the time
                LEFT_MOTOR.rotate(7200, true);

                // return to true if the motor is always rotating
                while (LEFT_MOTOR.isMoving()) {

// display and refresh the tachometer reading all // the time
LCD.drawString("Tacho Read: " + LEFT_MOTOR.getTachoCount(), 0, 0);

                        // determining if there is any button
// pressed, if yes then stop the motor
                        if (buttons.readButtons() > 0)
                                LEFT_MOTOR.stop();
                }

                // wait until the motor fully stopped
                while (LEFT_MOTOR.getRotationSpeed() > 0)
                        ;

                // display the tachometer reading after motor
// fully stopped
LCD.drawString("Tacho Read: " + LEFT_MOTOR.getTachoCount(), 0, 1);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

        }
}
```

## Regulating the Motor Speed

The motor class has a regulator thread to control motor speed. The reason behind for doing this is that a two-wheel vehicle will travel in a straight line only if both motors run at the same speed. The leJOS EV3 keeps each motor rotation synchronized to the system clock, and therefore the regulator compares the tachometer count with the speed multiplied by the elapsed time. It then adjusts the power to keep these two numbers matched as closely as possible. Some new methods used in this program include those shown in Table 2-5:

*Table 2-5. Regulator Methods Used in the Motor Class*

| | | Function |
|---|---|---|
| Stopwatch | `elapsed()` | Returns elapsed time in milliseconds. |
| | `reset()` | Reset the watch to 0. |

The Stopwatch class is contained in the package `lejos.util.Stopwatch`. The program should do the following:

1. Create a new stopwatch.
2. Start the two motors A and C running at 1 revolution/second. (One revolution is 360.)
3. Every 200 milliseconds, display all two tachometer count values in the same row.
4. Repeat step 3 four times, using a different row each time.
5. Print the maximum difference that you see between the motor tachometer counts.

The motors should remain within a few degrees of each other according to your observation, since we used the regulated large motor EV3 class.

```
//*********************************************************
//Wei Lu Java Robotics Programming with Lego EV3 p5.java
//Motor speed control comparison
//*********************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.utility.Stopwatch;

public class p5 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();
                // the row to print on
                int tachoRow = 0;

// instantiated a stopwatch class for setting up // the timer
                Stopwatch sw = new Stopwatch();

                // set the motor speed to 1 revolution per second
                LEFT_MOTOR.setSpeed(360);
                RIGHT_MOTOR.setSpeed(360);

                // motors moving forward
                LEFT_MOTOR.forward();
                RIGHT_MOTOR.forward();

                // variables for determining the maximum
// difference in tacho counts
                int maxTachoCountDiff = Integer.MIN_VALUE;
                int currTachoCountDiff;

                // perform four repetitions of the test
                for (int i = 0; i < 4; i++) {
                        // wait for 200 milliseconds
                        sw.reset();
                        while (sw.elapsed() < 2000)
                                Thread.yield();

// display the tacho counts and reset the // max, if changed
currTachoCountDiff = displayTachoCounts(tachoRow++);
if (currTachoCountDiff > maxTachoCountDiff)
                        maxTachoCountDiff = currTachoCountDiff;

                }

                // stop the motors
                LEFT_MOTOR.stop();
                RIGHT_MOTOR.stop();

                // display the maximum difference in tacho
// counts, then wait for exit
LCD.drawString("Max diff: " + maxTachoCountDiff, 0, tachoRow);

                buttons.waitForAnyPress();
        }

        /**
         * Displays the tachometer count for each motor
         *
         * @param row to print the count in
         *
 * @return Returns the difference between the tacho
 * counts of the two motors
         */
        private static int displayTachoCounts(int row) {
                // store the tacho counts for the two motors
                int tachoCountLeft = LEFT_MOTOR.getTachoCount();
int tachoCountRight = RIGHT_MOTOR.getTachoCount();

                // display the tacho counts
LCD.drawString("M1: " + tachoCountLeft + " M2: " + tachoCountRight, 0, row);

                // return the difference in the tacho counts
return Math.abs(tachoCountLeft - tachoCountRight);
        }
}
```

By running the program above, you should observe a result similar to the following:

```
Motor 1: 711        Motor 2: 710
Motor 1: 1493        Motor 2: 1492
Motor 1: 2212        Motor 2: 2212
Motor 1: 2934        Motor 2: 2934
Max diff: 1
```

The maximum difference between the two tachometer readings is 1, so you can tell that the two motors are virtually synchronized.

## Tracing a Straight Line

In this practice exercise, you need to write a program to run the robot forward for some predetermined amount of time (say 10,000 ms) and measure how far the robot traveled. The program should do the following:

1. Create a new stopwatch.
2. Start the two motors A and C running forward.
3. Calculate the elapsed time until it reaches 10,000 ms, displaying the elapsed time on the LCD screen.
4. Stop the two motors.
5. Calculate the ratio of distance to time in centimeters per second.

Repeat your program three times, and then calculate an average ratio of distance to time, which is the speed of robot.

```
//******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 p6.java
//Write a program to run the robot forward for some predetermined
//amount of time (say 10000 ms) and measure how far the robot went.
//******************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.utility.Stopwatch;

public class p6 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class  // for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // instantiated a stopwatch class for setting up the
// timer
                Stopwatch watch = new Stopwatch();

                // Begin running both motors
                LEFT_MOTOR.forward();
                RIGHT_MOTOR.forward();

                // Clear the screen
                LCD.clear();

                // Reset the time on the watch
                watch.reset();

                // Display the elapsed time on the LCD until 10000ms
                while (watch.elapsed() < 10000) {
                        Thread.yield();
                        LCD.drawString("" + watch.elapsed(), 0, 0);
                }

                // Stop the motors after 5 seconds
                LEFT_MOTOR.stop();
                RIGHT_MOTOR.stop();

                buttons.waitForAnyPress();
        }
}
```

On average, the robot traveled 99.7 cm in 10,000 milliseconds, so the robot averages a speed of 9.97 cm/sec while traveling forward at 1 revolution per second. Taking into consideration a tread diameter of approximately 3.1 cm and a motor speed set to 360 degrees/second, we can confirm that this result is pretty close to what we would expect; that is, 3.1 cm * 3.1415926 = 9.74 cm/s, which is pretty close to the speed that we observed; that is, 9.97 cm/sec.

## Other Motor Methods

There are many other methods provided with the EV3 large motors. Some of these methods are explained in Table 2-6:

***Table 2-6.*** *Other Methods*

| Method | Function |
|---|---|
| `boolean isMoving()` | This method is useful to test if the motor has finished rotating. `isMoving()` returns true when the motor is moving for any reason. For example, if a `forward()` or `backward()` method has been called, or if a `rotate()` task is in progress. |
| `int getLimitAngle()` | Returns the angle in number of degrees to which the motor is currently rotating. |
| `int getSpeed()` | Returns the current speed settings. |
| `int getRotationSpeed()` | Returns the current velocity of the motor in number of degrees per second. |
| `boolean isStalled()` | Helps you determine if the motor is stalled, or if the regulation of the motor speed has failed. |
| `resetTachoCount()` | This method sets the tachometer count to 0, it resets the origin on the mean time used by the regulator thread in deciding when to stop a rotation task. |
| `void setAcceleration(int acceleration)` | This method helps you to control how fast the motor speed will change from one speed to another. Acceleration is set in number of degrees per second. |
| `void getAcceleration()` | Returns the current acceleration value in number of degrees per second for the motor. |
| `suspendRegulation()` | Turns on/off the regulation of the motor. You can use this method if you want to mix regulated and unregulated control of the same motor. |

## Summary

In this chapter, you learned about EV3 motor features: how the motors rotate, how to set the speed of the motors, and the basic idea of speed regulation. Specifically, by using the six example Java leJOS projects provided in the chapter, you learned how to write and apply leJOS Java programming code to control and operate the Lego Mindstorms EV3 motors.

In the next chapter, you will learn about the various methods provided in the Pilot classes of leJOS EV3. You will also study how to apply these methods to control the wheeled vehicle so that it can trace out a predefined geometric shape with sides of a predefined length, including a square, triangle, and hexagon.

CHAPTER 3

# Controlling Wheeled Vehicles

This chapter introduces various methods provided in the *MovePilot* class of leJOS EV3. You will learn know how to apply these methods in the pilot classes to control the wheeled vehicle so that it can trace out a predefined geometric shape with sides of a predefined length, including a square, triangle, and hexagon. Specifically, this chapter includes nine example Java projects and covers the following topics:

- Introduction to the navigation API
- Basic movement using pilot classes
- Tracing out a square using `movepilot` and `differentialpilot`
- Tracing out a triangle using `movepilot` and `differentialpilot`
- Tracing out a hexagon using `movepilot` and `differentialpilot`

## Introduction to Navigation API

You studied how to create a simple control for the EV3 motors in Chapter 2. As an example, review the following motor testing program, `example1.java`:

```
//**************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example1.java
//an example for motor testing
//displaying tachocount about how many degrees rotated when //pressing //ESCAPE button.
//**************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;

public class example1 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
```

© Wei Lu 2016
W. Lu, *Beginning Robotics Programming in Java with LEGO Mindstorms*, DOI 10.1007/978-1-4842-2005-4_3

```
public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

                // instantiated LCD class for displyang and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                String message = "MOTOR Testing: ";

// set up the motor A speed, i.e. 100 degrees per // second
                LEFT_MOTOR.setSpeed(200);

                // motor A moving forward
                LEFT_MOTOR.forward();

                // displaying number of degrees rotated on the
// LCD until an ESCAPE button is pressed!

                while (buttons.getButtons() != Keys.ID_ESCAPE) {
                        LCD.clear();
                        LCD.drawString(message, 0, 1);
                                LCD.drawInt(LEFT_MOTOR.getTachoCount(), 0, 2);
                        Thread.sleep(1000);
                        LCD.refresh();
                }
        }
}
```

As illustrated in `example1.java`, when developing an EV3 robotics program to control motors, you need to indicate which motor you need to program for any action. In this example, you set the speed of Motor A with the following instruction:

```
static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);

LEFT_MOTOR.setSpeed(200);
After that you indicate that Motor A will turn forward using the following instruction:
LEFT_MOTOR.forward();
```

Usually, if you simply use a unique motor, the methods that you could use are as follows:

1. `forward()`
2. `backward()`
3. `rotate()`
4. `rotateTo()`
5. `setSpeed()`
6. `stop()`

In actual use, a vehicle is usually equipped with two wheels that are controlled and navigated independently with two motors. Such navigation is achieved by leJOS navigation API functions through which a set of classes and methods are provided to control the robot. The leJOS NXJ navigation classes that control vehicles deal with a hierarchical level of abstractions.

At the bottom level, the `NXTRegulatedMotor` class is created to control the motors that turn the wheels. A `DifferentialPilot` class is then used to control elementary moves of motors such as rotating in place, traveling in a straight line, or traveling in an arc.

At the top level, the `NavPathController` applies a `DifferentialPilot` to move the robot through a complicated path on a plane. An `OdometeryPoseProvider` is also used to record the information on the robot location and the direction to which it is heading. In such a hierarchical structure, the flow of control is based on a top-down approach; that is, the path controller controls the pilot and then the pilot controls the motors. In this case, the flow of information is from the bottom up in which the pilot collects information from the motors to control them. The pose provider uses odometer readings information collected from the pilot to update its current estimate on the robot pose, including the Cartesian coordinates (that is, axis x and axis y) of the robot and its heading angle (that is, the direction the robot is facing based on the number of degrees — facing the direction of the positive x axis has 0 degrees, and facing the direction of the positive y axis has 90 degrees). All this information is used by the path controller to calculate the distance to its destination and the direction. In the leJOS EV3, a `MovePilot` class is used to replace the `DifferentialPilot`. However, the fundamental idea remains the same.

## Basic Movement Using Pilot Classes

This chapter will cover basic movement using Pilot classes, which is one of the most important steps in navigation. As you have seen in Chapter 2, it is possible to move a robot around merely using the motor classes: that is, simply by rotating the motors forward and backward. However, to drive to specific locations (that is, to do a precise movement), it is necessary to have controls over the distances a vehicle moves forward or backward and the angles the robot turns. Therefore, the main goal for basic movement using pilot is to create vehicles that can perform precise moves. The pilot class is used to drive, steer, and turn a vehicle precisely, including straight-line travel, on-the-spot rotation, arcs, and stop.

By packaging a sequence of these movements together, a robot can travel from one location to another by repeatedly performing a combination of two essential steps, such as the amount the robot travels forward or backward and the amount the robot rotates clockwise and counterclockwise throughout the movement.

One of the advantages of using pilot class is that the actual physical characteristics of the robot are hidden in the navigator package except for the pilot. From the perspective of internal movement, a robot can roll, walk, jump, or fly from one location to another. However, from the external perspective, you only see the external pilot methods to make the robot movements all look the same. As a result, using pilot allows you to program diverse types of robots to participate in navigation, regardless of their physical construction.

In leJOS, a `Move` class is used to tell a robot what moves to make, or to indicate what kind of movement a robot just made. Some of the core methods of the `Move` class include the following:

| | |
|---|---|
| `getDistanceTraveled ()` | Obtains the distance the vehicle moves, normally in number of centimeters. |
| `getTurnAngle ()` | Gets the angle the vehicle rotated over the movement in number of degrees. |
| `getArcRadius ()` | Receives the radius of the arc that the vehicle travels. |

Next you will find out how to perform the movements with a physical robot. In leJOS, a pilot is a class that controls a specific robot. In particular, the `DifferentialPilot` class can control a robot with two wheels. In this chapter, you will use differential motor control because it is capable of all of the moves and it is simple to build the chassis for it. The `DifferentialPilot` class steers the vehicle by controlling the speed and direction of rotation of its motors. The pilot object needs to know to which ports the motors are connected and whether driving the motors forward makes the robot move forward or backward. The object also needs to know the diameter of the wheels and the width of the track: that is, the distance between the centers of the tracks of the two wheels. The reason for this is that the `DifferentialPilot` class uses the wheel diameter to calculate the distance it has traveled and uses the track width to calculate how far it has rotated. All of this information will be passed to the pilot constructor that is illustrated in the following line of code:

```
DifferentialPilot(float wheelDiameter, float trackWidth, Motor leftMotor, Motor rightMotor)
```

Otherwise, use the following constructor to set the Boolean variable *reverse* to true so that the motors will then rotate backward to make the robot move forward:

```
DifferentialPilot(float wheelDiameter, float trackWidth, Motor leftMotor, Motor rightMotor,
boolean reverse)
```

In conclusion, in order to control the robot moving in a straight line, you can use the following methods:

`void setTravelSpeed(double travelSpeed)` Sets the speed of the motors in the same unit of distance (for example, if diameter of the wheel is measured in centimeters, then the number set by this method will be in centimeters per second)

`void forward()` Starts the robot moving forward

`void backward()` Starts the robot moving backward

`void stop()` Stops the robot moving

`void rotate (double angle)` Rotates the robot in a number of degrees

In order to control the distance that the robot moves, you can use the following methods:

`void travel(double distance)` Moves the motors a specified distance in the same unit as the wheel diameter (that is, if diameter of the wheel is measured in centimeters, then the number set by this method will be in centimeters)

`getMovement().getDistanceTraveled()` Returns the distance the vehicle traveled

For instance, `example2.java` is illustrated below to show you how to make the robot move 20 centimeters using the `DifferentialPilot` class in leJOS NXJ.

```
//***************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example2.java
//an example for making basic movement using DifferentialPilot //in leJOS NXJ
//***************************************************************************

import lejos.nxt.*;
import lejos.robotics.navigation.*;

public class example2 {

        public static void main(String[] args) {

DifferentialPilot pilot = new DifferentialPilot(3.1, 17.5, Motor.A,Motor.C);
                pilot.travel(20, true); // 20 centimeters
pilot.rotate(90.0);
while (pilot.isMoving()) {

                        if (Button.ESCAPE.isPressed())
                                pilot.stop();
                        Button.waitForPress();

                }
        }
}
```

As seen from the example2.java, you can cause the robot to rotate in place by a specified angle by using the following:

```
void rotate(double degrees)
```

Moreover, you need to measure accurately the values for wheelDiameter and trackWidth when using this method to produce precise movement. The diameter is the widest measurement from one side of a circle to the other. According to my measurements, the wheel diameter on my robot is 3.1 centimeters. In order to record accurate rotations when the robot turns, it's also important to know the measurement from wheel to wheel, known as the *track width*. Since Lego tires are symmetrical, the best way to do this is to measure from the center of one tire to the center of the other tire. The track width in the above example is 17.5 centimeters, according to my measurements. The final parameters for the DifferentialPilot are the motors for the left and right wheels. In the above example program, the left motor is connected to port A and the right motor is connected to port C.

Since MovePilot is used to replace DifferentialPilot in the new leJOS EV3, the following example, example3.java, is provided to show you a basic movement control using MovePilot in leJOS EV3.

```
//*************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example3.java
//an example for making basic movement using MovePilot in //leJOS EV3
//*************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.*;

public class example3 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();
// instantiated LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// setup the wheel diameter of left (and right) motor // in centimeters, i.e. 2.8 cm
// the offset number is the distance between the center   // of wheel to
        // the center of robot, i.e. half of track width
Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

        // set up the chassis type, i.e. Differential pilot
Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },WheeledChassis.TYPE_
DIFFERENTIAL);
        MovePilot pilot = new MovePilot(chassis);

        // travel 100 centimeter
        pilot.travel(100);

        // rotate 90 degrees
        pilot.rotate(90.0);

        // press the ESCAPE button to stop moving
        while (pilot.isMoving()) {

                if (buttons.getButtons() == Keys.ID_ESCAPE)
                        pilot.stop();

                }

        // block the thread until a button is pressed
        buttons.waitForAnyPress();
        }
}
```

## Tracing Out a Square

In this section, you will develop a program to create a robot that traces out a square with sides of a predetermined length set to 1 meter. You will write, compile, and upload your code into Lego Mindstorms. In addition, you will also need to work out how long to run the motors to move forward a specified distance of *N* centimeters; for example, *N* is set to 100 centimeters in this case. Furthermore, you will work out how long to run the motors to rotate 90 degrees (the corners of the square).

An example program, `example4.java`, illustrates how to trace out a square using `MovePilot` in leJOS EV3.

```
//**************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example4.java
//an example for tracing out a square
//**************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.*;

public class example4 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// setup the wheel diameter of left (and right) // motor in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the // center of wheel to
                // the center of robot, i.e. half of track width
Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
Wheel wheel2 =
WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

        // set up the chassis type, i.e. Differential pilot
Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
        WheeledChassis.TYPE_DIFFERENTIAL);

MovePilot pilot = new MovePilot(chassis);

                // loop 4 times to trace out a square
                for (int sides = 0; sides < 4; sides++) {

                        // travel 100 centimeter
                        pilot.travel(100);
                        // rotate 90 degrees
                        pilot.rotate(90);
                }

                // press the ESCAPE button to stop moving
                while (pilot.isMoving()) {

if (buttons.getButtons() == Keys.ID_ESCAPE)
                                pilot.stop();

                }

                // block the thread until a button is pressed
                buttons.waitForAnyPress();
        }
}
```

In case you use leJOS NXJ, an example program, `example5.java`, follows to illustrate how to trace out a square using a set of methods provided in the `DifferentialPilot` class. The idea behind this is based on a benchmarking test showing that a vehicle could travel forward at a speed of 30.76 cm/second. Accordingly, the vehicle needs to travel for 3.25 seconds to complete one side of the square. To implement this, we applied a `sleep` function, which starts by moving the robot forward, waiting 3.25 seconds, and then stopping. Experimental results of evaluating the program showed that this algorithm is more effective than using the methods of the `DifferentialPilot` class directly.

```
//**************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example5.java
//an example for tracing out a square using DifferentialPilot in //leJOS NXJ
//**************************************************************************

import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.nxt.Motor;
import lejos.nxt.NXTRegulatedMotor;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.util.Stopwatch;

public class example5 {

// The vehicle's rate of travel forward in centimeters per
// second
        private static final double FW_SPEED = 30.666666667f;

// The vehicle's clockwise rotation speed in degrees per
// second
        private static final double ROTATION_SPEED = 248.297752809f;

        // The vehicle's wheel-width.
        private static final double WHEEL_DIAMETER = 3.1;

        // The vehicle's width
        private static final double TRACK_WIDTH = 17.5;
        // Reference to left motor
        private static NXTRegulatedMotor LEFT_MOTOR = Motor.A;

        // Reference to right motor
        private static NXTRegulatedMotor RIGHT_MOTOR = Motor.C;

        /**
         * The main entry point for the program.
         */
        public static void main(String[] args) {

                // construct the pilot using the static variables
DifferentialPilot pilot = new DifferentialPilot(WHEEL_DIAMETER,
TRACK_WIDTH, LEFT_MOTOR, RIGHT_MOTOR, false);

                // tracing out a square
                drawSquare(pilot, 100);

LCD.drawString("tracing is done, press button to exit.", 0, 0);
        Button.waitForPress();
        }

        /**
         * Draws a square of the specified side length
         *
         * @param pilot
         *            The pilot to use when drawing the square
         * @param sideLength
         *            The length of each leg of the square
         */
public static void drawSquare(DifferentialPilot pilot, double sideLength) {

                for (int i = 0; i < 4; i++) {

                        // delay 500ms to allow the motors to stabilize
                        sleep(500);

                        // move the vehicle forward
                        forward(pilot, sideLength);

                        // delay 500ms to allow the motors to stabilize
                        sleep(500);

// rotate the vehicle by 90 degrees to prepare
// for
                        // the next side of the square
                        rotate(pilot, 90);
                }
        }
        /**
         * Move the vehicle forward the specified distance
         *
         * @param pilot
         *            The pilot to use for moving forward
         * @param distance
         *            The distance to travel
         */
public static void forward(DifferentialPilot pilot, double distance) {
// get the number of milliseconds the vehicle should
// travel based on the
                // vehicle's speed
                        int travelTime = getMillisForTravel(distance);

                        // begin the pilot forward
                        pilot.forward();

                        // wait for the traveling to finish
                        sleep(travelTime);

                        // reached the destination - stop
                        pilot.stop();
                }

        /**
* Rotate the vehicle by the specified angle. Positive angles   * will result in
* a clockwise rotation. Negative angles will result in a
* counterclockwise
         * rotation.
         *
         * @param pilot
         * @param angle
*           The angle (in degrees) by which to rotate the
*                vehicle
         */
public static void rotate(DifferentialPilot pilot, double angle) {

// determine the number of milliseconds to rotate based // on the vehicle
                // speed
                int travelTime = getMillisForRotate(angle);

                // for negative angles, rotate counterclockwise
                if (angle < 0)
                        pilot.rotateLeft();
                else
                        // for positive angles, rotate clockwise
                        pilot.rotateRight();

                // block the thread until the motion is complete
                sleep(travelTime);
                pilot.stop();
        }
        /**
         * Gets the milliseconds to travel for a given travel distance
         *
         * @param distance
         *            The distance in centimeters to travel
         * @return Returns the number of milliseconds to travel
         */
        public static int getMillisForTravel(double distance) {
                return (int) ((distance / FW_SPEED) * 1000);
        }

        /**
* Gets the milliseconds to rotate for the specified number of  * degrees
         *
         * @param rotateDegree
         * @return the number of milliseconds to rotate
         */
        public static int getMillisForRotate(double rotateDegree) {
                return (int) ((rotateDegree / ROTATION_SPEED) * 1000);
        }

        /**
         * Sleep function using Thread.yield rather than thread.sleep
         *
         * @param millis
*          The number of milliseconds to block the
*               executing thread
         */
        public static void sleep(long millis) {

                // create the stopwatch
                Stopwatch sw = new Stopwatch();

// continue waiting while the elapsed time is less than // the time
                // specified
                while (sw.elapsed() < millis)
                        Thread.yield();
        }
}
```

## Tracing Out an Equilateral Triangle

In this section, you will develop a program to create a robot that traces out an equilateral triangle with sides of a predetermined length set to 1 meter. You will write, compile, and upload your code into Lego Mindstorms. An example program, `example6.java`, illustrates how to trace out an equilateral triangle using `MovePilot` in leJOS EV3.

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example6.java
//an example for tracing out an equilateral triangle
//*******************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.*;

public class example6 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

        public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// setup the wheel diameter of left (and right) motor // in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the center // of wheel to
                // the center of robot, i.e. half of track width
Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

                // set up the chassis type, i.e. Differential pilot
Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                WheeledChassis.TYPE_DIFFERENTIAL);

MovePilot pilot = new MovePilot(chassis);

                // loop 3 times to trace out an equilateral triangle
                for (int sides = 0; sides < 3; sides++) {

                        // travel 100 centimeter
                        pilot.travel(100);
                // rotate 120 degrees
                pilot.rotate(120);
            }

            // press the ESCAPE button to stop moving
            while (pilot.isMoving()) {

                if (buttons.getButtons() == Keys.ID_ESCAPE)
                    pilot.stop();

            }

            // block the thread until a button is pressed
            buttons.waitForAnyPress();
        }
}
```

In case you use leJOS NXJ, an example program, `example7.java`, follows to illustrate how to trace out a triangle using a set of methods provided in the `DifferentialPilot` class.

```
//**********************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example7.java
//an example for tracing out an equilateral triangle using //DifferentialPilot in leJOS NXJ
//**********************************************************************************************

import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.nxt.Motor;
import lejos.nxt.NXTRegulatedMotor;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.util.Stopwatch;

public class example7 {

// The vehicle's rate of travel forward in centimeters per
// second
        private static final double FW_SPEED = 30.666666667f;

        // The vehicle's clockwise rotation speed in degrees per
// second
        private static final double ROTATION_SPEED = 248.297752809f;

        // The vehicle's wheel-width.
        private static final double WHEEL_DIAMETER = 3.1;

        // The vehicle's track width
        private static final double TRACK_WIDTH = 17.5;

        // Reference to left motor
        private static NXTRegulatedMotor LEFT_MOTOR = Motor.A;
        // Reference to right motor
        private static NXTRegulatedMotor RIGHT_MOTOR = Motor.C;

        /**
         * The main entry point for the program.
         */
        public static void main(String[] args) {

                // construct the pilot using the static variables
DifferentialPilot pilot = new DifferentialPilot(WHEEL_DIAMETER,
TRACK_WIDTH, LEFT_MOTOR, RIGHT_MOTOR, false);

                // tracing out a triangle
                drawTriangle(pilot, 100);

LCD.drawString("tracing is done, press button to exit.", 0, 0);
                Button.waitForPress();
        }

        /**
         * Draw an equilateral triangle
         *
         * @param pilot
         *            The pilot to use to draw the triangle
         * @param sideLengthCm
         *            The side length of the triangle
         */
        public static void drawTriangle(DifferentialPilot pilot, double sideLength) {

                // iterate through the sides of the triangle
                for (int i = 0; i < 3; i++) {

                        sleep(500);

                        // travel along a side
                        forward(pilot, sideLength);

                        sleep(500);

                        // re-orient to travel along the next side
                        rotate(pilot, 120);
                }
        }

        /**
         * Move the vehicle forward the specified distance
         *
         * @param pilot
         *            The pilot to use for moving forward
         * @param distance
         *            The distance to travel
         */
        public static void forward(DifferentialPilot pilot, double distance) {
                // get the number of milliseconds the vehicle should
// travel based on the
                // vehicle's speed
                int travelTime = getMillisForTravel(distance);

                // begin the pilot forward
                pilot.forward();

                // wait for the traveling to finish
                sleep(travelTime);

                // reached the destination - stop
                pilot.stop();
        }

        /**
* Rotate the vehicle by the specified angle. Positive angles
* will result in
* a clockwise rotation. Negative angles will result in a
* counterclockwise
         * rotation.
         *
         * @param pilot
         * @param angle
         * The angle (in degrees) by which to rotate the vehicle
         */
        public static void rotate(DifferentialPilot pilot, double angle) {

                // determine the number of milliseconds to rotate based
// on the vehicle
                // speed
                int travelTime = getMillisForRotate(angle);

                // for negative angles, rotate counterclockwise
                if (angle < 0)
                        pilot.rotateLeft();
                else
                        // for positive angles, rotate clockwise
                        pilot.rotateRight();

                // block the thread until the motion is complete
                sleep(travelTime);
                pilot.stop();
        }

        /**
         * Gets the milliseconds to travel for a given travel distance
         *
         * @param distance
         *            The distance in centimeters to travel
         * @return Returns the number of milliseconds to travel
         */
        public static int getMillisForTravel(double distance) {
                return (int) ((distance / FW_SPEED) * 1000);
        }

        /**
         * Gets the milliseconds to rotate for the specified number of
 * degrees
         *
         * @param rotateDegree
         * @return the number of milliseconds to rotate
         */
        public static int getMillisForRotate(double rotateDegree) {
                return (int) ((rotateDegree / ROTATION_SPEED) * 1000);
        }

        /**
         * Sleep function using Thread.yield rather than thread.sleep
         *
         * @param millis
 * The number of milliseconds to block the executing thread
         */
        public static void sleep(long millis) {

                // create the stopwatch
                Stopwatch sw = new Stopwatch();

// continue waiting while the elapsed time is less than // the time
                // specified
                while (sw.elapsed() < millis)
                        Thread.yield();
        }
}
```

## Tracing Out a Regular Hexagon

In this section, you will develop a program to create a robot that traces out a regular hexagon with sides of a predetermined length set to 50 centimeters. You will write, compile, and upload your code into Lego Mindstorms. An example program, `example8.java`, illustrates how to trace out a regular hexagon using `MovePilot` in leJOS EV3.

```
//*************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example8.java
//an example for tracing out a regular hexagon
//*************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.*;

public class example8 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys // class for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// setup the wheel diameter of left (and right) // motor in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the // center of wheel to
                // the center of robot, i.e. half of track width
Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

                // set up the chassis type, i.e. Differential
// pilot
Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                WheeledChassis.TYPE_DIFFERENTIAL);

MovePilot pilot = new MovePilot(chassis);

                // loop 6 times to trace out a regular hexagon
                for (int sides = 0; sides < 6; sides++) {

                        // travel 100 centimeter
                        pilot.travel(50);

                        // rotate 60 degrees
                        pilot.rotate(60);
                }

                // press the ESCAPE button to stop moving
                while (pilot.isMoving()) {

if (buttons.getButtons() == Keys.ID_ESCAPE)
                                pilot.stop();

                }
                // block the thread until a button is pressed
                buttons.waitForAnyPress();
        }
}
```

In case you use leJOS NXJ, an example program, `example9.java`, is provided here to illustrate how to trace out a hexagon using a set of methods provided in the `DifferentialPilot` class.

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 example9.java
//an example for tracing out a regular hexagon using //DifferentialPilot in leJOS NXJ
//*******************************************************************

import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.nxt.Motor;
import lejos.nxt.NXTRegulatedMotor;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.util.Stopwatch;

public class example9 {

        // The vehicle's rate of travel forward in centimeters per
// second
        private static final double FW_SPEED = 30.666666667f;

        // The vehicle's clockwise rotation speed in degrees per
// second
        private static final double ROTATION_SPEED = 248.297752809f;

        // The vehicle's wheel-width.
        private static final double WHEEL_DIAMETER = 3.1;

        // The vehicle's wheel-bas
        private static final double TRACK_WIDTH = 17.5;

        // Reference to left motor
        private static NXTRegulatedMotor LEFT_MOTOR = Motor.A;

        // Reference to right motor
        private static NXTRegulatedMotor RIGHT_MOTOR = Motor.C;

        /**
         * The main entry point for the program.
         */
        public static void main(String[] args) {

                // construct the pilot using the static variables
DifferentialPilot pilot = new DifferentialPilot(WHEEL_DIAMETER,
TRACK_WIDTH, LEFT_MOTOR, RIGHT_MOTOR, false);
```

```
                // tracing out a triangle
                drawHexagon(pilot, 50);

LCD.drawString("tracing is done, press button to exit.", 0, 0);
                Button.waitForPress();
        }

        /**
         * Draw a hexagon with sides of the specified length
         *
         * @param pilot
         *            The pilot to use for drawing the hexagn
         * @param sideLengthCm
         *            The length of each side of the regular hexagon
         */
public static void drawHexagon(DifferentialPilot pilot, double sideLength) {

                // iterate through the sides of the hexagon
                for (int i = 0; i < 6; i++) {

                        sleep(500);

                        // move the vehicle along an edge
                        forward(pilot, sideLength);

                        sleep(500);

                        // rotate the bot to traverse the next leg
                        rotate(pilot, 60);
                }
        }

        /**
         * Move the vehicle forward the specified distance
         *
         * @param pilot
         *            The pilot to use for moving forward
         * @param distance
         *            The distance to travel
         */
        public static void forward(DifferentialPilot pilot, double distance) {
                // get the number of milliseconds the vehicle should
// travel based on the
                // vehicle's speed
                int travelTime = getMillisForTravel(distance);

                // begin the pilot forward
                pilot.forward();

                // wait for the traveling to finish
                sleep(travelTime);
                // reached the destination - stop
                pilot.stop();
        }

        /**
 * Rotate the vehicle by the specified angle. Positive angles
 * will result in
         * a clockwise rotation. Negative angles will result in a
 * counterclockwise
         * rotation.
         *
         * @param pilot
         * @param angle
         * The angle (in degrees) by which to rotate the vehicle
         */
        public static void rotate(DifferentialPilot pilot, double angle) {

// determine the number of milliseconds to rotate based // on the vehicle
                // speed
                int travelTime = getMillisForRotate(angle);

                // for negative angles, rotate counterclockwise
                if (angle < 0)
                        pilot.rotateLeft();
                else
                        // for positive angles, rotate clockwise
                        pilot.rotateRight();

                // block the thread until the motion is complete
                sleep(travelTime);
                pilot.stop();
        }

        /**
         * Gets the milliseconds to travel for a given travel distance
         *
         * @param distance
         *            The distance in centimeters to travel
         * @return Returns the number of milliseconds to travel
         */
        public static int getMillisForTravel(double distance) {
                return (int) ((distance / FW_SPEED) * 1000);
        }

        /**
         * Gets the milliseconds to rotate for the specified number of
 * degrees
         *
         * @param rotateDegree
         * @return the number of milliseconds to rotate
         */
        public static int getMillisForRotate(double rotateDegree) {
                return (int) ((rotateDegree / ROTATION_SPEED) * 1000);
        }

        /**
         * Sleep function using Thread.yield rather than thread.sleep
         *
         * @param millis
         * The number of milliseconds to block the
         * executing thread
         */
        public static void sleep(long millis) {

                // create the stopwatch
                Stopwatch sw = new Stopwatch();

// continue waiting while the elapsed time is less than // the time
                // specified
                while (sw.elapsed() < millis)
                        Thread.yield();
        }
}
```

## Summary

In this chapter, you learned about the various methods provided in the `MovePilot` class of leJOS EV3 and the `DifferentialPilot` class of leJOS NXJ. Based on those methods, you now know how to control the movement of wheeled vehicles precisely. In particular, the nine example Java leJOS projects illustrate in detail how to trace out a predefined geometric shape with sides of a predefined length using the pilot class with both Lego Mindstorms EV3 and Lego Mindstorms NXT.

In the next chapter, you will learn the basics of the Cartesian coordinate system used in leJOS EV3. Then, based on that, you will study how to apply programming methods in the Navigation class to control the wheeled vehicle in order to trace out a predefined path with coordinates in a two-dimensional plane. Moreover, the major hardware components of the Lego EV3 brick will be introduced.

CHAPTER 4

# Coordinators and Navigator API

This chapter introduces you the basics of the Cartesian coordinate system used in leJOS EV3. It also teaches you how to apply programming methods in the Navigation class to control the wheeled vehicle in order to trace out a predefined path with coordinates in a two-dimensional plane. Moreover, you will learn about the major hardware components of the Lego EV3 brick, such as the LCD display and the keys for interacting with the robot. You will also learn how to apply the methods used to control the LCD display and buttons used to provide input to and obtain output from the robot.

In particular, this chapter includes seven example Java projects and covers the following topics:

- Introduction to the Cartesian coordinate system
- Basics of the navigator API functions
- Controlling the EV3 brick hardware
- Programming practice with buttons and the LCD display
- Programming practice on tracing a two-dimensional plane

## Cartesian Coordinate System Basics

From our perspective, we can easily describe a location in words such as "I am located at the corner of 10th Avenue and 2nd Street," or "I am located at 100 Main Street, Keene, New Hampshire." However, such descriptions don't mean anything to Lego robots, as they have no concept of semantics. Instead, the Lego robots only understand numbers.

Thus when programming Lego robots, a Cartesian coordinate system is used to describe locations. As illustrated in Figure 4-1, a two-dimensional Cartesian coordinate system keeps track of two numbers: the values of X and Y. Numbers grow larger and smaller along the X-axis and Y-axis. Both axes start at 0 and include positive and negative numbers. The axes X and Y divide the Cartesian coordinate system into four quadrants, namely, areas I, II, III, and IV. Any point in a two-dimensional area can be plotted on this grid using values of X and Y.

W. Lu, *Beginning Robotics Programming in Java with LEGO Mindstorms*, DOI 10.1007/978-1-4842-2005-4_4

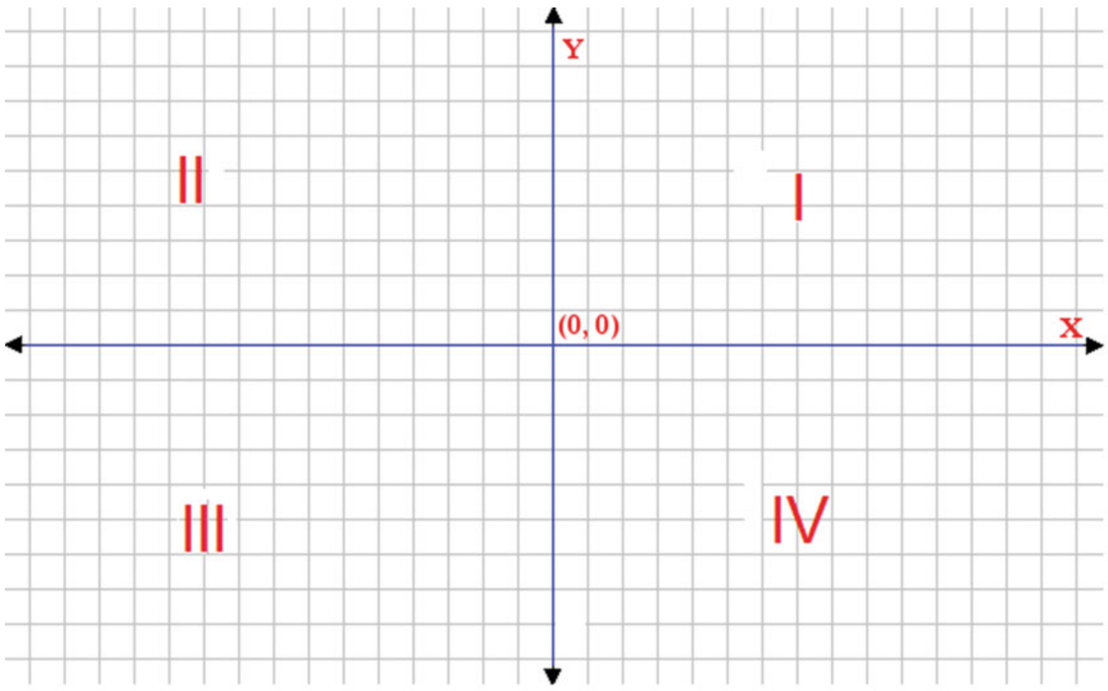


***Figure 4-1.*** *A Cartesian coordinate system*

Moreover, in a Cartesian coordinate system, rotations to the left side are designated as positive rotations. This means that if you rotate positive 90 degrees (+90), you have rotated counterclockwise. Similarly, a rotation of negative 90 degrees(-90) is equivalent to rotating in a clockwise direction.

## Navigator API Functions

In Chapter 3, you learned about pilots, such as `MovePilot` and `DifferentialPilot`, and how pilots allow a robot to perform precise moves and drive specific distances. Using the pilot, you will learn about another class called the *Navigator*, which tells the pilot how to drive to a specific location based on the Cartesian coordinate system.

The Navigator takes a pilot object in its constructor and then calculates a series of movements from one location to another. The Navigator actually knows nothing about how the robot works or how the pilot object works out moving around. Instead, the Navigator simply asks the pilot to execute instructions of movements.

The following code statement instantiates a Navigator and then drives to the target coordinate x = 50, y = 50, in which we will assume a pilot exists:

```
Navigator navtest = new Navigator(pilot);
navtest.goTo(50,50);
```

As you learned in Chapter 3, you can build a pilot object using the following program:

```
// setup the wheel diameter of left (and right) motor in
// centimeters,
        // i.e. 2.8 cm
        // the offset number is the distance between the center of
// wheel to
        // the center of robot, i.e. half of track width
Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

        // set up the chassis type, i.e. Differential pilot
Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 }, WheeledChassis.TYPE_
DIFFERENTIAL);
        MovePilot pilot = new MovePilot(chassis);
```

In a navigation class, the target coordinates are also known waypoints. You can create your own method to generate a set of waypoints and then feed the coordinates to the Navigator using waypoint objects. The following shows you how to use the waypoint:

```
Waypoint wp = new Waypoint (50, 50);
navtest.goTo(wp);
```

In the case of finding a path, once a path is found, you can then put the waypoints into a navigation queue using the `addWaypoint()` method. For example,

```
navtest.addWaypoint(new Waypoint(200,200));
```

The Navigator drives a pilot object to make the move control. This is achieved by a `PoseProvider` that keeps track of the sequence of robot positions in a route. The route is an instance of the path class, and it behaves as a first-in, first-out (FIFO) queue. When a waypoint is reached, it is removed from the route queue and the robot goes to the next waypoint. At any time, new waypoints can be added to the end of the route queue.

The following is a list of methods in the Navigation class. All of the methods provided by the Navigation class are non-blocking, which means that the methods return immediately once they complete their function.

`void setPath(Path path)` Sets the path that the Navigator traverses.

`void addWaypoint(Waypoint waypoint)` Adds a waypoint to the end of the path.

`void addWaypoint(float x, float y )` Constructs a new waypoint having the coordinate (x,y) and then adds it to the end of the path.

`void addWaypoint(float x, float y , heading )` Creates a new waypoint having the coordinate (x,y) and then adds it to the end of the path. You can use this method to specify the heading of the robot when it reaches to the waypoint.

`void followPath()` Starts the robot traversing along an existing/current route.

`void followPath(Path path )` Starts the robot traversing the path to be followed.

`void goTo(Waypoint destination)` Starts the robot moving toward the destination. If no path exists, a new one is created consisting of the destination; otherwise, the destination is added to the existing path.

`void goTo(double x, double y)` Starts the robot moving toward the destination specified by the coordinate (x,y). If no path exists, a new one is created consisting of the destination; otherwise, the new waypoint is added to the existing path.

`void goTo(double x, double y, double heading)` Starts the robot moving toward the destination specified by the coordinate (x,y). If no path exists, a new one is created consisting of the destination; otherwise, the new waypoint is added to the existing path. You can use this method to specify the heading of the robot when it reaches the destination waypoint.

`leJOSvoid stop()` Stops the robot but preserves the route so that you can resume its path traversal if you call `followPath()`.

`boolean isMoving()` Returns true if the robot is moving toward a waypoint.

`boolean pathCompleted()` Returns true if the robot has reached the final waypoint.

`void rotateTo(double direction)` Rotates the robot to a new directional angle in the Cartesian plane. When the x-axis is 0, for example, `rotateTo(0)` will align the robot with the x-axis. If the y-axis is 90 degrees, `rotateTo(90)` aligns it with the y-axis. The numerical value of direction is the absolute heading to which you can rotate the robot, and it ranges from 0 to 360.

`void waitForStop()` Returns true if the robot stopped at the final waypoint of the path and is no longer moving.

`void singleStep(boolean yes)` Controls whether the robot stops at each waypoint. If yes, you can call this method with a true parameter and then the robot stops at each waypoint.

Here is a simple example, `ch4p1.java`, to move your robot from starting point (0, 0) to the location with coordinate (50, 50). The units of measurement here are in centimeters because the diameter and track width employed by pilot uses centimeters(for example, 2.8 centimeters and 18 centimeters).

```
//****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch4p1.java
//an example for navigation testing
//****************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.*;

public class ch4p1 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);

static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);
public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// setup the wheel diameter of left (and right) motor // in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the center // of wheel to
                // the center of robot, i.e. half of track width
Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

                // set up the chassis type, i.e. Differential pilot
Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                WheeledChassis.TYPE_DIFFERENTIAL);
                MovePilot pilot = new MovePilot(chassis);

                Navigator navtest = new Navigator(pilot);

                // define a new waypoint as destination
                Waypoint wp = new Waypoint (50, 50);

                // robot moves to the destination waypoint
                navtest.goTo(wp);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();
        }
}
```

## Controlling the EV3 Brick Hardware

The EV3 brick is equipped with 6 AA batteries, or a rechargeable battery pack that is slightly larger. This durable brick also contains a 32-bit ARM9 processor running at 300MHz. This processor has direct access to 64MB of RAM and 16MB of flash memory. To conserve battery life, the flash RAM stores data and programs even when there is no power input. Compared to modern computers with 4GB of RAM, the RAM used by the Lego EV3 is very small. However, this is actually plenty for your robotics projects because modern computers consume a lot of their RAM capacity for running their operating systems, generating graphics, and performing other memory-intensive tasks. The robotics programs, however, don't employ heavy-duty graphics or perform processor-intensive computing tasks.

In this section, you will familiarize yourself with the EV3 brick including the buttons used for input, the liquid crystal display (LCD), and the small speaker used for output. leJOS EV3 provides API (Application Program Interface) functions used for controlling all of the hardware.

The LCD class in leJOS EV3 can be used in text mode or graphics mode. For text display, the EV3 LCD screen is 16 characters wide by 8 characters deep. It is addressed by using the coordinates (x, y), as shown in Figure 4-2, in which x ranges from 0 to 15, and y ranges from 0 to 7.

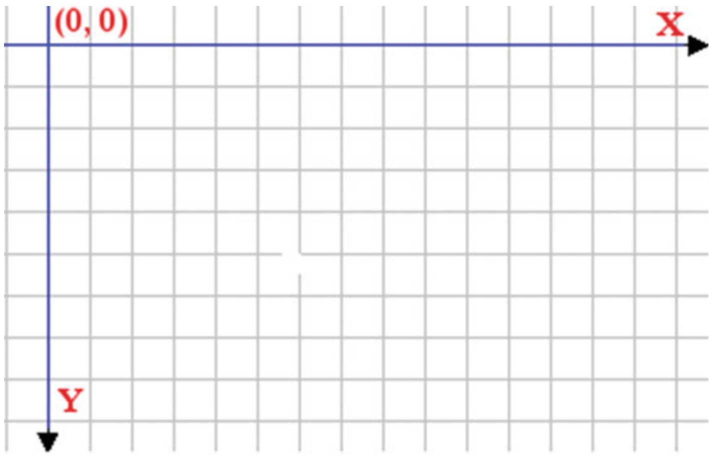


***Figure 4-2.*** *Coordinate system of LCD display*

The methods used to write to the LCD screen in text mode follow:

> `void drawString(String str, int x, int y)` Displays a string of text on the LCD screen starting at coordinate (x,y).
>
> `void drawInt(int i, int x, int y)` Displays an integer starting at coordinate (x,y). The integer is left-aligned and takes up as many characters as are necessary.
>
> `void drawInt(int i, int places, int x, int y)` Displays an integer starting at coordinate (x,y) with leading spaces to occupy at least the number of characters specified by the places parameter. This means that the method always writes to a fixed number of characters specified in places and the previous value will always be fully overwritten. For example, if the value of places is set to 5, that means there are 5 characters to be overwritten when displaying an integer number on the LCD.
>
> `void clear()` Clears the display.

As an example, the program `ch2p2.java` illustrates how to display the free memory space on the LCD of EV3 brick:

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch4p2.java
//An example to test LCD displaying methods
//*****************************************************************

// import EV3 hardware packages for EV brick finding, activating
// keys and LCD
import lejos.hardware.ev3.EV3;
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.lcd.TextLCD;

public class ch4p2 {

        public static void main(String[] args) throws InterruptedException {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantized LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

                // drawing text on the LCD screen based on coordinates
                lcddisplay.drawString("Free RAM:", 0, 0);

                // displaying free memory on the LCD screen
lcddisplay.drawInt((int) Runtime.getRuntime().freeMemory(), 0, 1);

                // exit program after any button pressed
                buttons.waitForAnyPress();
        }

}
```

You can also write to the LCD display with `System.out.println(String str)`. For example, `System.out.println("hello")` will display the string `hello` on the screen and overwrite the existing characters. By default, the LCD display is refreshed automatically. If you want to control when the LCD is refreshed, you can call `lcddisplay.setAutoRefresh(false)` to turn off auto-refreshing and call `lcddisplay.refresh()` when you want to refresh the display.

All six control keys on the EV3 brick are reprogrammable under leJOS EV3. You can use events to listen for key presses and react accordingly when one is activated. The `Keys` class contains static instances of the six keys. These six instances are `ENTER`, `ESCAPE`, `LEFT`, `RIGHT`, `UP`, and `DOWN`.

Usually, when programming, you want to wait or block the process until a key is pressed. Thus the simplest way to achieve this is to use the `waitForAnyPress()` method. For example, to stop code until `ESCAPE` key is pressed, you can do the following:

```
buttons.getButtons()==Keys.ID_ESCAPE
```

You can also use a simple while loop to stop your code while it waits for the user to press an `ESCAPE` button:

```
        while(buttons.getButtons() != Keys.ID_ESCAPE) { }
```

The following program, `ch4p3.java`, tests if a button is pressed:

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch4p3.java
//an example for button testing
//*****************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;

public class ch4p3 {
        public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();
                // Keys class for buttons
                Keys buttons = ev3brick.getKeys();

                // press the ESCAPE key to exit the program
                while (buttons.getButtons() != Keys.ID_ESCAPE) {
                        // clearing the LCD screen at first
                        LCD.clear();
                        // press the ENTER key so the ENTER will be
                        // displayed on the LCD screen
                        if (buttons.getButtons() == Keys.ID_ENTER) {
                                // displaying ENTER on the LCD screen
                                LCD.drawString("ENTER", 0, 0);
                                // leave the string ENTER on the screen
                                // for 2 seconds
                                Thread.sleep(2000);
                        }

                }
        }
}
```

Other methods used for controlling keys include the following:

```
static int waitForAnyPress()
```

To wait for any key to be pressed, the above command returns the id code of the key that is pressed as illustrated here:

| **button** | **UP** | **ENTER** | **DOWN** | **RIGHT** | **LEFT** | **ESCAPE** |
|---|---|---|---|---|---|---|
| **code** | 1 | 2 | 4 | 8 | 16 | 32 |

`static int readButtons()` Reads the current state of all of the keys. The return value is the sum of all of the codes of the keys that are pressed.

As illustrated in the above program, `ch4p3.java`, you will find that `Thread.sleep(2000)` is used to add in various delays so that the robot can take breaks to do other things. This happens in many cases when using sensors (for example, a light sensor or color sensor) to record rapid events that require extreme accuracy. Such events could be waiting for a color sensor to detect a black line or an ultrasonic senor to detect a wall. Therefore, the method `Thread.sleep()` can be useful for timing different actions or timestamping events for later analysis.

## Programming Practice with the LCD Display

In this practice session, you will develop a program that shows "`Here is my RAM`" in the LCD screen for five seconds and then print out "`I got it.`" An example program, `ch4p4.java`, illustrates how to achieve this goal.

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch4p4.java
//A programming practice example to display free RAM on LCD
//*****************************************************************

// import EV3 hardware packages for EV brick finding, activating
// keys and LCD
import lejos.hardware.ev3.EV3;
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.lcd.TextLCD;
import lejos.utility.Stopwatch;

public class ch4p4 {

public static void main(String[] args) throws InterruptedException {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

                // LCD class for displaying and Keys class for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

                // for timing dialogs
                Stopwatch sw = new Stopwatch();

                // drawing free RAM on the LCD screen based on
                // coordinates
                lcddisplay.drawString("Here is my RAM:", 0, 0);

                // displaying free memory on the LCD screen
                lcddisplay.drawInt((int) Runtime.getRuntime().freeMemory(), 0, 1);
                sw.reset();

                // wait for 5 seconds, then display a message
                while (sw.elapsed() < 5000)
                        Thread.yield();
                sw.reset();
                lcddisplay.drawString("I got it", 0, 2);
                // exit program after any button pressed
                buttons.waitForAnyPress();
        }
}
```

## Programming Practice with Key Presses

In this programming practice session, you will develop two programs to press keys and display corresponding messages on the LCD. In the first program, you need to display the button being pressed on the LCD. Its pseudocode is illustrated in the following example:

```
while(ESCAPSE is not pressed){
        Clear LCD
        If ENTER is pressed
                Display “ENTER” on first row
                Else if LEFT is pressed
                Display “LEFT” on the first row
                else if RIGHT is pressed
                Display “RIGHT” on the first row
                else if UP is pressed
                Display “UP” on the first row
                else if DOWN is pressed
                Display “DOWN” on the first row
}
```

An example program, ch4p5.java, shows you how to complete this practice session.

```
//*************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch4p5.java
//A programming practice example to display various buttons when //they are pressed
//*************************************************************************************

// import EV3 hardware packages for EV brick finding, activating
// keys and LCD
import lejos.hardware.ev3.EV3;
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.lcd.TextLCD;

public class ch4p5 {

        public static void main(String[] args) throws InterruptedException {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

                // LCD class for displaying and Keys class for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();
// continue waiting for button pressed while the user // has not pressed
                // escape button
                while (buttons.getButtons() != Keys.ID_ESCAPE) {
                        // display a message for enter
                        if (buttons.getButtons() == Keys.ID_ENTER) {
                                lcddisplay.clear();
                                lcddisplay.drawString("ENTER", 0, 0);
                        }
                        // display a message for left button
                        else if (buttons.getButtons() == Keys.ID_LEFT) {
                                lcddisplay.clear();
                                lcddisplay.drawString("LEFT", 0, 0);
                        }
                        // display a message for right button
                        else if (buttons.getButtons() == Keys.ID_RIGHT) {
                                lcddisplay.clear();
                                lcddisplay.drawString("RIGHT", 0, 0);
                        }
                        // display a message for up button
                        else if (buttons.getButtons() == Keys.ID_UP) {
                                lcddisplay.clear();
                                lcddisplay.drawString("UP", 0, 0);
                        }
                        // display a message for down button
                        else if (buttons.getButtons() == Keys.ID_DOWN) {
                                lcddisplay.clear();
                                lcddisplay.drawString("DOWN", 0, 0);
                        }
                }
        }
}
```

In the second practice, you need to write a program that waits for ENTER to be pressed before carrying out the rest of the program. The pseudocode is illustrated in the following example:

```
Clear LCD
Display "Press ENTER to continue" then Refresh LCD
Wait for the ESCAPE to be pressed and released
Display a message that the ESCAPE is indeed pressed and released
```

An example program, `ch4p6.java`, shows you how to complete this practice session.

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch4p6.java
//A programming practice example to display ESCAPE button pressed
// and released
//*******************************************************************

// import EV3 hardware packages for EV brick finding, activating
// keys and LCD
import lejos.hardware.ev3.EV3;
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.lcd.TextLCD;
import lejos.utility.Stopwatch;

public class ch4p6 {

public static void main(String[] args) throws InterruptedException {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

                // LCD class for displaying and Keys class for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

                // print instructions for the user
                lcddisplay.clear();
                lcddisplay.drawString("Prs ENTER to cont", 0, 0);
                lcddisplay.refresh();

                // wait until ENTER key is pressed
                while (buttons.waitForAnyPress() != Keys.ID_ENTER)
                        Thread.yield();

// wait until ESCAPE key is pressed, then wait for it // to be released
                while (buttons.getButtons() != Keys.ID_ESCAPE)
                        Thread.yield();

                // once escape is released, indicate that it was and
// pause for a moment
                // (i.e. 5 seconds) before exiting
                lcddisplay.drawString("Escape was pressed", 0, 1);
                Stopwatch sw = new Stopwatch();
                while (sw.elapsed() < 5000)
                        Thread.yield();
        }
}
```

## Programming Practice with Navigator API

The leJOS Navigator interface provides a set of methods that you can use to control the movements of a robot precisely, including the ones for moving to any location and for controlling the direction of the movement. In this practice session, you are going to write a program in which the robot will travel in a two-dimensional plane, as shown in Figure 4-3:

```
starting at A(0, 0),
go to B(50, 50),
then, go back A(0, 0)
then go to C(-50, 50),
then finally come back to A(0, 0).
```

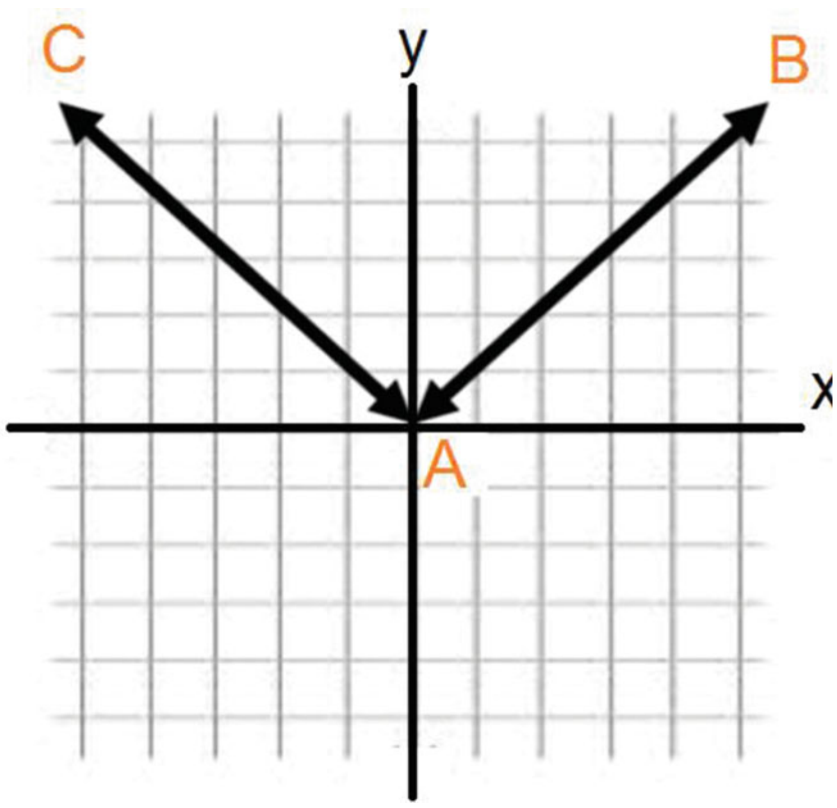


***Figure 4-3.*** *A two-dimensional plane for testing navigator API*

---

■ **Note** The units of the coordinates are measured in centimeters in this practice session. Thus, you must make sure that your track width and diameter are set to centimeters.

---

The pseudocode for this program follows:

```
DifferentialPilot ev3robot = new DifferentialPilot(diam,trackwidth,Motor.A,Motor.C);

NavPathController navbot = new NavPathController(ev3robot);

Button waitForPress to start

Display destination’s coordinate (50, 50) on LCD

Display the message "Press ENTER key" to continue; Go to location with coordinate 50,50

Display destination’s coordinate (0, 0) on LCD

Display the message "Press ENTER key" to continue; Go to location with coordinate 0,0
Display destination's coordinate (-50, 50) on LCD Display the message "Press ENTER key" to
continue; Go to location with coordinate -50,50

Display destination's coordinate (0, 0) on LCD Display the message "Press ENTER key" to
continue; Go to location with coordinate 0,0

Button waitForPress to exit
```

An example program, ch4p7.java, shows you below how to complete this practice.

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch4p7.java
//A programming example to practice navigator API functions
//*****************************************************************

// import EV3 hardware packages for EV brick finding, activating
// keys and LCD
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.TextLCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.MovePilot;
import lejos.robotics.navigation.Navigator;
import lejos.robotics.navigation.Waypoint;
import lejos.utility.TextMenu;

public class ch4p7 {

static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.A);
static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(MotorPort.C);

public static void main(String[] args) throws InterruptedException {

                // menu options for displaying navigator controlling
                String[] menuItems = new String[1];
                menuItems[0] = "Task 1: Navigator";

                // display menu
                TextMenu menu = new TextMenu(menuItems);
                menu.setTitle("NavRobot");

// setup the wheel diameter of left (and right) motor // in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the center // of wheel to
                // the center of robot, i.e. half of track width
Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);
```

```
                // set up the chassis type, i.e. Differential pilot
Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                WheeledChassis.TYPE_DIFFERENTIAL);
                MovePilot ev3robot = new MovePilot(chassis);

                Navigator navbot = new Navigator(ev3robot);

                // run routine based on menu choice
                switch (menu.select()) {
                case 0:
                        navigate(navbot);
                        break;
                }
        }

        // navigate()
        // demo navigating using way points

        private static void navigate(Navigator nav) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

                // set up way points
                Waypoint wp1 = new Waypoint(50, 50);
                Waypoint wp2 = new Waypoint(0, 0);
                Waypoint wp3 = new Waypoint(-50, 50);

                // clear menu
                lcddisplay.clear();

                // display current location
                lcddisplay.drawString("At 0, 0", 0, 0);
                lcddisplay.drawString("Press ENTER", 0, 2);
                lcddisplay.drawString("to continue", 0, 3);

                // wait until ENTER key is pressed
                while (buttons.waitForAnyPress() != Keys.ID_ENTER)
                        Thread.yield();

                // navigate to way point one, display new location
                lcddisplay.clear();
                nav.goTo(wp1);
                lcddisplay.drawString("At 50, 50", 0, 0);
                lcddisplay.drawString("Press ENTER", 0, 2);
                lcddisplay.drawString("to continue", 0, 3);
                // wait until ENTER key is pressed
                while (buttons.waitForAnyPress() != Keys.ID_ENTER)
                        Thread.yield();

                // navigate to way point two, display new location
                lcddisplay.clear();
                nav.goTo(wp2);
                lcddisplay.drawString("At 0, 0", 0, 0);
                lcddisplay.drawString("Press ENTER", 0, 2);
                lcddisplay.drawString("to continue", 0, 3);

                // wait until ENTER key is pressed
                while (buttons.waitForAnyPress() != Keys.ID_ENTER)
                        Thread.yield();

                // navigate to way point 3, display new location
                lcddisplay.clear();
                nav.goTo(wp3);
                lcddisplay.drawString("At -50, 50", 0, 0);
                lcddisplay.drawString("Press ENTER", 0, 2);
                lcddisplay.drawString("to continue", 0, 3);

                // wait until ENTER key is pressed
                while (buttons.waitForAnyPress() != Keys.ID_ENTER)
                        Thread.yield();

                // navigate to way point 2, display new location
                lcddisplay.clear();
                nav.goTo(wp2);
                lcddisplay.drawString("At 0, 0", 0, 0);
                lcddisplay.drawString("Press ENTER", 0, 2);
                lcddisplay.drawString("to exit", 0, 3);

                // wait until ENTER key is pressed
                while (buttons.waitForAnyPress() != Keys.ID_ENTER)
                        Thread.yield();
        }
}
```

As you have seen in the Cartesian coordinate system illustrated in Figure 4-3, the coordinate at node A is (0,0), the coordinate at node B is (50,50), and the coordinate at node C is (-50,50). The path of the robot starts at node A, traverses to node B, then backtracks to node A, traverses to node C, and finally backtracks to node A. Nodes A, B, and C actually form small elements in the binary tree, and this practices shows the simplest form of traversing a binary tree.

## Summary

In this chapter, you learned about the Cartesian coordinate system used in leJOS EV3. You also studied how to apply various methods in the Navigation class to control the wheeled vehicles so they can trace out a predefined path with coordinates in a two-dimensional plane. Moreover, you learned about the major hardware components of EV3 brick: that is, the LCD display and the six keys on the EV3 brick for interacting with your robot. Finally, you studied applying built-in system methods for controlling the LCD display and the buttons in order to provide input to and/or obtain output from your robot.

In the next chapter you will move onto the next component of the book, that is, building intelligence for your robot by using search strategies. In particular, will learn how to implement a depth-first search (DFS) algorithm on the tiny Java Virtual Machine (JVM), that is, LeJOS EV3, and then integrating the DFS algorithm in the LeJOS-based robotics system for localization and path planning, enhancing the existing pathfinding approaches within the LeJOS system.

CHAPTER 5

# Depth-First Search Algorithm and Its Implementation with Lego EV3

The act of searching falls under the *artificial intelligence (AI)* umbrella. The major goal of *AI* is to give computers the ability to think: in other words, to mimic human behavior. The problem with this mimicry is that, unfortunately, a computer's "brain" doesn't function in the same manner as a human brain: computers require a series of well-reasoned steps to process in order to find a solution. Thus, your goal is to deconstruct a complicated task and convert it into simpler steps that the robotics system can handle. This conversion, from something complex to something simple, is what the search algorithm will do.

In this chapter, you will learn how to implement a *depth-first search (DFS)* algorithm on the tiny *Java Virtual Machine (JVM)*: that is, of course, leJOS EV3. As you know, leJOS EV3 is an open source project created to develop a technological infrastructure in which Java technology is applied to programming software for robots. Java is an object-oriented programming language. One of the most important features implemented in leJOS is the leJOS navigation API (discussed in Chapter 4), which can be used to achieve the goal in which a convenient set of classes and methods are provided to control the robot.

The classes that control vehicles deal with several levels of abstraction. At the very bottom, there are the motors that turn the wheels, which are controlled by the `RegulatedMotor` class. The `MovePilot` (or `DifferentialPilot` in leJOS NXJ) class uses the motors to control elementary moves: rotate in place, travel in a straight line, or travel in an arc. At the next level up, the `NavPathController` uses a `DifferentialPilot` to move the robot through a complicated path on a plane. To perform navigation, the path controller needs the robot location and the direction in which it is heading. It uses an `OdometeryPoseProvider` to keep this information up to date. In particular, this chapter will cover the following topics:

- A new depth-first search (DFS) algorithm that can be applied to building arbitrary tree structures generically.
- Applying and integrating the proposed DFS algorithm in the leJOS-based robotics system for localization and path planning, enhancing the existing pathfinding approaches within leJOS system in the process.

## Overview of DFS Algorithm

Let's first examine how humans solve search problems. First, a representation of how the search problem exists is required. Figure 5-1 is an example of a search tree. It is a series of interconnected nodes that through which we will search:

W. Lu, *Beginning Robotics Programming in Java with LEGO Mindstorms*, DOI 10.1007/978-1-4842-2005-4_5

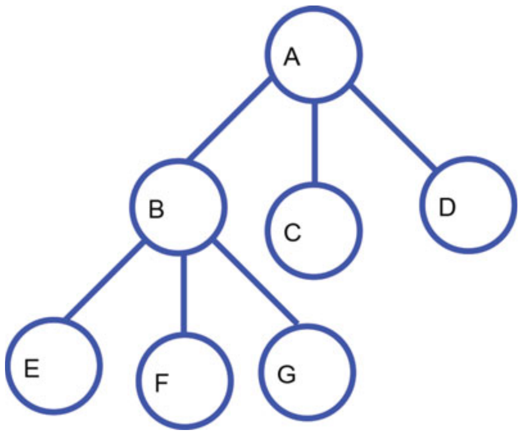


***Figure 5-1.*** *Tree structure of a path*

Depth-first search (DFS) works by taking a node; checking its neighbors; expanding the first node it finds among those neighbors; checking to see if that expanded node is your destination; and if not, continue to explore more nodes. For example, if you want to find a path from A to E, you can use two lists to keep track of what you are doing: an *open list* and a *closed list*. An open list keeps track of what you need to do, and a closed list keeps track of what you have already done.

At the beginning, you only have your starting point, node A. You haven't done anything to it yet, so let's add it to your *open list*. Then you have an open list including `<A>` and a closed list including `<empty>`. Now let's explore the neighbors of your A node. Node A's neighbors are the B, C, and D nodes. Because you are now done with your A node, you can remove it from your open list and add it to your closed list. Your current open list then includes `<B, C, D>` and the closed list contains `<A>`. Now our *open list* contains three items.

For depth-first search, you always explore the first item from your open list. The first item in your open list is the B node. B is not your destination, so let's explore its neighbors. Because you have now expanded B, you are going to remove it from the open list and add it to the closed list. Your new nodes are E, F, and G, and you add these nodes to the beginning of your open list. Then you have an open list including `<A, B>`, and a closed list including `<E, F, G, C, D>`. Now expand the E node. Since it is your intended destination, you should stop. Therefore, you receive the route `A->B->E` that is interpreted from the closed list by using the regular depth-first search algorithm.

Next let's see how to use the depth-search approach to solve a path location problem that could be applied to a GPS system when navigating from a starting city to a destination city. Suppose that you want to drive from city *A* (Keene, NH, for example) to city *S* (let's say Boston, MA). Given the following route path, determine a plan for you to start from *A* and end at *S* using a depth-first search strategy.

| City | Distance |
|---|---|
| A to B | 20 miles |
| A to C | 10 miles |
| A to D | 10 miles |
| A to E | 20 miles |
| B to F | 10 miles |
| B to M | 20 miles |
| B to G | 10 miles |

(*continued*)

| City | Distance |
|---|---|
| C to H | 10 miles |
| C to I | 15 miles |
| D to J | 20 miles |
| E to K | 15 miles |
| E to L | 15 miles |
| M to N | 20 miles |
| M to O | 20 miles |
| I to P | 40 miles |
| P to R | 20 miles |
| P to S | 20 miles |

Based on the above information, you can plot a route tree, as illustrated in Figure 5-2, displaying all of the possible routes in between the two cities:

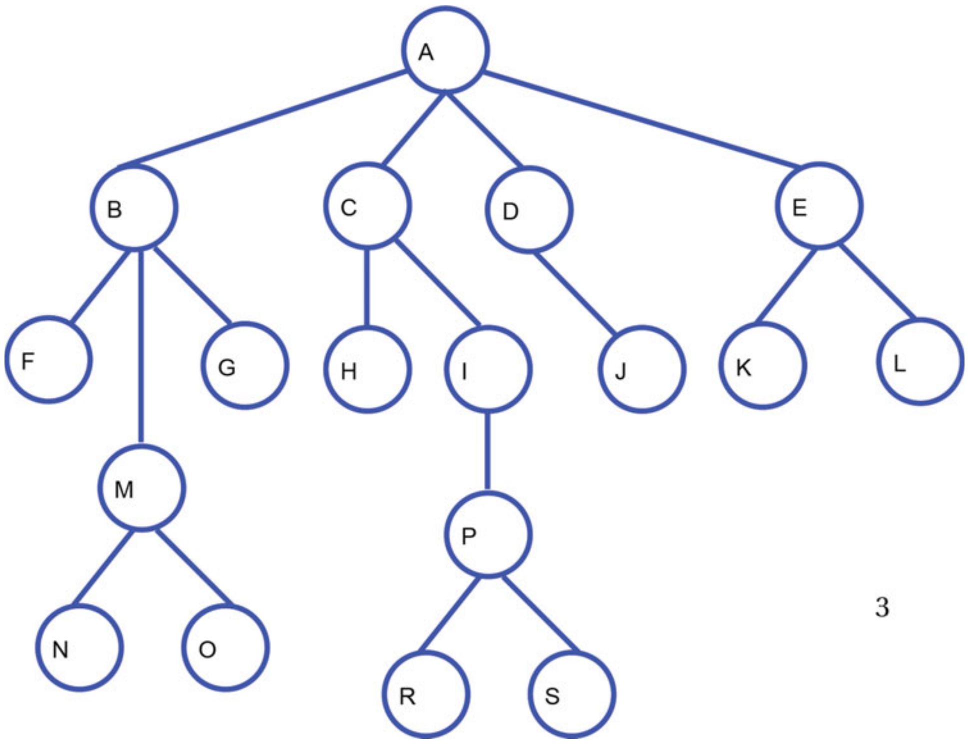


***Figure 5-2.*** *Routes in between two cities*

The following program is an example code that can be used to find a path for you to schedule a travel plan automatically by using the depth-first search algorithm:

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch5p1_main.java
//Driver class to set up map using ch5p1_GraphNode, ch5p1_Link, and //ch5p1_Graph classes.
//Calls a depth-first search in ch5p1_Graph class to create //navigation path from a start
//and end node.
//*******************************************************************

public class ch5p1_main {

        public static void main(String[] args) {

// These objects used to define what your graph looks // like
                ch5p1_Graph searchGraph = new ch5p1_Graph();
ch5p1_GraphNode A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, R, S;
ch5p1_Link AB, AC, AD, AE, BF, BM, BG, CH, CI, DJ, EK, EL, MN, MO, IP, PR, PS;

                // define each node
                A = new ch5p1_GraphNode("A");
                B = new ch5p1_GraphNode("B");
                C = new ch5p1_GraphNode("C");
                D = new ch5p1_GraphNode("D");
                E = new ch5p1_GraphNode("E");
                F = new ch5p1_GraphNode("F");
                G = new ch5p1_GraphNode("G");
                H = new ch5p1_GraphNode("H");
                I = new ch5p1_GraphNode("I");
                J = new ch5p1_GraphNode("J");
                K = new ch5p1_GraphNode("K");
                L = new ch5p1_GraphNode("L");
                M = new ch5p1_GraphNode("M");
                N = new ch5p1_GraphNode("N");
                O = new ch5p1_GraphNode("O");
                P = new ch5p1_GraphNode("P");
                R = new ch5p1_GraphNode("R");
                S = new ch5p1_GraphNode("S");

                // define which GraphNodes are connected
                AB = new ch5p1_Link(A, B);
                AC = new ch5p1_Link(A, C);
                AD = new ch5p1_Link(A, D);
                AE = new ch5p1_Link(A, E);
                BF = new ch5p1_Link(B, F);
                BM = new ch5p1_Link(B, M);
                BG = new ch5p1_Link(B, G);
                CH = new ch5p1_Link(C, H);
                CI = new ch5p1_Link(C, I);
                DJ = new ch5p1_Link(D, J);
                EK = new ch5p1_Link(E, K);
```

```
EL = new ch5p1_Link(E, L);
MN = new ch5p1_Link(M, N);
MO = new ch5p1_Link(M, O);
IP = new ch5p1_Link(I, P);
PR = new ch5p1_Link(P, R);
PS = new ch5p1_Link(P, S);

// add all nodes and links to your graph object
searchGraph.addNode(A);
searchGraph.addNode(B);
searchGraph.addNode(C);
searchGraph.addNode(D);
searchGraph.addNode(E);
searchGraph.addNode(F);
searchGraph.addNode(G);
searchGraph.addNode(H);
searchGraph.addNode(I);
searchGraph.addNode(J);
searchGraph.addNode(K);
searchGraph.addNode(L);
searchGraph.addNode(M);
searchGraph.addNode(N);
searchGraph.addNode(O);
searchGraph.addNode(P);
searchGraph.addNode(R);
searchGraph.addNode(S);

searchGraph.addLink(AB);
searchGraph.addLink(AC);
searchGraph.addLink(AD);
searchGraph.addLink(AE);
searchGraph.addLink(BF);
searchGraph.addLink(BM);
searchGraph.addLink(BG);
searchGraph.addLink(CH);
searchGraph.addLink(CI);
searchGraph.addLink(DJ);
searchGraph.addLink(EK);
searchGraph.addLink(EL);
searchGraph.addLink(MN);
searchGraph.addLink(MO);
searchGraph.addLink(IP);
searchGraph.addLink(PR);
searchGraph.addLink(PS);

// run depth-first search to get from start to
// destination
searchGraph.dfsTraverse(
                searchGraph.nodes.get(searchGraph.nodes.indexOf(A)),
                searchGraph.nodes.get(searchGraph.nodes.indexOf(S)));
                // display path created using dfsTraverse
                // This will be display the path from start to
                // destination

                System.out.println("the path from your current city to the destination city
                is: ");
                for (int i = searchGraph.dfsPath.size() - 1; i >= 0; i--) {
                                        if(i!=0)
                System.out.print(searchGraph.dfsPath.get(i).cityName + "->");
                        else                            System.out.print(searchGraph.dfsPath.
                                                        get(i).cityName);
                }
        }
}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3
//ch5p1_GraphNode.java
//Represents name of a graph
//*******************************************************************

public class ch5p1_GraphNode {
        String cityName;

        public ch5p1_GraphNode(String cityName) {
                this.cityName = cityName;
        }

        public String toString() {
                return cityName;
        }
}

//*******************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch5p1_Graph.java
//Represents GraphNodes connected through Links, including method //for doing a depth-first
//search traversal of the graph.
//the method dfsTraverse creates a dfsPath list which the robot will //follow to demonstrate
//how the depth-first search works.
//*******************************************************************************************

import java.util.ArrayList;
import java.util.Stack;

public class ch5p1_Graph {

        // nodes and links define the physical creation of your Graph
        ArrayList<ch5p1_GraphNode> nodes;
        ArrayList<ch5p1_Link> links;

        // Two lists used for traversing
        ArrayList<ch5p1_GraphNode> dfsTraverse;
        Stack<ch5p1_Link> travelStack = new Stack<ch5p1_Link>();
```

```
// List used to define where you would like to move
ArrayList<ch5p1_GraphNode> dfsPath = new ArrayList<ch5p1_GraphNode>();

// Constructor of ch5p1_Graph class
public ch5p1_Graph() {
        nodes = new ArrayList<ch5p1_GraphNode>();
        links = new ArrayList<ch5p1_Link>();
        dfsTraverse = new ArrayList<ch5p1_GraphNode>();
}// end constructor

// addNode()
// Add a city node to the graph

public void addNode(ch5p1_GraphNode node) {
        nodes.add(node);
}// end addNode

// addLink
// Add link to the graph

public void addLink(ch5p1_Link link) {
        links.add(link);
}// end addLink

// dfsTraverse()
// perform depth-first search on the graph

public void dfsTraverse(ch5p1_GraphNode from, ch5p1_GraphNode to) {
        boolean matched;
        ch5p1_Link found;

        // determine if there is a link between from and to
        // if there is a match then add the link to the
        // travelStack and
        // add the nodes to dfsPath
        // This will ultimately repeated by the end of the
        // search

        matched = match(from, to);
        if (matched) {
                travelStack.push(new ch5p1_Link(from, to));
                dfsPath.add(new ch5p1_GraphNode(to.cityName));
                dfsPath.add(new ch5p1_GraphNode(from.cityName));
                return;
        }

        // if there is no match found you could another path
        // findings
        found = find(from);
```

```
// if you find a new connection then you could add it // to the travelStack
                // and
                // and the start node to dfsPath
// recursively call dfsTraverse with the link's to as // start and our
                // destination as the end

                if (found != null) {
                        travelStack.push(new ch5p1_Link(from, to));
                        dfsTraverse(found.to, to);
                        dfsPath.add(new ch5p1_GraphNode(from.cityName));
                }

                // backtrack if you cannot find a new connection
                else if (travelStack.size() > 0) {
                        found = travelStack.pop();
                        dfsTraverse(found.from, found.to);
                        dfsPath.add(new ch5p1_GraphNode(from.cityName));
                }
        }// end dfsTraverse()

        // find() method is used to
        // find the next link to try exploring

        public ch5p1_Link find(ch5p1_GraphNode from) {

                // iterate through the list of links
                for (int a = 0; a < links.size(); a++) {
                        // link found
                        if (links.get(a).from.equals(from) && !links.get(a).skip) {
                                ch5p1_Link foundList = new ch5p1_Link(links.get(a).from,
                                                links.get(a).to);
                                // mark this link as used so we don't
// match it again
                                links.get(a).skip = true;
                                return foundList;
                        }
                }
                return null; // not found
        }// end find()

        // match() method is used to determine if there is a link
// between a starting
        // node and an ending node

        public boolean match(ch5p1_GraphNode from, ch5p1_GraphNode to) {

                // iterate through list of links
                for (int a = links.size() - 1; a >= 0; a--) {
                        if (links.get(a).from.equals(from) && links.get(a).to.equals(to)
                                        && !links.get(a).skip) {
                                links.get(a).skip = true;
                                return true;
                        }
                }
                return false;
        }// end match()
}// end Graph.java

//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch5p1_Link.java
//Represents a link between two GraphNodes
//*****************************************************************

public class ch5p1_Link {

        ch5p1_GraphNode from;
        ch5p1_GraphNode to;

// boolean skip is used for traversal to determine if the path // has already
        // been visited or not
        boolean skip;

        public ch5p1_Link(ch5p1_GraphNode from, ch5p1_GraphNode to) {
                this.from = from;
                this.to = to;
                skip = false;
        }
}
```

In the above program, the `dfsTraverse()` method applied a recursive approach to perform the depth-first search in which a `match()` method was used to check the route graph and determine if there is any path in between from and to. If the answer was yes, then it obtained the information and pushed the route connection information into the stack and then returned the route path. Otherwise, if the answer was no, it called a `find()` method to check the path between from and the other cities, if it can find one, the program then returned the object of connection information, otherwise the program returned null. If it can find such a path, it could then push the information to the top of stack, then call `dfsTraverse()` recursively, and save the city into the new departure cities. Otherwise, the program backtracked and called `dfsTraverse()` recursively until it found the destination.

The result of running this program is as follows:

```
The path from your current city to the destination city is:
A->B->F->B->M->N->M->O->M->B->G->B->A->C->H->C->I->P->S
```

This result shows you the exact path when traversing a graph using a DFS search strategy including the trace back in more detail.

## leJOS EV3-Based DFS Algorithm

In a leJOS-based DFS algorithm, each node on the path is a class node called *WPNode*, which is defined as:

```
        public WPNode(String newname, WayPoint newwp) {
                nodename = newname;
                nodewp = newwp;
                seen = false;
```

```
            parent = this;
            connections = new ArrayList<WPNode>();
    }
```

The pseudocode for the leJOS-based DFS algorithm is described in the following:

1. Constructing the generic tree for search space, such as:

    ```
    A = new WPNode("A", new WayPoint(0, 0));
    B = new WPNode("B", new WayPoint(-5, 5));
    C = new WPNode("C", new WayPoint(5, 5));
    A.addLeaf(B);
    A.addLeaf(C);
    ```

2. Declaring a stack to save the route path, for example:

    ```
    Stack<WPNode> DFSpath = new Stack<WPNode>();
    ```

3. Setting the current node to the root node, say *A*. While the destination node is not found, loop the following:

    a. If the current node has children, set the first unseen node to the current node and then return.

    b. If the current node has no unseen children, set its parent to the current node and then return.

4. Once the destination node is found, push the destination node to the stack and then push each parent node to the stack.

5. Generate pilot for two-motor movements, and then set pilot to use appropriate dimensions and motors.

6. Pop the waypoint of each path node, and apply the `goto(int x, int y)` method to direct the robot to move to the next node.

Based on the above leJOS-based DFS algorithm , you can develop a program for your robot so that it can travel the path between starting node A and destination node M, as illustrated in Figure 5-3:

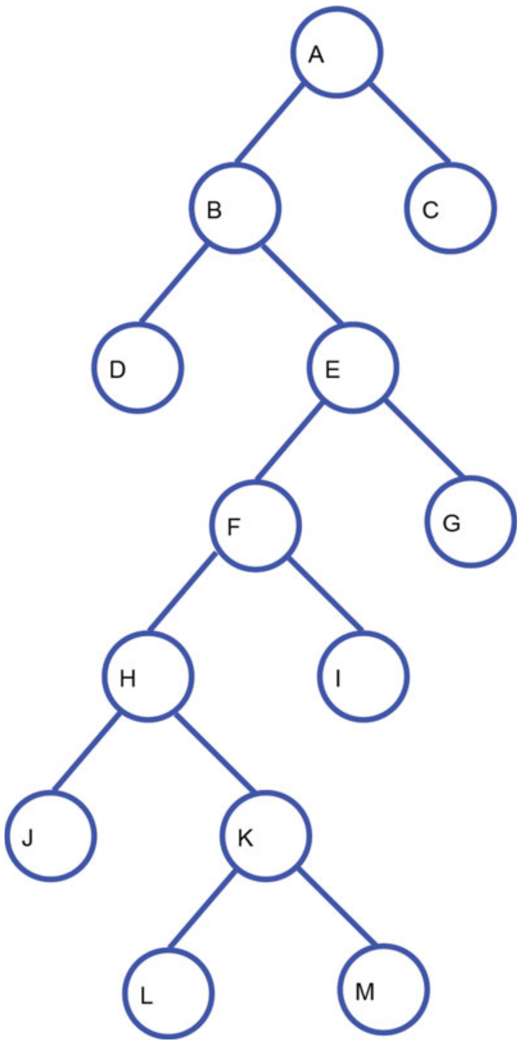


***Figure 5-3.*** *Path in between two nodes*

The coordinate at A is (0,0).

The coordinate at B is (-5,5).

The coordinate at C is (5,5).

The coordinate at D is (-10,10).

The coordinate at E is (0,10).

The coordinate at F is (-5,15).

The coordinate at G is (5,15).

The coordinate at H is (-10,20).

The coordinate at I is (0,20).

The coordinate at J is (-15,25).

The coordinate at K is (-5,25).

The coordinate at L is (-10,30).

The coordinate at M is (0,30).

Your program should at least display the destination's coordinate on the LCD and then display the message "`Press ENTER key to continue.`" Press enter, and your robot moves to the next node. For instance, suppose that your robot starts from `A (0,0)` and you want to explore a path to `E (0,10)`. Assume that the path your robot explores using a depth-first search is `A -> B -> E`. At the starting point A, your program should do the following:

Display the destination's coordinate `B(-5,5)` on the LCD and display the message "`Press ENTER key to continue`".

Go to the location with coordinate `-5,5`.

Display the destination's coordinate `E(0, 10)` on the LCD and display the message "`Press ENTER key to continue.`"

Go to location with coordinate `0,10`.

Moreover, your problem should have a string called *destination,* so it's intelligent enough when changing the value of the destination, your robot can explore a new path from starting node A to the new destination node. (We assume that the starting node is always A, so the from string can be hard-coded.)

The following programs represent the implementation of a leJOS-based DFS algorithm to explore a path from node A to the destination node M:

```
//******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch5p2_main.java
//Driver class to set up map using ch5p2_GraphNode, ch5p2_Link, and //ch5p2_Graph classes.
//Calls a depth-first search in ch5p2_Graph class to create //navigation path from a start
//and end node and then robots will follow the path to move from //start node
//to destination node
//******************************************************************

// import EV3 hardware packages for EV brick finding, activating
// keys and LCD
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.MovePilot;
import lejos.robotics.navigation.Navigator;

public class ch5p2_main {

        static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(
                        MotorPort.A);
        static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(
                        MotorPort.C);

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();

// setup the wheel diameter of left (and right) motor // in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the center // of wheel to
                // the center of robot, i.e. half of track width
                Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
                Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

                // set up the chassis type, i.e. Differential pilot
                Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                                WheeledChassis.TYPE_DIFFERENTIAL);
                MovePilot ev3robot = new MovePilot(chassis);

                Navigator navbot = new Navigator(ev3robot);

// These objects used to define what your graph looks // like
                ch5p2_Graph searchGraph = new ch5p2_Graph();
                ch5p2_GraphNode A, B, C, D, E, F, G, H, I, J, K, L, M;
                ch5p2_Link AB, AC, BD, BE, EF, EG, FH, FI, HJ, HK, KL, KM;

                // define each node
                A = new ch5p2_GraphNode("A", 0, 0);
                B = new ch5p2_GraphNode("B", -5, 5);
                C = new ch5p2_GraphNode("C", 5, 5);
                D = new ch5p2_GraphNode("D", -10, 10);
                E = new ch5p2_GraphNode("E", 0, 10);
                F = new ch5p2_GraphNode("F", -5, 15);
                G = new ch5p2_GraphNode("G", 5, 15);
                H = new ch5p2_GraphNode("H", -10, 20);
                I = new ch5p2_GraphNode("I", 0, 20);
                J = new ch5p2_GraphNode("J", -15, 25);
                K = new ch5p2_GraphNode("K", -5, 25);
                L = new ch5p2_GraphNode("L", -10, 30);
                M = new ch5p2_GraphNode("M", 0, 30);

                // define which GraphNodes are connected
                AB = new ch5p2_Link(A, B);
                AC = new ch5p2_Link(A, C);
                BD = new ch5p2_Link(B, D);
                BE = new ch5p2_Link(B, E);
                EF = new ch5p2_Link(E, F);
                EG = new ch5p2_Link(E, G);
                FH = new ch5p2_Link(F, H);
                FI = new ch5p2_Link(F, I);
                HJ = new ch5p2_Link(H, J);
                HK = new ch5p2_Link(H, K);
                KL = new ch5p2_Link(K, L);
                KM = new ch5p2_Link(K, M);

                // add all nodes and links to your graph object
                searchGraph.addNode(A);
                searchGraph.addNode(B);
                searchGraph.addNode(C);
                searchGraph.addNode(D);
                searchGraph.addNode(E);
                searchGraph.addNode(F);
                searchGraph.addNode(G);
                searchGraph.addNode(H);
                searchGraph.addNode(I);
                searchGraph.addNode(J);
                searchGraph.addNode(K);
                searchGraph.addNode(L);
                searchGraph.addNode(M);

                searchGraph.addLink(AB);
                searchGraph.addLink(AC);
                searchGraph.addLink(BD);
                searchGraph.addLink(BE);
                searchGraph.addLink(EF);
                searchGraph.addLink(EG);
                searchGraph.addLink(FH);
                searchGraph.addLink(FI);
                searchGraph.addLink(HJ);
                searchGraph.addLink(HK);
                searchGraph.addLink(KL);
                searchGraph.addLink(KM);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // run depth-first search to get from start to
// destination
                searchGraph.dfsTraverse(
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(A)),
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(M)));

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// Robot moves through path from start to destination // by using
                // dfsTraverse
                for (int i = searchGraph.dfsPath.size() - 1; i >= 0; i--) {
                        // go to node
                        navbot.goTo(searchGraph.dfsPath.get(i).xLocation,
                                searchGraph.dfsPath.get(i).yLocation);

                        LCD.clear();

                        // display current location
                        LCD.drawString("At location " + searchGraph.dfsPath.get(i).cityName
                                + ", ", 0, 0);

                        LCD.drawString(searchGraph.dfsPath.get(i).xLocation + ", "
                                + searchGraph.dfsPath.get(i).yLocation, 0, 1);

                        LCD.drawString("Press ENTER key", 0, 2);

                        // block the thread until a button is pressed
                        buttons.waitForAnyPress();
                }
        }
}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3
// ch5p2_GraphNode.java
//Represents name and coordinates of a node on a graph
//*******************************************************************

public class ch5p2_GraphNode {
        String cityName;
        int xLocation, yLocation;

        public ch5p2_GraphNode(String cityName, int xLocation, int yLocation) {
                this.cityName = cityName;
                this.xLocation = xLocation;
                this.yLocation = yLocation;

        }

        public String toString() {
                return cityName + " ("+xLocation +"," + yLocation + ")";
        }
}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch5p2_Link.java
//Represents a link between two GraphNodes
//*******************************************************************
public class ch5p2_Link {

        ch5p2_GraphNode from;
        ch5p2_GraphNode to;
```

```
// boolean skip is used for traversal to determine if the path // has already
        // been visited or not
        boolean skip;

        public ch5p2_Link(ch5p2_GraphNode from, ch5p2_GraphNode to) {
                this.from = from;
                this.to = to;
                skip = false;
        }
}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch5p2_Graph.java
//Represents GraphNodes connected through Links, including method //for doing a depth-first
//search of the graph.
//the method dfsTraverse creates a dfsPath list which the robot will //follow to demonstrate
//how the depth-first search works.
//*******************************************************************

import java.util.ArrayList;
import java.util.Stack;

public class ch5p2_Graph {

        // nodes and links define the physical creation of your Graph
        ArrayList<ch5p2_GraphNode> nodes;
        ArrayList<ch5p2_Link> links;

        // Two lists used for traversing
        ArrayList<ch5p2_GraphNode> dfsTraverse;
        Stack<ch5p2_Link> travelStack = new Stack<ch5p2_Link>();

        // List used to define where you would like to move
        ArrayList<ch5p2_GraphNode> dfsPath = new ArrayList<ch5p2_GraphNode>();

        // Constructor of ch5p2_Graph class
        public ch5p2_Graph() {
                nodes = new ArrayList<ch5p2_GraphNode>();
                links = new ArrayList<ch5p2_Link>();
                dfsTraverse = new ArrayList<ch5p2_GraphNode>();
        }// end constructor

        // addNode()
        // Add a city node to the graph

        public void addNode(ch5p2_GraphNode node) {
                nodes.add(node);
        }// end addNode

        // addLink
        // Add link to the graph
        public void addLink(ch5p2_Link link) {
                links.add(link);
        }// end addLink

        // dfsTraverse()
        // perform depth-first search on the graph

        public void dfsTraverse(ch5p2_GraphNode from, ch5p2_GraphNode to) {
                boolean matched;
                ch5p2_Link found;

                // determine if there is a link between from and to
                // if there is a match then add the link to the
// travelStack and
                // add the nodes to dfsPath
                // This will be ultimately repeated by the end of the
                 // search

                matched = match(from, to);
                if (matched) {
                        travelStack.push(new ch5p2_Link(from, to));
                        dfsPath.add(new ch5p2_GraphNode(to.cityName,to.xLocation,
                        to.yLocation));
                        dfsPath.add(new ch5p2_GraphNode(from.cityName,from.xLocation, from.
                        yLocation));
                        return;
                }

                // if there is no match found you could another path
// findings
                found = find(from);

// if you find a new connection then you could add it // to the travelStack
                // and
                // and the start node to dfsPath
// recursively call dfsTraverse with the link's to as // start and our
                // destination as the end

                if (found != null) {
                        travelStack.push(new ch5p2_Link(from, to));
                        dfsTraverse(found.to, to);
                        dfsPath.add(new ch5p2_GraphNode(from.cityName,from.xLocation, from.
                        yLocation));
                }

                // backtrack if you cannot find a new connection
                else if (travelStack.size() > 0) {
                        found = travelStack.pop();
                        dfsTraverse(found.from, found.to);
                        dfsPath.add(new ch5p2_GraphNode(from.cityName,from.xLocation, from.
                        yLocation));
                }
        }// end dfsTraverse()

        // find() method is used to
        // find the next link to try exploring

        public ch5p2_Link find(ch5p2_GraphNode from) {

                // iterate through the list of links
                for (int a = 0; a < links.size(); a++) {
                        // link found
                        if (links.get(a).from.equals(from) && !links.get(a).skip) {
                                ch5p2_Link foundList = new ch5p2_Link(links.get(a).from,
                                                links.get(a).to);
                                // mark this link as used so we don't match it again
                                links.get(a).skip = true;
                                return foundList;
                        }
                }
                return null; // not found
        }// end find()

        // match() method is used to determine if there is a link
// between a starting
        // node and an ending node

        public boolean match(ch5p2_GraphNode from, ch5p2_GraphNode to) {

                // iterate through list of links
                for (int a = links.size() - 1; a >= 0; a--) {
                        if (links.get(a).from.equals(from) && links.get(a).to.equals(to)
                                        && !links.get(a).skip) {
                                links.get(a).skip = true;
                                return true;
                        }
                }
                return false;
        }// end match()
}// end Graph.java
```

## Summary

In this chapter, you learned the fundamentals of a depth-first search algorithm and now know how to apply it to solve the searching program in practice. You also learned how to build problem-solving agents (that is, your robot) based on the depth-first search algorithm and the Navigation class that you studied in previous chapters in which the problem-solving agents were able to find a route path intelligently from a starting point to any destination.

In the next chapter, you will learn how to implement a breadth-first search (BFS) algorithm on leJOS EV3 and how to integrate the implemented BFS algorithm in the leJOS-based robotics system for localization and path planning.

CHAPTER 6

# Breadth-First Search and Its Implementation with Lego Mindstorms

Just as you were introduced to the depth-first search (DFS) algorithm in Chapter 5, in this chapter you will learn how to implement a *breadth-first search (BFS)* algorithm in leJOS EV3. Specifically, this chapter will cover the following topics:

- A new breadth-first search (BFS) algorithm that can be applied to build arbitrary tree structures generically.
- Applying and integrating the proposed BFS algorithm in the leJOS-based robotics system for localization and path planning, which enhances the existing pathfinding approaches within the leJOS system.

## Overview of BFS Algorithm

Let's first review how humans solve a search problem. First, you need a representation of how your search problem exists. Figure 6-1 is an example of your search tree. It shows a series of interconnected nodes through which you will be searching:

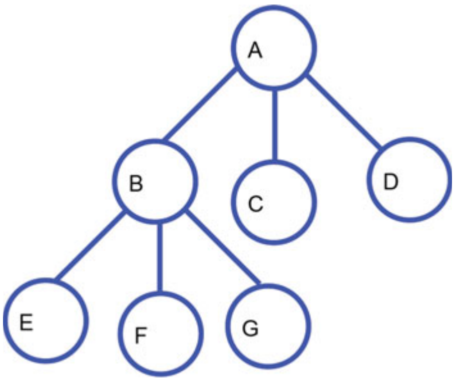


***Figure 6-1.*** *Tree structure of a path*

The breadth-first search algorithm works by taking a node; checking its neighbors; expanding the first node it finds among its neighbors; checking to see if that expanded node is its destination; and if not, continues exploring the other neighboring nodes. For example, if you want to find a path from A to E, you can use two lists to keep track of what you are doing: an open list and a closed list. An open list keeps track of what you need to do, and a closed list keeps track of what you have already done.

At the beginning, you only have your starting point, node A. You haven't done anything to it yet, so let's add it to your open list. Now you have an open list that includes `<A>` and a closed list that includes `<empty>`.

Now let's explore the neighbors of your A node. Node A's neighbors are the B, C, and D nodes. Because you are now finished with the A node, you can remove it from your open list and add it to your closed list. Then the current open list includes `<B, C, D>`, and the closed list contains `<A>`. Now your open list contains three items.

For breadth-first search, you always explore the first node from your open list. The first item in the open list is the B node. B is not your destination, so now let's explore its neighbors. Because you have now expanded B, you are going to remove it from the open list and add it to the closed list. Your new nodes are E, F, and G, and you can add these nodes to the end of your open list. Then you have a closed list that includes `<A,B>` and an open list that includes `< C, D, E, F, G>`.

Next you expand the C node. Since it is not the intended destination, you can remove it from your open list and add it to your closed list. The current open list then includes `<D, E, F, G>`, and the closed list contains `<A, B, C>`. C has no child, so you move on to visit node D. Since it is also not the intended destination, you can remove it from your open list and add it to your closed list. The current open list thus includes `<E, F, G>`, and the and closed list contains `<A, B, C, D>`.

Next you expand node E. Since it is your intended destination, you stop. Thus you receive the route `A->B->E` that is interpreted from the closed list by using the regular breadth-first search algorithm.

Next let's see how to use the breadth-first search approach to solve a path-finding problem that could be applied to a GPS system when navigating from a starting city to a destination city. Suppose that you want drive from city A (for example, Keene, NH) to city S (for instance, Boston, MA). Given the following route path, work out a plan for you to start from A and reach S using a breadth-first search strategy.

| City | Distance |
|---|---|
| A to B | 20 miles |
| A to C | 10 miles |
| A to D | 10 miles |
| A to E | 20 miles |
| B to F | 10 miles |
| B to M | 20 miles |
| B to G | 10 miles |
| C to H | 10 miles |
| C to I | 15 miles |
| D to J | 20 miles |
| E to K | 15 miles |
| E to L | 15 miles |
| M to N | 20 miles |
| M to O | 20 miles |
| I to P | 40 miles |
| P to R | 20 miles |
| P to S | 20 miles |

Based on the above information, you can plot a route tree as illustrated in Figure 6-2, displaying all of the possible routes in between the two cities:

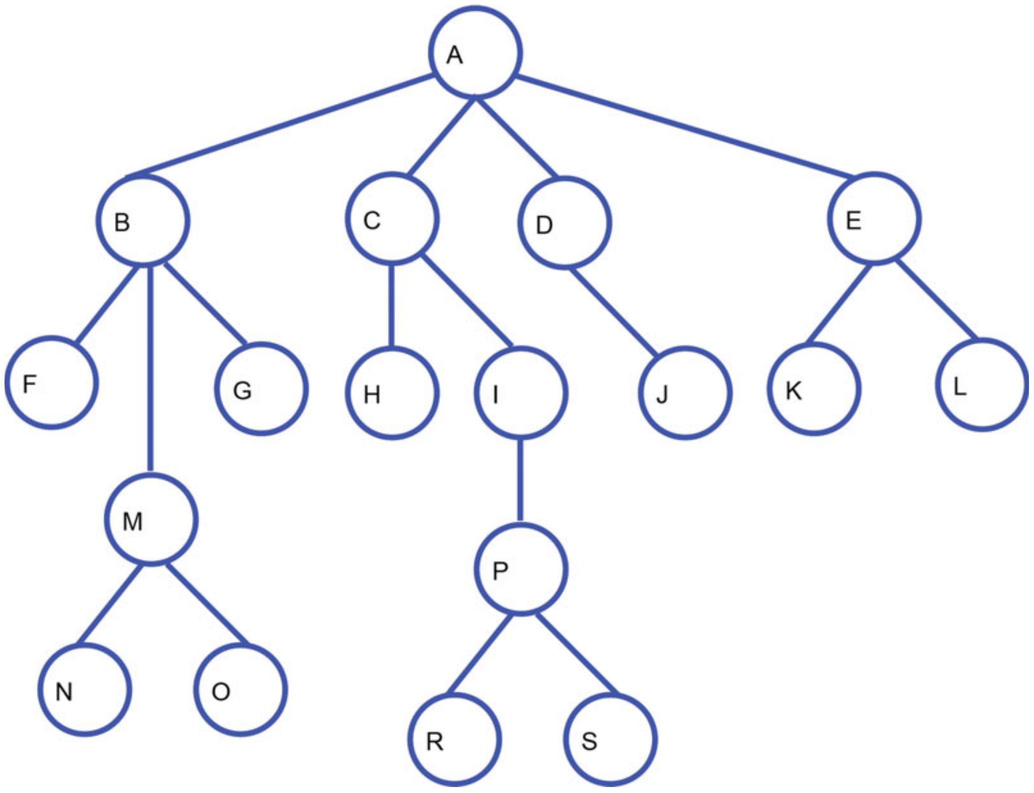


***Figure 6-2.*** *Routes in between two cities*

The following program can be used to find a path for you to schedule a travel plan automatically by using the breadth-first search algorithm:

```
//********************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch6p1_main.java
//Driver class to call a breadth-first search algorithm
//to create navigation path from a start to an end node.
//********************************************************************

import java.util.*;

class ch6p1_main {

// This array holds the connection information between two cities.
        ch6p1_GraphNode cityLinks[] = new ch6p1_GraphNode[200];

// number of path connections on the route graph
        int numConnections = 0;
```

```
// backtrack stack to store the city nodes
        Stack closedList = new Stack();

        public static void main(String args[]) {

                // define the start city name
                String from = "A";

                // define the destination city name
                String to = "S";

                ch6p1_main bfsTraveral = new ch6p1_main();

                // set up the route graph
                bfsTraveral.graph();

                bfsTraveral.isNode(from, to);

                if (bfsTraveral.closedList.size() != 0) {
                        System.out
                                        .println("the path from your current city to the
                                         destination city is: ");
                        bfsTraveral.BFSroute(to);
                }

        }

        // Add link between two cities

        void addLink(String parent, String child) {
                if (numConnections < 200) {
                        cityLinks[numConnections] = new ch6p1_GraphNode(parent, child);
                        numConnections++;
                } else
                        System.out.println("error to add link\n");
        }

        // Initialize the path link to construct the graph.

        void graph() {

                addLink("A", "B");
                addLink("A", "C");
                addLink("A", "D");
                addLink("A", "E");
                addLink("B", "F");
                addLink("B", "M");
                addLink("B", "G");
                addLink("C", "H");
                addLink("C", "I");
                addLink("D", "J");
                addLink("E", "K");
                addLink("E", "L");
                addLink("M", "N");
                addLink("M", "O");
                addLink("I", "P");
                addLink("P", "R");
                addLink("P", "S");
        }

        // determine to see if there is a link matched between
// startCity and endCity
        // if match found, return true, otherwise return false

        int matched(String from, String to) {
                for (int i = numConnections - 1; i > -1; i--) {
                        if (cityLinks[i].startCity.equals(from)
                                        && cityLinks[i].endCity.equals(to) && !cityLinks[i].
                                        visited) {

                        // set up visited to true to prevent re-visit
                                cityLinks[i].visited = true;

                        // match found
                                return 1;
                        }
                }
                // match not found
                return 0;
        }

        // Given parent to find any child connected with this parent

        ch6p1_GraphNode find(String parent) {
                for (int i = 0; i < numConnections; i++) {
                        if (cityLinks[i].startCity.equals(parent) && !cityLinks[i].visited) {

                                ch6p1_GraphNode f = new ch6p1_GraphNode(cityLinks[i].
                                startCity,
                                                cityLinks[i].endCity);

                        // set up visited to true to prevent re-visit
                                cityLinks[i].visited = true;

                        // child (or leaf) returned
                                return f;
                        }
                }

                // if parent has no child return nothing
                return null;
        }
```

```
// using breadth-first search and determining if there is any // route existing
        // in between startCity and endCity

        void isNode(String from, String to) {

                int directconn;
                ch6p1_GraphNode citynode;

                Stack resetList = new Stack();

        // determine if there is any direct link between from and to
        // if yes push the link of the two cities into closedList
// stack
                directconn = matched(from, to);
                if (directconn != 0) {
                        closedList.push(new ch6p1_GraphNode(from, to));
                        return;
                }

                // find all the children cities connected with the
// specified parent node

                while ((citynode = find(from)) != null) {
                        resetList.push(citynode);

                        // check further if there is any direct
// connection between the child
                        // and grandchild

                        if ((directconn = matched(citynode.endCity, to)) != 0) {
                                resetList.push(citynode.endCity);
                                closedList.push(new ch6p1_GraphNode(from, citynode.
                                endCity));
                                closedList.push(new ch6p1_GraphNode(citynode.endCity, to));
                                return;
                        }
                }

                // reset the visited boolean to unvisited and do the
// breadth first
                // search next

                for (int i = resetList.size(); i != 0; i--)
                        resetSkip((ch6p1_GraphNode) resetList.pop());

                // then try the next neighboring city nodes
                citynode = find(from);
                if (citynode != null) {
```

```
                closedList.push(new ch6p1_GraphNode(from, to));
                isNode(citynode.endCity, to);
            } else if (closedList.size() > 0) {

                // trace back and try another link
                citynode = (ch6p1_GraphNode) closedList.pop();
                isNode(citynode.startCity, citynode.endCity);
            }
        }

        // reset visited field of specified parent city node

        void resetSkip(ch6p1_GraphNode citynode) {
            for (int i = 0; i < numConnections; i++)
                if (cityLinks[i].startCity.equals(citynode.startCity)
                        && cityLinks[i].endCity.equals(citynode.endCity))
                    cityLinks[i].visited = false;
        }

        // Show the route obtained by the BFS algorithm

        void BFSroute(String to) {

            int num = closedList.size();

            Stack reverseList = new Stack();

            ch6p1_GraphNode citynode;

            // Reverse the stack to show the path

            for (int i = 0; i < num; i++)
                reverseList.push(closedList.pop());

            for (int i = 0; i < num; i++) {
                citynode = (ch6p1_GraphNode) reverseList.pop();
                System.out.print(citynode.startCity + " -> ");
            }
            System.out.println(to);
        }

}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3
//ch6p1_GraphNode.java
//Represents name of a graph
//*******************************************************************
```

```
public class ch6p1_GraphNode {

        String startCity;
        String endCity;

        // to determine if the city has been visited or not
        boolean visited;

        ch6p1_GraphNode(String s, String d) {
                startCity = s;
                endCity = d;
                visited = false;
        }
}
```

Given the `startCity` A and `endCity` S, the result of running the program is as follows:

```
The path from your current city to the destination city is this:
A -> C -> I -> P -> S
```

This result shows you the exact path when traversing a graph using a BFS search strategy without including the intermediate trace-back nodes.

## leJOS EV3-Based BFS Algorithm

In a leJOS-based BFS algorithm, each node on the path is a class node called `WPNode`, which is defined as the following:

```
        public WPNode(String newname, WayPoint newwp) {
                nodename = newname;
                nodewp = newwp;
                seen = false;
                parent = this;
                connections = new ArrayList<WPNode>();
        }
```

Following is a description of the pseudocode for the leJOS-based BFS algorithm:

1. Constructing the generic tree for search space, such as:

```
                A = new WPNode("A", new WayPoint(0, 0));
                B = new WPNode("B", new WayPoint(-5, 5));
                C = new WPNode("C", new WayPoint(5, 5));
                A.addLeaf(B);
                A.addLeaf(C);
```

2. Declare a stack to save the route path, such as:

   ```
   Stack<WPNode> BFSpath = new Stack<WPNode>();
   ```

3. Set the current node to root node, say *A*. While the destination node is not found, loop the following:

   a. If the current node has children, set the first unseen node to the current node and then return.

   b. If the current node has no unseen children, set its parent to the current node and then return.

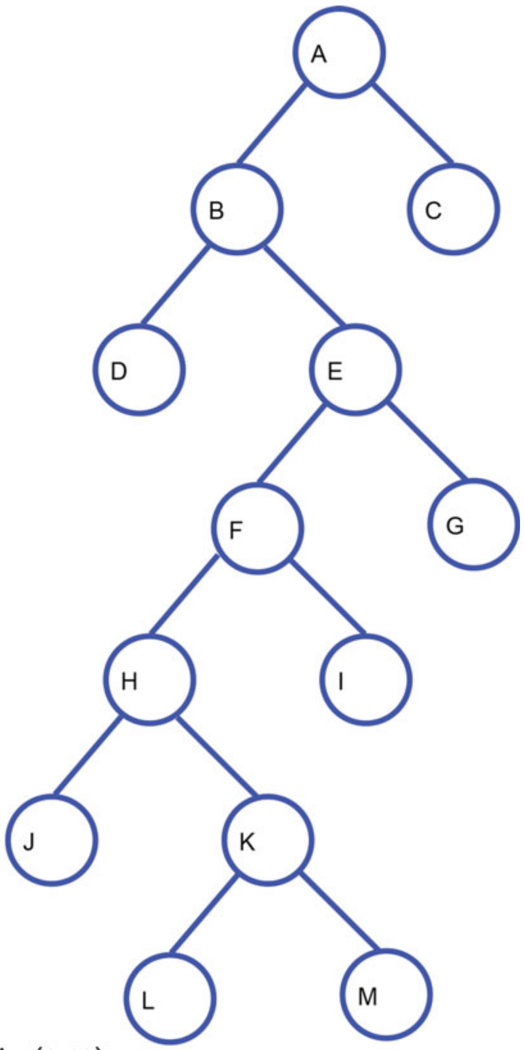
4. Once the destination node is found, push the destination node to the stack and then push each parent node to the stack.

5. Generate pilot for two-motor movement, and then set pilot to use appropriate dimensions and motors.

6. Pop the waypoint of each path node and apply the `goto(int x, int y)` method to direct robots moving to the next node.

Based on the above leJOS-based BFS algorithm, you can develop a program for your robot so that it can travel the path between starting node A and destination node M, as illustrated in Figure 6-3 in which you have the following:

```
The coordinate at A is (0,0).
The coordinate at B is (-5,5).
The coordinate at C is (5, 5).
The coordinate at D is (-10,10).
The coordinate at E is (0,10).
The coordinate at F is (-5,15).
The coordinate at G is (5,15).
The coordinate at H is (-10,20).
The coordinate at I is (0,20).
The coordinate at J is (-15,25).
The coordinate at K is (-5,25).
The coordinate at L is (-10,30).
The coordinate at M is (0,30).
```



```
The coordinate at I is (0,20).
The coordinate at J is (-15,25).
The coordinate at K is (-5,25).
The coordinate at L is (-10,30).
The coordinate at M is (0,30).
```

***Figure 6-3.*** *Path in between two nodes*

Your program should at least display the destination's coordinate on the LCD and then display a message "`Press ENTER key to continue`." Press Enter, and your robot moves to the next node. For example, suppose that your robot starts from `A (0,0)` and you want to explore a path to `G (5,15)`. Assume that the path that your robot explores using breadth-first search is `A -> B -> E -> G`. At starting point A, your program should do the following:

> Display the destination's coordinate `B(-5, 5)` on the LCD and show the message "`Press ENTER key to continue`."

Go to the location with coordinate `-5,5`.

Display the destination's coordinate `E(0, 10)` on the LCD, and show the message "`Press ENTER key to continue`."

Go to the location with coordinate `0,10`.

Display the destination's coordinate `G(5, 15)` on the LCD, and show the message "`Press ENTER key to continue`."

Go to the location with coordinate `5,15`.

Furthermore, your problem should have a string called `destination`, so that it's intelligent enough when changing the value of the destination, your robot can explore a new path from starting node A to the new destination node. (We assume the starting node is always A, so the from string can be hard-coded.)

The following programs represent the implementation of a leJOS-based BFS algorithm to explore a path from node A to destination node M:

```
//********************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch6p2_main.java
//Driver class to set up map using ch6p2_GraphNode, ch6p2_Link, and //ch6p2_Graph classes.
//Calls a breadth-first search in ch6p2_Graph class to create //navigation path from a start
//and end node and then robots will follow the path to move from //start node
//to destination node
//********************************************************************************************

// import EV3 hardware packages for EV brick finding, activating
// keys and LCD
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.MovePilot;
import lejos.robotics.navigation.Navigator;

public class ch6p2_main {

        static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(
                        MotorPort.A);
        static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(
                        MotorPort.C);

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();
```

```
// setup the wheel diameter of left (and right) motor // in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the center // of wheel to
                // the center of robot, i.e. half of track width
                Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
                Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

                // set up the chassis type, i.e. Differential pilot
                Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                                WheeledChassis.TYPE_DIFFERENTIAL);
                MovePilot ev3robot = new MovePilot(chassis);

                Navigator navbot = new Navigator(ev3robot);

        // These objects used to define what your graph looks like
                ch6p2_Graph searchGraph = new ch6p2_Graph();
                ch6p2_GraphNode A, B, C, D, E, F, G, H, I, J, K, L, M;
                ch6p2_Link AB, AC, BD, BE, EF, EG, FH, FI, HJ, HK, KL, KM;

                // define each node
                A = new ch6p2_GraphNode("A", 0, 0);
                B = new ch6p2_GraphNode("B", -5, 5);
                C = new ch6p2_GraphNode("C", 5, 5);
                D = new ch6p2_GraphNode("D", -10, 10);
                E = new ch6p2_GraphNode("E", 0, 10);
                F = new ch6p2_GraphNode("F", -5, 15);
                G = new ch6p2_GraphNode("G", 5, 15);
                H = new ch6p2_GraphNode("H", -10, 20);
                I = new ch6p2_GraphNode("I", 0, 20);
                J = new ch6p2_GraphNode("J", -15, 25);
                K = new ch6p2_GraphNode("K", -5, 25);
                L = new ch6p2_GraphNode("L", -10, 30);
                M = new ch6p2_GraphNode("M", 0, 30);

                // define which GraphNodes are connected
                AB = new ch6p2_Link(A, B);
                AC = new ch6p2_Link(A, C);
                BD = new ch6p2_Link(B, D);
                BE = new ch6p2_Link(B, E);
                EF = new ch6p2_Link(E, F);
                EG = new ch6p2_Link(E, G);
                FH = new ch6p2_Link(F, H);
                FI = new ch6p2_Link(F, I);
                HJ = new ch6p2_Link(H, J);
                HK = new ch6p2_Link(H, K);
                KL = new ch6p2_Link(K, L);
                KM = new ch6p2_Link(K, M);

                // add all nodes and links to your graph object
                searchGraph.addNode(A);
                searchGraph.addNode(B);
                searchGraph.addNode(C);
                searchGraph.addNode(D);
                searchGraph.addNode(E);
                searchGraph.addNode(F);
                searchGraph.addNode(G);
                searchGraph.addNode(H);
                searchGraph.addNode(I);
                searchGraph.addNode(J);
                searchGraph.addNode(K);
                searchGraph.addNode(L);
                searchGraph.addNode(M);

                searchGraph.addLink(AB);
                searchGraph.addLink(AC);
                searchGraph.addLink(BD);
                searchGraph.addLink(BE);
                searchGraph.addLink(EF);
                searchGraph.addLink(EG);
                searchGraph.addLink(FH);
                searchGraph.addLink(FI);
                searchGraph.addLink(HJ);
                searchGraph.addLink(HK);
                searchGraph.addLink(KL);
                searchGraph.addLink(KM);

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // run breadth-first search to get from start to
                // destination
                searchGraph.bfsTraverse(
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(A)),
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(M)));

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// Robot moves through path from start to destination // by using
                // bfsTraverse
                for (int i = 0; i < searchGraph.bfsTraverse.size(); i++) {

                        // go to node
                        navbot.goTo(searchGraph.bfsPath.get(i).xLocation,
                                        searchGraph.bfsPath.get(i).yLocation);

                        LCD.clear();

                        // display current location
                        LCD.drawString("At location " + searchGraph.bfsPath.get(i).cityName
                                        + ", ", 0, 0);
                        LCD.drawString(searchGraph.bfsPath.get(i).xLocation + ", "
                                + searchGraph.bfsPath.get(i).yLocation, 0, 1);

                        LCD.drawString("Press ENTER key", 0, 2);

                        // block the thread until a button is pressed
                        buttons.waitForAnyPress();

                        if (i == searchGraph.bfsTraverse.size() - 1) {
                                navbot.goTo(searchGraph.bfsTraverse.get(i).to.yLocation,
                                                searchGraph.bfsTraverse.get(i).
                                                to.xLocation);
                                LCD.drawString("At location "
                                                + searchGraph.bfsTraverse.get(i).
                                                  to.cityName, 0, 0);
                                // block the thread until a button is
                                // pressed
                                buttons.waitForAnyPress();
                        }
                }
        }
}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch6p2_Link.java
//Represents a link between two GraphNodes
//*******************************************************************

public class ch6p2_Link {

        ch6p2_GraphNode from;
        ch6p2_GraphNode to;

// boolean skip is used for traversal to determine if the path // has already
        // been visited or not
        boolean skip;

        public ch6p2_Link(ch6p2_GraphNode from, ch6p2_GraphNode to) {
                this.from = from;
                this.to = to;
                skip = false;
        }
}

//*********************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 //ch6p2_GraphNode.java
//Represents name and coordinates of a node on a graph
//*********************************************************************************************

public class ch6p2_GraphNode {
        String cityName;
        int xLocation, yLocation;
        public ch6p2_GraphNode(String cityName, int xLocation, int yLocation) {
                this.cityName = cityName;
                this.xLocation = xLocation;
                this.yLocation = yLocation;

        }

        public String toString() {
                return cityName + " ("+xLocation +"," + yLocation + ")";
        }
}

//*******************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch6p2_Graph.java
//Represents GraphNodes connected through Links, including method //for doing a breadth-
  first
//search traversal of the graph.
//the method bfsTraverse creates a bfsPath list which the robot will //follow to demonstrate
//how the breadth-first search works.
//*******************************************************************************************

import java.util.ArrayList;

public class ch6p2_Graph {
        // nodes and links define the physical creation of your Graph
        ArrayList<ch6p2_GraphNode> nodes;
        ArrayList<ch6p2_Link> links;

        // a list used for traversing
        ArrayList<ch6p2_Link> bfsTraverse;

        // List used to define where you would like to move
        ArrayList<ch6p2_GraphNode> bfsPath = new ArrayList<ch6p2_GraphNode>();

        // Constructor of ch6p2_Graph class
        public ch6p2_Graph() {
                nodes = new ArrayList<ch6p2_GraphNode>();
                links = new ArrayList<ch6p2_Link>();
                bfsTraverse = new ArrayList<ch6p2_Link>();

        }// end constructor

        // addNode()
        // Add a city node to the graph

        public void addNode(ch6p2_GraphNode node) {
                nodes.add(node);
        }// end addNode

        // addLink
        // Add link to the graph
        public void addLink(ch6p2_Link link) {
                links.add(link);
        }// end addLink

        // bfsTraverse()
        // perform breadth-first search on the graph

        public void bfsTraverse(ch6p2_GraphNode from, ch6p2_GraphNode to) {

                ArrayList<ch6p2_Link> resetList = new ArrayList<ch6p2_Link>();
                boolean matched = false;
                ch6p2_Link l;

                // determine if there is a link between from and to
                // if there is a match then add the link to the
                // travelStack and
                // add the nodes to bfsPath
                // This will ultimately repeated by the end of the search

                matched = match(from, to);

                // add to bfsTraverse if match is found
                if (matched) {
                        bfsTraverse.add(new ch6p2_Link(from, to));
                        return;
                }

                // continue while links exist
                while ((l = find(from)) != null) {
                        resetList.add(l);
                        if ((matched = match(l.to, to))) {
                                resetList.add(new ch6p2_Link(l.from, to));
                                bfsTraverse.add(new ch6p2_Link(from, l.to));
                                bfsTraverse.add(new ch6p2_Link(l.to, to));
                                return;
                        }
                }

                for (int i = resetList.size(); i != 0; i--) {
                        resetSkip((ch6p2_Link) resetList.remove(i - 1));
                }

// if you find a new connection then you could add it // to the travelStack
                // and
                // and the start node to bfsPath
// recursively call bfsTraverse with the link's to as // start and our
                // destination as the end

                l = find(from);
                if (l != null) {
                        bfsTraverse.add(new ch6p2_Link(from, to));
                bfsPath.add(new ch6p2_GraphNode(from.cityName, from.xLocation,
                                from.yLocation));
                bfsTraverse(l.to, to);
            }

            // backtrack if you cannot find a new connection
            else if (bfsTraverse.size() > 0) {
                l = (ch6p2_Link) bfsTraverse.remove(bfsTraverse.size() - 1);
                bfsPath.remove(bfsPath.size() - 1);
                bfsTraverse(l.from, l.to);
            }
        }// end bfsTraverse

        // resetSkip
        // when backtracking reset skip flag so we can visit nodes
// again
        public void resetSkip(ch6p2_Link l) {
            for (int i = 0; i < links.size(); i++) {
                if (links.get(i).from.equals(l.from)
                                && links.get(i).to.equals(l.to)) {
                    links.get(i).skip = false;
                }
            }
        }

        // match() method is used to determine if there is a link
// between a starting
        // node and an ending node

        public boolean match(ch6p2_GraphNode from, ch6p2_GraphNode to) {
            // iterate through list of links
            for (int x = links.size() - 1; x >= 0; x--) {
                if (links.get(x).from.equals(from) && links.get(x).to.equals(to)
                                && !links.get(x).skip) {
                    links.get(x).skip = true;
                    return true;
                }
            }
            return false;
        }// end match

        // find() method is used to
        // find the next link to try exploring

        public ch6p2_Link find(ch6p2_GraphNode from) {

            // iterate through the list of links
            for (int x = 0; x < links.size(); x++) {
                // link found
                if (links.get(x).from.equals(from) && !links.get(x).skip) {
                    ch6p2_Link l = new ch6p2_Link(links.get(x).from,
                                                links.get(x).to);
                                links.get(x).skip = true;
// mark this link as used so we don't
                                                                                                //
match it again
                                return l;
                        }
                }
                return null; // not found
        }// end find()
}// end Graph.java
```

## Summary

In this chapter, you learned the fundamentals of the breadth-first search algorithm and how to apply it to solve the searching program in practice. You also learned how to build problem-solving agents based on the breadth-first search algorithm and the Navigation class that you studied in previous chapters, in which problem-solving agents intelligently find a route path from a starting point to any destination.

In the next chapter, you will get the basic idea behind the heuristic search strategy, learn how to implement a hill-climbing algorithm on leJOS EV3, and then integrate the implemented hill-climbing algorithm in the leJOS-based robotics system for localization and path planning.

CHAPTER 7

# Hill-Climbing Search and Its Implementation with Lego Mindstorms

Just as you were introduced to the depth-first search (DFS) algorithm in Chapter 5 and the breadth-first search algorithm in Chapter 6, in this chapter, you will learn the basic ideas behind heuristic search strategies and how to implement a hill-climbing algorithm, which is one of the most typical heuristic approaches in leJOS EV3. Specifically, this chapter will cover the following topics:

- A new hill-climbing search algorithm, which can be applied to building arbitrary tree structures generically.
- Applying and integrating the proposed hill-climbing algorithm in the leJOS-based robotics system for localization and path planning, which enhances the existing pathfinding approaches within the leJOS system.

## Introduction to Heuristic Search

Problem solving is one of the fundamental issues in many artificial intelligence (AI)-related applications. Typically, there are two problem categories. The first one can be solved by using some sort of deterministic procedure, such as the calculation of an absolute value, such as the distance between two nodes according to their coordinates. It is very easy for computers to solve this kind of problem, as they simply translate it into an equation that it can then execute. The second type of problem, however, does not always lead to a straightforward deterministic solution. This happens a lot in the real world, and many problems can be solved only by searching for a solution using search strategies, such as breadth-first search or depth-first search, which you have seen in Chapters 5 and 6.

As you move forward in this chapter, you will learn about the heuristic search approach. You will learn about some representations and terminology used in AI search strategies through the following example:

*Assume that you lose a book somewhere in the school building. This building is illustrated in Figure 7-1.*

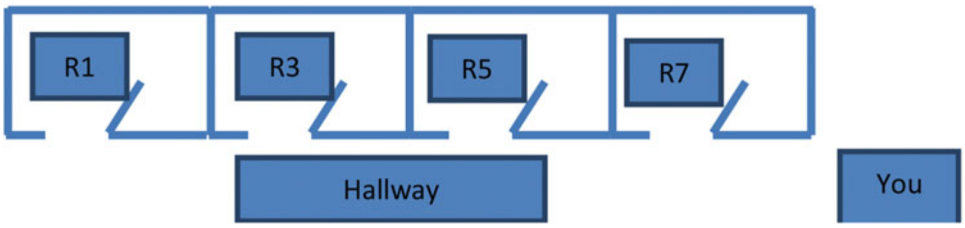


***Figure 7-1.*** *Building topology*

You are standing at the front of the hallway at the "You" block. As you begin your search, you check room R8. Not finding your book there, you return to the hallway and then check room R7. Not finding your book there, you then return to the hallway, and so on. By checking each room and returning back and forth to the hallway, you finally find your book in room R2. Such a scenario can be easily translated into a graphic as shown in Figure 7-2 below:

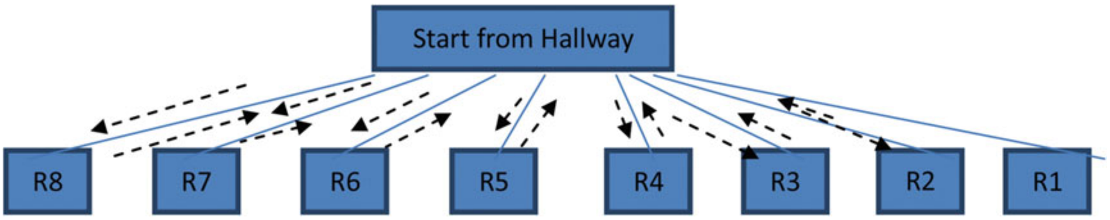


***Figure 7-2.*** *Graphic representation of your search strategy*

As you have seen in Chapters 5 and 6, representing search problems in a graphical format is beneficial, as it provides a convenient way to depict how a solution is found. Throughout this book, when introducing AI search algorithms, you will find that the following terms are used:

*Node* A discrete point: for example, the hallway in Figure 7-2.

*Terminal node* The node that ends a path: for example, room R8 in Figure 7-2.

*Search space* The set of all of the nodes in a graph.

*Goal* The node that the search aims to reach.

*Heuristics* Utility function to determine if any specific node is a better next choice than the another.

*Solution* A set of nodes connected and directed en route to the destination goal.

In the above lost book scenario, each classroom in the building is called a *node*. The entire building, including the eight classrooms, is the *search space*. The *goal*, as it turns out, is room R2. Finally, the *solution* path is shown in Figure 7-2 via dashed arrows. Classrooms R1, R2, R3, R4, R5, R6, R7, and R8 are *terminal nodes* because they lead nowhere. Heuristics are not represented in the graphic. Rather, they are techniques that you might employ to help you better find a path.

Given the above example, you probably think that searching for a solution is not difficult at all; that is, you simply start at the beginning and then search the nodes one by one to reach to your goal. This is true only in an extremely simple case such as the above lost book example, because the search space is so small (that is, there are only eight nodes, and you have to try at most eight times to find the solution). In the real world, however, the search space is usually very large. For example, think about how many cities there are in New Hampshire versus how many cities there are in the entire United States? As the search space grows, so does the number of possible paths to the goal. Moreover, one essential issue when adding another node to the search space is that it often adds much more than one path and thus is not linear. In other words, the number of potential pathways to reach the goal can increase in a nonlinear fashion when the size of the search space grows.

In a nonlinear situation, the number of possible paths can quickly become very large. For instance, consider the number of methods that you can use to arrange three objects A, B, and C. The six total possible permutations are as follows:

| | | |
|---|---|---|
| A | B | C |
| A | C | B |
| B | C | A |
| B | A | C |
| C | B | A |
| C | A | B |

When you add one more object, you will see that the total possible number of permutations is seven, because the number of methods in which N objects can be arranged is equal to N! (N factorial). Therefore, if you have four objects to arrange, there would be 4! permutations (that is, 24 permutations). With five objects, the number becomes 120 permutations, and with six objects it is increased to 720 permutations. With 100 objects the number of possible permutations is huge: 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000. In such a case, an exhaustive search that examines all of the nodes would not work because it consumes too much time and too many computing resources. As a result, AI-based search techniques are essential to search for a solution. The three most fundamental techniques are depth-first search, breadth-first search, and hill- climbing search.

Comparing and evaluating the performance of the various AI-based search techniques can be very complicated. In this book, we are mainly concerned with only two measurements: (1) how quickly a solution is found and (2) how good the solution is. When you try to find a solution with minimum effort, the first measurement – that is, how quickly a solution is found – is especially important, and it is typically affected both by the size of the search space and by the number of nodes actually traversed in the process of finding the solution. This is because backtracking from dead ends is time consuming, and you want a search that seldom retraces its steps. In the other cases, the *quality* of the solution is more important. In AI-based searching, there is a difference between finding the best solution and finding a good solution. Finding the best solution requires an exhaustive search as this is the only way to know that the best solution has been found or that *global optimization* has been achieved. Finding a good solution, on the other hand, means finding a solution that is within a defined set of constraints, but it does not matter if a better solution might exist and that what you achieve could be a *local optimal* solution.

As you saw in Chapter 5, the depth-first approach can find a good solution when the first try is exclusive of backtracking. However, when we reorganize the graphic as illustrated in Figure 5-2 of Chapter 5, finding a solution might involve considerable backtracking. As a result, the outcome of the example introduced in Chapter 5 cannot be generalized. Moreover, the performance of depth-first searches can be quite poor when a particularly long branch with no solution at the end is explored. In such a case, depth-first searches waste time both in exploring this branch and backtracking to the goal.

In the example illustrated in Chapter 6, the breadth-first search performs very well, finding a reasonable solution. Similar to depth-first searches, this solution cannot be generalized because the first path to be found depends on the actual physical organization of the node information. Comparing depth-first search and breadth-first searches, you can find different paths through the same search space. Breadth-first searching works well when the goal is not located too deeply into the search space. It works very poorly, however, when the goal is located several layers deep. In such a cases, a breadth-first search expends substantial effort during the backtracking process.

Based on the above, it is therefore inaccurate to assert that one search method is always superior to the other. In many cases, the manner in which a problem is defined can help you choose the appropriate search method. As you have learned in Chapters 5 and 6, neither the depth-first search approach nor the breadth-first search approach attempts to make any rational guesses about whether one node in the search space is closer to the goal than another. Instead, both search strategies simply move from one node to the next using a prescribed pattern until the goal is finally reached. In order to increase the probability that a search can reach its goal faster, you need to add heuristic capabilities to the search algorithm.

*Heuristics* are simply rules that increase the likelihood that a search moves forward in a correct direction. For example, assume that you are lost on White Mountain in New Hampshire. You still want to climb to the top of mountain, and you know that it very icy and windy at the summit. The woods on White Mountain are so thick that you cannot see far ahead. Furthermore, the trees are too big to climb in order to get a look around. However, you know that the temperature at the summit will be colder than at your current position, so that when you are nearing the summit you will start to “feel” it, as it gets colder and colder.

You begin by moving uphill. You know that you are moving in the correct direction, because you are traveling along a path where you feel colder practically with each step. As you move, you can feel the temperature decrease. Finally, you reach the top of the mountain. In this scenario, the heuristic information used to find the summit does not guarantee success. However, it increases the probability of an early success, leading you to reach the goal quickly.

The basic idea of a heuristic search is that you try to focus on paths that seem to be leading you nearer to your goal rather than trying all possible search paths. Even though you cannot make sure that you are really near your goal, you might be able to make a good guess and heuristics help you in making that guess.

When doing a heuristic search, you need a utility function that scores a node in the search tree according to how close you are to the goal state. Although it represents just a guess, it should still be helpful in maximizing or minimizing some constraint. For example, when you apply GPS to set up a route from a starting place to a destination, there are two possible constraints that you may want to minimize. The first one is the faster time, and the second one is the shortest distance. The shortest route does not necessarily translate to the fastest time, or vice versa. In the example in this chapter, we are focused on the hill-climbing search built on the depth-first search framework, which aims to minimize a constraint called the *number of connections*.

## Overview of Hill-Climbing Search

In hill climbing, the basic idea is always to head toward a state that is better than the current one. Thus if you are at town A and you need to get to town B and town C (with your ultimate goal being town D), then you should make a move if town B or town C appears to be nearer to town D than town A. In the steepest ascent, a hill-climbing search will always make your next state the best successor to your current state, and it will only make a move if that successor is better than your current state. This can be described as follows:

1. Start with current state (initial state).
2. Until current state = goal state, OR there is no change in current state, do the following:
   a. Get the successors of the current state, and use the utility function to assign a score to each successor.
   b. If one of the successors has a better score than the current state, then set the new current state to be the successor with the best score.
   c. Since the algorithm does not attempt to test exhaustively every node and path, you don't have to maintain a node except the current state.
3. Hill climbing terminates when there are no successors of the current state that are better than the current state itself.

Next let's see how to use the hill-climbing search approach to solve a pathfinding problem that could be applied to a GPS system when navigating from a starting city to a destination city. Suppose that you want to drive from city *A* (for instance, New York City, NY) to *E* (for example, Boston, MA) with the minimum number of nodes using the hill-climbing search algorithm, as shown in Figure 7-3. Assume that we use the Cartesian coordinate system as follows:

The coordinate at A is (0,0).

The coordinate at B is (-10,20).

The coordinate at C is (-15,20).

The coordinate at D is (0,20).

The coordinate at E is (0,30).

The coordinate at F is (-15,30).

The coordinate at G is (10,20).

The coordinate at H is (10,10).

Given the following route path, work out a plan for you to start from *A* and reach to *E* using a hill-climbing search strategy.

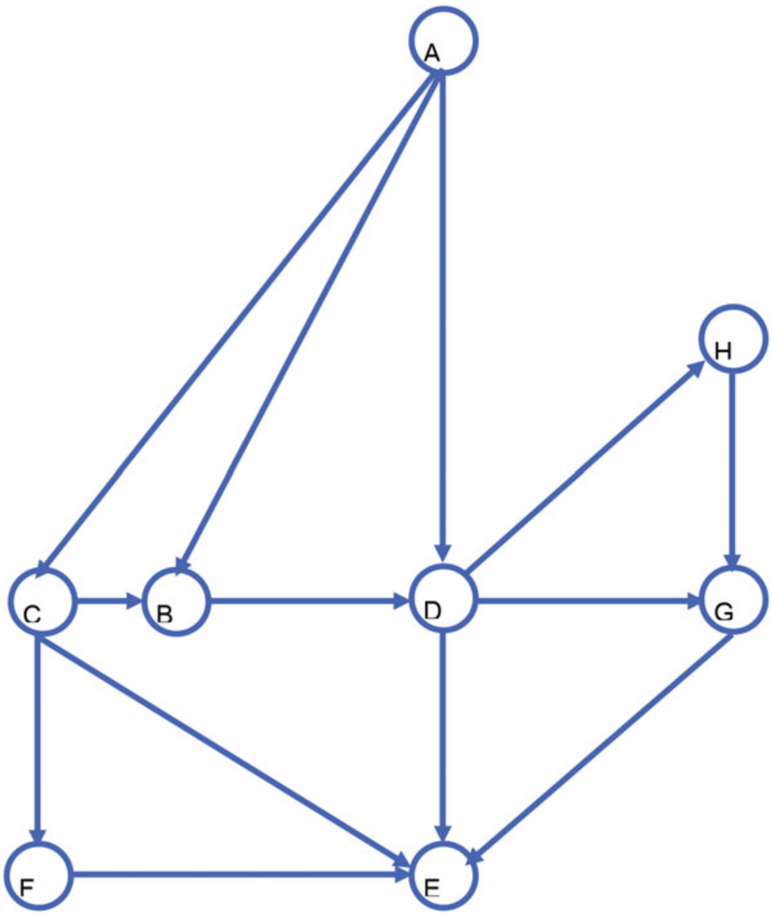


***Figure 7-3.*** *Routes in between two cities*

The following programs can be used to find a path for you to schedule a travel plan automatically by using the hill-climbing search algorithm:

```
//*********************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch7p1_main.java
//Driver class to set up map using ch7p1_GraphNode, ch7p1_Link, and //ch7p1_Graph classes.
//Calls a hill-climbing search in ch7p1_Graph class to create //navigation path from a start
//and end node
//*********************************************************************************

public class ch7p1_main {

        public static void main(String[] args) {

// These objects used to define what your graph looks // like
                ch7p1_Graph searchGraph = new ch7p1_Graph();
                ch7p1_GraphNode A, B, C, D, E, F, G, H;
```

```
ch7p1_Link AB, AC, AD, BD, CB, CE, CF, DE, DH, FE, GE, HG;

// define each node
A = new ch7p1_GraphNode("A", 0, 0);
B = new ch7p1_GraphNode("B", -10, 20);
C = new ch7p1_GraphNode("C", -15, 20);
D = new ch7p1_GraphNode("D", 0, 20);
E = new ch7p1_GraphNode("E", 0, 30);
F = new ch7p1_GraphNode("F", -15, 30);
G = new ch7p1_GraphNode("G", 10, 20);
H = new ch7p1_GraphNode("H", 10, 10);

// define which GraphNodes are connected
AB = new ch7p1_Link(A, B, dist(A.xLocation, A.yLocation, B.xLocation,
                B.yLocation));
AC = new ch7p1_Link(A, C, dist(A.xLocation, A.yLocation, C.xLocation,
                C.yLocation));
AD = new ch7p1_Link(A, D, dist(A.xLocation, A.yLocation, D.xLocation,
                D.yLocation));
BD = new ch7p1_Link(B, D, dist(B.xLocation, B.yLocation, D.xLocation,
                D.yLocation));
CB = new ch7p1_Link(C, B, dist(C.xLocation, C.yLocation, B.xLocation,
                B.yLocation));
CE = new ch7p1_Link(C, E, dist(C.xLocation, C.yLocation, E.xLocation,
                E.yLocation));
CF = new ch7p1_Link(C, F, dist(C.xLocation, C.yLocation, F.xLocation,
                F.yLocation));
DE = new ch7p1_Link(D, E, dist(D.xLocation, D.yLocation, E.xLocation,
                E.yLocation));
DH = new ch7p1_Link(D, H, dist(D.xLocation, D.yLocation, H.xLocation,
                H.yLocation));
FE = new ch7p1_Link(F, E, dist(F.xLocation, F.yLocation, E.xLocation,
                E.yLocation));
GE = new ch7p1_Link(G, E, dist(G.xLocation, G.yLocation, E.xLocation,
                E.yLocation));
HG = new ch7p1_Link(H, G, dist(H.xLocation, H.yLocation, G.xLocation,
                G.yLocation));

// add all nodes and links to your graph object
searchGraph.addNode(A);
searchGraph.addNode(B);
searchGraph.addNode(C);
searchGraph.addNode(D);
searchGraph.addNode(E);
searchGraph.addNode(F);
searchGraph.addNode(G);
searchGraph.addNode(H);

searchGraph.addLink(AB);
searchGraph.addLink(AC);
searchGraph.addLink(AD);
```

```
                searchGraph.addLink(BD);
                searchGraph.addLink(CB);
                searchGraph.addLink(CE);
                searchGraph.addLink(CF);
                searchGraph.addLink(DE);
                searchGraph.addLink(DH);
                searchGraph.addLink(FE);
                searchGraph.addLink(GE);
                searchGraph.addLink(HG);

                // run hill-climbing search to get from start to
                // destination
                searchGraph.hillTraverse(
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(A)),
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(E)));

                // display path created using hillTraverse
                // This will be display the path from start to
                // destination

                System.out
                                .println("the path from your current city to the destination
                                city is: ");
                for (int i = searchGraph.hillPath.size() - 1; i >= 0; i--) {
                        int count = 0;
                        if (i != 0) {
                                for (int j = searchGraph.hillPath.size() - 1; j >= 0; j--) {
                                        if (searchGraph.hillPath.get(i).cityName ==
                                       searchGraph.hillPath
                                                        .get(j).cityName)
                                                count++;

                                }
                                if (count == 1)
                                        System.out.print(searchGraph.hillPath.get(i).
                                        cityName
                                                        + "->");
                        } else
                                System.out.print(searchGraph.hillPath.get(i).cityName);
                }

        }

        static int dist(int x1, int y1, int x2, int y2) {
                int distance = 0;

                distance = (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1);

                return distance;
        }
}
```

```
//******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch7p1_Link.java
//Represents a link between two GraphNodes
//******************************************************************

public class ch7p1_Link {

        ch7p1_GraphNode from;
        ch7p1_GraphNode to;

        int distance;

// boolean skip is used for traversal to determine if the path // has already
        // been visited or not
        boolean skip;

        public ch7p1_Link(ch7p1_GraphNode from, ch7p1_GraphNode to, int d) {
                this.from = from;
                this.to = to;
                distance = d;
                skip = false;
        }
}
//******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3
//        ch7p1_GraphNode.java
//Represents name and coordinates of a node on a graph
//******************************************************************

public class ch7p1_GraphNode {
        String cityName;
        int xLocation, yLocation;

        public ch7p1_GraphNode(String cityName, int xLocation, int yLocation) {
                this.cityName = cityName;
                this.xLocation = xLocation;
                this.yLocation = yLocation;

        }

        public String toString() {
                return cityName + " ("+xLocation +"," + yLocation + ")";
        }
}
//**********************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch7p1_Graph.java
//Represents GraphNodes connected through Links, including method //for doing a hill
climbing
//search traversal of the graph.
//the method hillTraverse creates a hillPath list which the robot //will follow to
demonstrate
//how the hill-climbing search works.
//**********************************************************************************************
```

```
import java.util.ArrayList;
import java.util.Stack;

public class ch7p1_Graph {

        // nodes and links define the physical creation of your Graph
        ArrayList<ch7p1_GraphNode> nodes;
        ArrayList<ch7p1_Link> links;

        // Two lists used for traversing
        ArrayList<ch7p1_GraphNode> hillTraverse;
        Stack<ch7p1_Link> travelStack = new Stack<ch7p1_Link>();

        // List used to define where you would like to move
        ArrayList<ch7p1_GraphNode> hillPath = new ArrayList<ch7p1_GraphNode>();

        // Constructor of ch7p1_Graph class
        public ch7p1_Graph() {
                nodes = new ArrayList<ch7p1_GraphNode>();
                links = new ArrayList<ch7p1_Link>();
                hillTraverse = new ArrayList<ch7p1_GraphNode>();
        }// end constructor

        // addNode()
        // Add a city node to the graph

        public void addNode(ch7p1_GraphNode node) {
                nodes.add(node);
        }// end addNode

        // addLink
        // Add link to the graph

        public void addLink(ch7p1_Link link) {
                links.add(link);
        }// end addLink

        // hillTraverse()
        // perform hill-climbing search on the graph

        public void hillTraverse(ch7p1_GraphNode from, ch7p1_GraphNode to) {
                // boolean matched;
                int distance;
                ch7p1_Link found;

                // determine if there is a link between from and to
                // if there is a match then add the link to the
// travelStack and
                // add the nodes to hillPath
                // This will ultimately repeated by the end of the
// search
```

```
                distance = match(from, to);
                if (distance != 0) {
                        travelStack.push(new ch7p1_Link(from, to, distance));
                        hillPath.add(new ch7p1_GraphNode(to.cityName, to.xLocation,
                                        to.yLocation));
                        hillPath.add(new ch7p1_GraphNode(from.cityName, from.xLocation,
                                        from.yLocation));
                        return;
                }

                // if there is no match found you could another path
// findings
                found = find(from);

// if you find a new connection then you could add it // to the travelStack
                // and
                // and the start node to hillPath
// recursively call hillTraverse with the link's to as // start and our
                // destination as the end

                if (found != null) {
                        travelStack.push(new ch7p1_Link(from, to, found.distance));
                        hillTraverse(found.to, to);
                        hillPath.add(new ch7p1_GraphNode(from.cityName, from.xLocation,
                                        from.yLocation));
                }

                // backtrack if you cannot find a new connection
                else if (travelStack.size() > 0) {
                        found = travelStack.pop();
                        hillTraverse(found.from, found.to);
                        hillPath.add(new ch7p1_GraphNode(from.cityName, from.xLocation,
                                        from.yLocation));
                }
        }// end hillTraverse()

        // find() method is used to
        // find the next link to try exploring

        public ch7p1_Link find(ch7p1_GraphNode from) {

                int pos = -1;
                int dist = 0;

                // iterate through the list of links
                for (int a = 0; a < links.size(); a++) {

                        if (links.get(a).from.equals(from) && !links.get(a).skip) {

                                // Use the longest flight.
                                if (links.get(a).distance > dist) {
                                        pos = a;
                                        dist = links.get(a).distance;
                                }
                        }
                }

                // link found
                if (pos != -1) {

                        // mark this link as used so we don't match it
// again
                        links.get(pos).skip = true;

                        ch7p1_Link foundList = new ch7p1_Link(links.get(pos).from,
                                        links.get(pos).to, links.get(pos).distance);
                        return foundList;
                }

                return null; // not found
        }// end find()

        // match() method is used to determine if there is a link
// between a starting
        // node and an ending node

        public int match(ch7p1_GraphNode from, ch7p1_GraphNode to) {

                // iterate through list of links
                for (int a = links.size() - 1; a >= 0; a--) {
                        if (links.get(a).from.equals(from) && links.get(a).to.equals(to)
                                        && !links.get(a).skip) {
                                links.get(a).skip = true;
                                return links.get(a).distance;
                        }
                }
                return 0;
        }// end match()
}// end Graph.java
```

Given `startCity` A and `endCity` E, the result of running the programs is as follows:

```
the path from your current city to the destination city is:
A->C->E
```

This result shows you the exact path for traversing a graphic using a hill-climbing search strategy without including the intermediate trace-back nodes.

## leJOS EV3-Based Hill-Climbing Algorithm

In this section, you need to develop a program for your robot so that it can travel the path between starting node A and destination node E, as shown in Figure 7-3, with the minimum number of nodes and using the hill-climbing search algorithm.

You program should at minimum show the destination's coordinate on the LCD and then display a message "`Press ENTER key to continue`." When the Enter key is pressed, your robot should move to the next node. For instance, suppose that your robot starts from `A (0,0)` and wants to explore a path to `E (0,30)`. Assume that the path your robot explores with the minimum number of nodes passed using hill-climbing search is `A -> C -> E`. At starting point A, your program should do the following:

Display destination's coordinate `C(-15, 20)` on the LCD.

Display the message "`Press ENTER key to continue`".

Go to the location with coordinate -15,20.

Display the destination's coordinate `E(0, 30)` on the LCD.

Display the message "`Press ENTER key to continue.`"

Go to the location with coordinate 0,30.

Moreover, your problem should have a string called *destination,* so that it is intelligent enough when changing the value of the destination that your robot can explore a new path from starting node A to the new destination node, the robot will pass the minimum number of nodes to reach destination node. (We assume that the starting node is always A, so the from string can be hard-coded.)

The following programs represent the implementation of the leJOS-based hill-climbing algorithm designed to explore a path from node A to the destination node E:

```
//******************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch7p2_main.java
//Driver class to set up map using ch7p2_GraphNode, ch7p2_Link, and //ch7p2_Graph classes.
//Calls a hill-climbing search in ch7p2_Graph class to create //navigation path from a start
//and end node and then robots will follow the path to move from //start node
//to destination node
//******************************************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.MovePilot;
import lejos.robotics.navigation.Navigator;

public class ch7p2_main {

        static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(
                        MotorPort.A);
        static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(
                MotorPort.C);

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();

// setup the wheel diameter of left (and right) motor // in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the center // of wheel to
                // the center of robot, i.e. half of track width
                Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
                Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

                // set up the chassis type, i.e. Differential pilot
                Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                                WheeledChassis.TYPE_DIFFERENTIAL);
                MovePilot ev3robot = new MovePilot(chassis);

                Navigator navbot = new Navigator(ev3robot);

// These objects used to define what your graph looks // like
                ch7p2_Graph searchGraph = new ch7p2_Graph();
                ch7p2_GraphNode A, B, C, D, E, F, G, H;
                ch7p2_Link AB, AC, AD, BD, CB, CE, CF, DE, DH, FE, GE, HG;

                // define each node
                A = new ch7p2_GraphNode("A", 0, 0);
                B = new ch7p2_GraphNode("B", -10, 20);
                C = new ch7p2_GraphNode("C", -15, 20);
                D = new ch7p2_GraphNode("D", 0, 20);
                E = new ch7p2_GraphNode("E", 0, 30);
                F = new ch7p2_GraphNode("F", -15, 30);
                G = new ch7p2_GraphNode("G", 10, 20);
                H = new ch7p2_GraphNode("H", 10, 10);

                // define which GraphNodes are connected
                AB = new ch7p2_Link(A, B, dist(A.xLocation, A.yLocation, B.xLocation,
                                B.yLocation));
                AC = new ch7p2_Link(A, C, dist(A.xLocation, A.yLocation, C.xLocation,
                                C.yLocation));
                AD = new ch7p2_Link(A, D, dist(A.xLocation, A.yLocation, D.xLocation,
                                D.yLocation));
                BD = new ch7p2_Link(B, D, dist(B.xLocation, B.yLocation, D.xLocation,
                                D.yLocation));
                CB = new ch7p2_Link(C, B, dist(C.xLocation, C.yLocation, B.xLocation,
                                B.yLocation));
                CE = new ch7p2_Link(C, E, dist(C.xLocation, C.yLocation, E.xLocation,
                                E.yLocation));
                CF = new ch7p2_Link(C, F, dist(C.xLocation, C.yLocation, F.xLocation,
                                F.yLocation));
                DE = new ch7p2_Link(D, E, dist(D.xLocation, D.yLocation, E.xLocation,
                                E.yLocation));
                DH = new ch7p2_Link(D, H, dist(D.xLocation, D.yLocation, H.xLocation,
                                H.yLocation));
                FE = new ch7p2_Link(F, E, dist(F.xLocation, F.yLocation, E.xLocation,
                                E.yLocation));
                GE = new ch7p2_Link(G, E, dist(G.xLocation, G.yLocation, E.xLocation,
                                E.yLocation));
                HG = new ch7p2_Link(H, G, dist(H.xLocation, H.yLocation, G.xLocation,
                                G.yLocation));

                // add all nodes and links to your graph object
                searchGraph.addNode(A);
                searchGraph.addNode(B);
                searchGraph.addNode(C);
                searchGraph.addNode(D);
                searchGraph.addNode(E);
                searchGraph.addNode(F);
                searchGraph.addNode(G);
                searchGraph.addNode(H);

                searchGraph.addLink(AB);
                searchGraph.addLink(AC);
                searchGraph.addLink(AD);
                searchGraph.addLink(BD);
                searchGraph.addLink(CB);
                searchGraph.addLink(CE);
                searchGraph.addLink(CF);
                searchGraph.addLink(DE);
                searchGraph.addLink(DH);
                searchGraph.addLink(FE);
                searchGraph.addLink(GE);
                searchGraph.addLink(HG);

                // run hill-climbing search to get from start to
// destination
                searchGraph.hillTraverse(
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(A)),
                                searchGraph.nodes.get(searchGraph.nodes.indexOf(E)));

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

// Robot moves through path from start to destination // by using
                // hillTraverse

                for (int i = searchGraph.hillPath.size() - 1; i >= 0; i--) {
                        int count = 0;
                        for (int j = searchGraph.hillPath.size() - 1; j >= 0; j--) {
                                if (searchGraph.hillPath.get(i).cityName == searchGraph.
                                hillPath
                                                .get(j).cityName)
                                        count++;

                        }
                        if (count == 1) {
                                // go to node
                                navbot.goTo(searchGraph.hillPath.get(i).xLocation,
                                                searchGraph.hillPath.get(i).yLocation);

                                LCD.clear();

                                // display current location
                                LCD.drawString("At location "
                                                + searchGraph.hillPath.get(i).cityName + ",
                                                  ", 0, 0);

                                LCD.drawString(searchGraph.hillPath.get(i).xLocation + ", "
                                                + searchGraph.hillPath.get(i).yLocation, 0,
                                                  1);

                                LCD.drawString("Press ENTER key", 0, 2);

                                // block the thread until a button is
// pressed
                                buttons.waitForAnyPress();

                        }

                }

        }

        static int dist(int x1, int y1, int x2, int y2) {
                int distance = 0;

                distance = (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1);

                return distance;
        }
}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch7p2_Link.java
//Represents a link between two GraphNodes);
//*******************************************************************
```

```
public class ch7p2_Link {

        ch7p2_GraphNode from;
        ch7p2_GraphNode to;

        int distance;

// boolean skip is used for traversal to determine if the path // has already
        // been visited or not
        boolean skip;

        public ch7p2_Link(ch7p2_GraphNode from, ch7p2_GraphNode to, int d) {
                this.from = from;
                this.to = to;
                distance = d;
                skip = false;
        }
}
//*************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 //ch7p1_GraphNode.java
//Represents name and coordinates of a node on a graph
//*************************************************************************

public class ch7p2_GraphNode {
        String cityName;
        int xLocation, yLocation;

        public ch7p2_GraphNode(String cityName, int xLocation, int yLocation) {
                this.cityName = cityName;
                this.xLocation = xLocation;
                this.yLocation = yLocation;

        }

        public String toString() {
                return cityName + " (" + xLocation + "," + yLocation + ")";
        }
}
//*********************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch7p2_Graph.java
//Represents GraphNodes connected through Links, including method //for doing a hill
climbing
//search traversal of the graph.
//the method hillTraverse creates a hillPath list which the robot //will follow to
demonstrate
//how the hill-climbing search works.
//*********************************************************************************************

import java.util.ArrayList;);
import java.util.Stack;
```

```
public class ch7p2_Graph {

        // nodes and links define the physical creation of your Graph
        ArrayList<ch7p2_GraphNode> nodes;
        ArrayList<ch7p2_Link> links;

        // Two lists used for traversing
        ArrayList<ch7p2_GraphNode> hillTraverse;
        Stack<ch7p2_Link> travelStack = new Stack<ch7p2_Link>();

        // List used to define where you would like to move
        ArrayList<ch7p2_GraphNode> hillPath = new ArrayList<ch7p2_GraphNode>();

        // Constructor of ch7p2_Graph class
        public ch7p2_Graph() {
                nodes = new ArrayList<ch7p2_GraphNode>();
                links = new ArrayList<ch7p2_Link>();
                hillTraverse = new ArrayList<ch7p2_GraphNode>();
        }// end constructor

        // addNode()
        // Add a city node to the graph

        public void addNode(ch7p2_GraphNode node) {
                nodes.add(node);
        }// end addNode

        // addLink
        // Add link to the graph

        public void addLink(ch7p2_Link link) {
                links.add(link);
        }// end addLink

        // hillTraverse()
        // perform hill-climbing search on the graph

        public void hillTraverse(ch7p2_GraphNode from, ch7p2_GraphNode to) {
                // boolean matched;
                int distance;
                ch7p2_Link found;

                // determine if there is a link between from and to
                // if there is a match then add the link to the
// travelStack and
                // add the nodes to hillPath
                // This will ultimately repeated by the end of the
// search

                distance = match(from, to);
                if (distance != 0) {
                        travelStack.push(new ch7p2_Link(from, to, distance));
                        hillPath.add(new ch7p2_GraphNode(to.cityName, to.xLocation,
                                    to.yLocation));
                        hillPath.add(new ch7p2_GraphNode(from.cityName, from.xLocation,
                                    from.yLocation));
                        return;
                }

                // if there is no match found you could another path
// findings
                found = find(from);

// if you find a new connection then you could add it // to the travelStack
                // and
                // and the start node to hillPath
// recursively call hillTraverse with the link's to as // start and our
                // destination as the end

                if (found != null) {
                        travelStack.push(new ch7p2_Link(from, to, found.distance));
                        hillTraverse(found.to, to);
                        hillPath.add(new ch7p2_GraphNode(from.cityName, from.xLocation,
                                    from.yLocation));
                }

                // backtrack if you cannot find a new connection
                else if (travelStack.size() > 0) {
                        found = travelStack.pop();
                        hillTraverse(found.from, found.to);
                        hillPath.add(new ch7p2_GraphNode(from.cityName, from.xLocation,
                                    from.yLocation));
                }
        }// end hillTraverse()

        // find() method is used to
        // find the next link to try exploring);

        public ch7p2_Link find(ch7p2_GraphNode from) {

                int pos = -1;
                int dist = 0;

                // iterate through the list of links
                for (int a = 0; a < links.size(); a++) {

                        if (links.get(a).from.equals(from) && !links.get(a).skip) {

                                // Use the longest flight.
                                if (links.get(a).distance > dist) {
                                        pos = a;
                                        dist = links.get(a).distance;
                                }
                        }
                }

                // link found
                if (pos != -1) {

                        // mark this link as used so we don't match it
// again
                        links.get(pos).skip = true;

                        ch7p2_Link foundList = new ch7p2_Link(links.get(pos).from,
                                        links.get(pos).to, links.get(pos).distance);
                        return foundList;
                }

                return null; // not found
        }// end find()

        // match() method is used to determine if there is a link
// between a starting
        // node and an ending node

        public int match(ch7p2_GraphNode from, ch7p2_GraphNode to) {

                // iterate through list of links
                for (int a = links.size() - 1; a >= 0; a--) {
                        if (links.get(a).from.equals(from) && links.get(a).to.equals(to)
                                        && !links.get(a).skip) {
                                links.get(a).skip = true;
                                return links.get(a).distance;
                        }
                }
                return 0;
        }// end match()
}// end Graph.java);
```

## Summary

In this chapter, you learned about fundamentals of heuristic searching and the hill-climbing algorithm. You now know how to apply the hill-climbing search algorithm to solve a searching program in practice. You also learned how to build problem-solving agents based on the hill-climbing search algorithm and the Navigation class that you studied in previous chapters in which such agents found a route path intelligently from the starting point to any destination using the shortest number of connections.

In the next chapter, you will learn Dijkstra's algorithm, which can be used to find the optimal solution rather than a (or “any”) solution, as defined by the heuristic function in hill climbing. Also, you will learn how to integrate Dijkstra's algorithm in the leJOS-based robotics system for localization and path planning.

CHAPTER 8

# Dijkstra's Algorithm and Its Implementation with Lego Mindstorms

Following up on Chapter 7's discussion of heuristic search algorithms, this chapter will address Dijkstra's algorithm. This algorithm can be used to find the optimal solution, rather than a "or any" solution, as defined by the heuristic function in the hill-climbing algorithm. In particular, this chapter will cover the following topics:

- A new Dijkstra's search algorithm, which can be applied to build arbitrary tree structures generically.
- Applying and integrating the proposed Dijkstra's algorithm in the leJOS-based robotics system for localization and path planning, which enhances the existing pathfinding approaches within the leJOS system.

## Introduction to Dijkstra's Algorithm

The hill-climbing algorithm, discussed in Chapter 7, attempts to minimize the number of connections. As you learned, such a hill-climbing search found a good route — not the best one, but one that was acceptable. Moreover, all of the previous search algorithms introduced so far (including BSF and DFS) are able to determine a solution: that is, any solution, not necessarily the best solution or even a "good" one. Defining heuristics helped to improve the likelihood of finding a good solution; however, no attempt was made to ensure that an optimal solution was found. In order to achieve an optimal solution: that is, the *best* solution, the well-known Dijkstra's algorithm needs to be applied:

***In any given graph and at any starting node, Dijkstra's algorithm discovers the shortest path from the starting node to all other nodes.***

Figure 8-1 displays a graphic representing connected nodes: nodes are blue circles labeled A-J. A path is a blue line connecting two nodes, and each path has an associated distance beside it. Note: the lengths are not meant to be to scale.

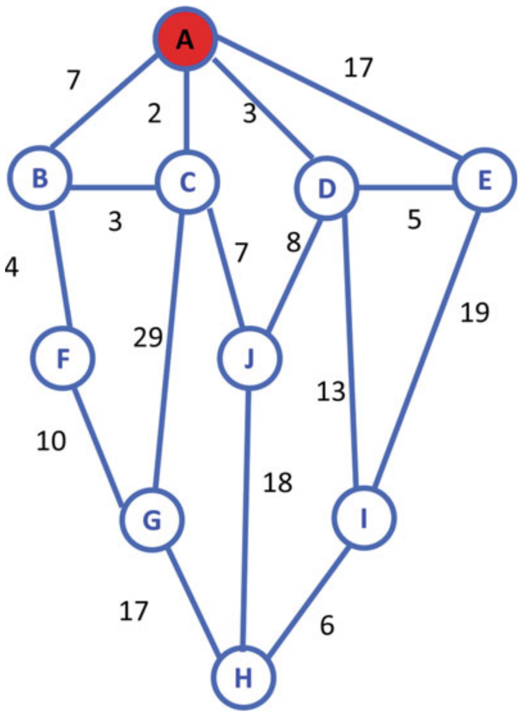


*Figure 8-1.* *Graphic representation of connected nodes*

Node A is your starting node, and you want to find the shortest path to all of the other nodes in the graphic. To do this, you generate a table. This table contains the distance to all nodes in the graphic from the perspective of the starting node A.

As seen in Table 8-1, the initial entries for distances are all set to infinity (or some notional maximum value). This ensures that any path found will be shorter than the initial value stored in the table.

*Table 8-1.* *Initial entries for distances to nodes from Node A*

| Node | Distance to Node from Node A |
|---|---|
| B | INFINTE |
| C | INFINTE |
| D | INFINTE |
| E | INFINTE |
| F | INFINTE |
| G | INFINTE |
| H | INFINTE |
| I | INFINTE |
| J | INFINTE |

Node A is the starting node, and as such you will first examine all of the possible paths away from this node. The options are shown in Table 8-2.

***Table 8-2.*** *Distance to nodes (from Node A) accessible from Node A*

| Node | Distance to Node from Node A |
|---|---|
| B | 7 |
| C | 2 |
| D | 3 |
| E | 17 |

These values are used to update Table 8-1, and thus you now have Table 8-3.

***Table 8-3.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A |
|---|---|
| B | 7 |
| C | 2 |
| D | 3 |
| E | 17 |
| F | INFINTE |
| G | INFINTE |
| H | INFINTE |
| I | INFINTE |
| J | INFINTE |

Figure 8-2 shows the routes marked in red. You have four paths from Node A. However, these paths are not yet guaranteed to be the shortest path. To be sure that you have determined the shortest path, you have to keep going.

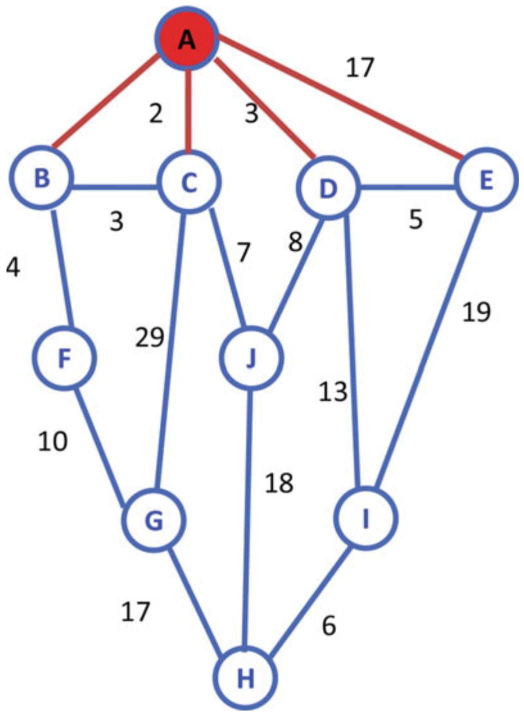


***Figure 8-2.** Graphic representation in which starting node A has been visited*

The next move in the algorithm is to go to the nearest node from Node A. In this case, that is Node C. At Node C, you have paths available to Nodes B, G, and J. When calculating the distances, you must determine them from Node A. The new distances are shown in Table 8-4.

***Table 8-4.** Distance to nodes accessible from Node A*

| Node | Distance to Node from Node A |
|---|---|
| B | 5 |
| G | 31 |
| J | 9 |

These values are then compared to the values that appear in Table 8-3. You will observe that both of these values are less than the current values stored in the table. As such, Table 8-3 transforms to Table 8-5 as follows.

***Table 8-5.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A |
|---|---|
| B | 5 |
| C | 2 |
| D | 3 |
| E | 17 |
| F | INFINTE |
| G | 31 |
| H | INFINTE |
| I | INFINTE |
| J | 9 |

This step illustrates one of the advantages of Dijkstra's algorithm: the route to Node B is not the most direct route, but it is the shortest one. Dijkstra's algorithm can find the shortest route, even when that route is not the most direct one.

Again, all paths accessible from Node C have been checked, and the table of the paths has been updated. Node C is marked as having been visited, as shown in Figure 8-3.

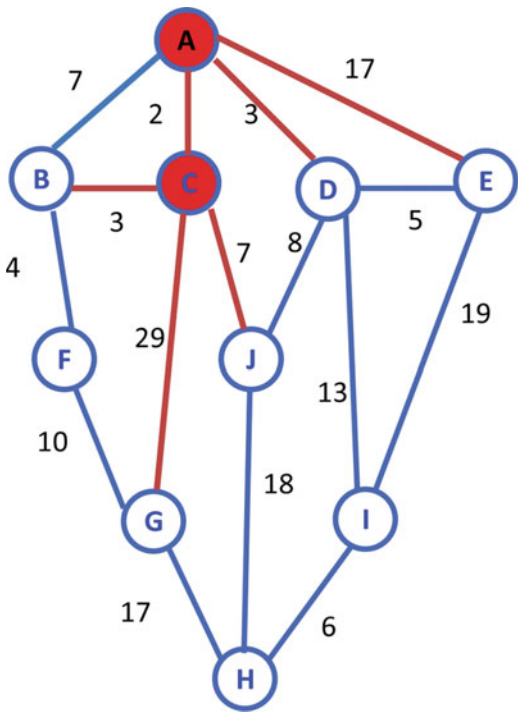


***Figure 8-3.*** *Graphic representation in which Node C has been visited*

In Dijkstra's algorithm, a visited node is never revisited. Furthermore, once a node has been marked as visited, the path to that node is known to be the shortest route from the initial node.

As such, you should add another column to your table, as shown in Table 8-6.

***Table 8-6.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A | Visited |
|---|---|---|
| B | 5 | No |
| C | 2 | Yes |
| D | 3 | No |
| E | 17 | No |
| F | INFINTE | No |
| G | 31 | No |
| H | INFINTE | No |
| I | INFINTE | No |
| J | 9 | No |

As these values are being updated, the route that accompanies these distances also needs to be stored. Once again, the table of paths is checked, and the shortest path to a node that has not been visited is found. This node becomes the next current node. In this case, that is Node D. The following paths are available from Node D:

***Table 8-7.*** *Distance to nodes accessible from Node A*

| Node | Distance to Node from Node A |
|---|---|
| J | 11 |
| I | 16 |
| E | 8 |

The table of all paths is updated to reflect this, and node D is marked as visited. This locks in the shortest path to Node D, also.

As can be seen in Table 8-8, the next-nearest node to Node A is Node B. All paths from Node B are examined next. In this instance, you have a path to a node that is marked as visited, Node C. We already know that the path to Node C is as short as it can get.

***Table 8-8.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A | Visited |
|---|---|---|
| B | 5 | No |
| C | 2 | Yes |
| D | 3 | Yes |
| E | 8 | No |
| F | INFINTE | No |
| G | 31 | No |
| H | INFINTE | No |
| I | 16 | No |
| J | 9 | No |

As the Figure 8-4 shows, when you check the path, the only other node accessible from Node B is Node F. This updates our paths as shown in Table 8-9:

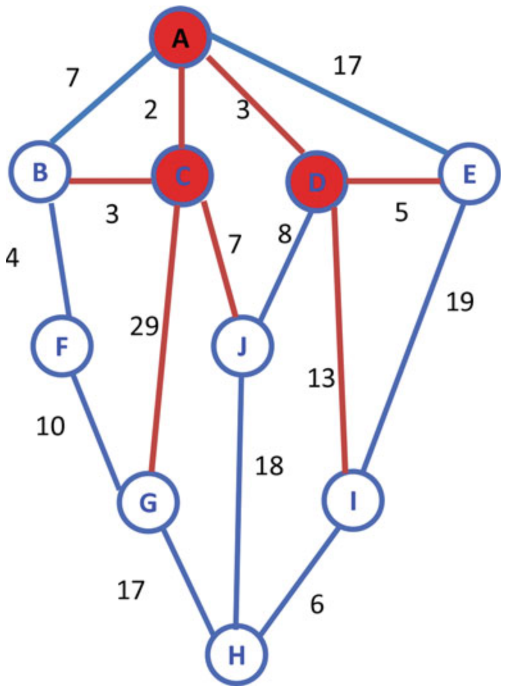


***Figure 8-4.*** *Graphic representation with Node C and Node D marked visited*

***Table 8-9.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A | Visited |
|---|---|---|
| B | 5 | Yes |
| C | 2 | Yes |
| D | 3 | Yes |
| E | 8 | No |
| F | 9 | No |
| G | 31 | No |
| H | INFINTE | No |
| I | 16 | No |
| J | 9 | No |

Table 8-9 again tells us that the next node for you to visit is Node E. You then add up the paths and mark the nodes as visited. Next you check the path, and the only other node accessible from Node E is Node I. The distance from Node A to Node I is 27, which is larger than the one that is shown in the table. Therefore, there is no update for the table in this round, and Node E is marked visited. Continuing with Table 8-9, you know that the next node to visit is Node F. As Figure 8-4 shows, when you check the path, the only other node accessible from Node F is Node G. This updates the paths as shown in Table 8-10.

***Table 8-10.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A | Visited |
|---|---|---|
| B | 5 | Yes |
| C | 2 | Yes |
| D | 3 | Yes |
| E | 8 | Yes |
| F | 9 | Yes |
| G | 19 | No |
| H | INFINTE | No |
| I | 16 | No |
| J | 9 | No |

Table 8-10 tells us that the next node for you to visit is Node J. You then add up the paths and mark the nodes as visited. Next you check the path, and the only other node accessible from Node J is Node H. This updates the paths as shown in Table 8-11.

***Table 8-11.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A | Visited |
|---|---|---|
| B | 5 | Yes |
| C | 2 | Yes |
| D | 3 | Yes |
| E | 8 | Yes |
| F | 9 | Yes |
| G | 19 | No |
| H | 27 | No |
| I | 16 | No |
| J | 9 | Yes |

Table 8-11 tells us that the next node for you to visit is Node I. Next you add up the paths and mark the nodes as visited. When you check the path, the only other node accessible from Node I is Node H. This updates the paths as shown in Table 8-12.

***Table 8-12.*** *Entries for distances to nodes from Node A*

| Node | Distance to Node from Node A | Visited |
|---|---|---|
| B | 5 | Yes |
| C | 2 | Yes |
| D | 3 | Yes |
| E | 8 | Yes |
| F | 9 | Yes |
| G | 19 | No |
| H | 22 | No |
| I | 16 | Yes |
| J | 9 | Yes |

Table 8-12 tells us that the next node to visit is Node G. You then add up the paths and mark the nodes as visited. When you check the path, the only other node accessible from Node G is Node H. The distance from Node A to Node H is 36, which is larger than the 22 shown in the table. Therefore, there is no update to the table and Node G is marked visited. Now all of the nodes have been visited, and the search will be terminated. Eventually, you will see Table 8-13, which shows the shortest distance from Node A to all other nodes in the graphic shown in Figure 8-1.

***Table 8-13.*** *Entries for distances to nodes from Node A*

| Nod | Distance to Node from Node A | Visited |
|---|---|---|
| B | 5 | Yes |
| C | 2 | Yes |
| D | 3 | Yes |
| E | 8 | Yes |
| F | 9 | Yes |
| G | 19 | Yes |
| H | 22 | Yes |
| I | 16 | Yes |
| J | 9 | Yes |

Using the graphic shown in Figure 8-1, the following programs use Dijkstra's algorithm to find the shortest path from starting Node A to all of the other nodes.

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3ch8p1_main.java
//Driver class to set up map using ch8p1_edge, find the shortest //path from
//starting node A to all the other nodes in a given graph.
//*******************************************************************

import java.util.ArrayList;

public class ch8p1_main {
        static ArrayList<ch8p1_edge> graph = null;
        static ch8p1_edge[] parents = null;

        static ArrayList<String> unsolvedConn = null;
        static ArrayList<String> solvedConn = null;
        private static ch8p1_minpath finalPath;

        public static void main(String[] args) {

                // initialize the nodes set
                String[] nodes = { "A", "B", "C", "D", "E", "F", "G", "H", "I", "J" };

                // initialize the map with the nodes
                graph = new ArrayList<ch8p1_edge>();

                graph.add(new ch8p1_edge("A", "B", 7));
                graph.add(new ch8p1_edge("A", "C", 2));
                graph.add(new ch8p1_edge("A", "D", 3));
                graph.add(new ch8p1_edge("A", "E", 17));
                graph.add(new ch8p1_edge("B", "F", 4));
                graph.add(new ch8p1_edge("C", "B", 3));
                graph.add(new ch8p1_edge("C", "G", 29));
                graph.add(new ch8p1_edge("C", "J", 7));
                graph.add(new ch8p1_edge("D", "J", 8));
                graph.add(new ch8p1_edge("D", "I", 13));
                graph.add(new ch8p1_edge("D", "E", 5));
                graph.add(new ch8p1_edge("E", "I", 19));
                graph.add(new ch8p1_edge("F", "G", 10));
                graph.add(new ch8p1_edge("G", "H", 17));
                graph.add(new ch8p1_edge("J", "H", 18));
                graph.add(new ch8p1_edge("I", "H", 6));

                // initialize the unsolved nodes
                unsolvedConn = new ArrayList<String>();

                // sets the parent node in the unsolved connection
// ArrayList to A
                unsolvedConn.add(nodes[0]);

                // initialize the solved nodes
                solvedConn = new ArrayList<String>();

                // Add all nodes to the solved connection tree
                for (int i = 1; i < nodes.length; i++)
                        solvedConn.add(nodes[i]);

                // create a parent array that will store all the edges
                parents = new ch8p1_edge[nodes.length];

// Set the initial node to A and make its parent null // with a weight cost
                // of zero
                parents[0] = new ch8p1_edge(null, nodes[0], 0);

                for (int i = 0; i < solvedConn.size(); i++) {
// get all of the String node names that could be // attached the
                        // root
                        String n = solvedConn.get(i);

                        // Check the weights of all the nodes that are
// attached to the root
                        // A node
                        // If they are attached will return positive
// weight if not will
                        // return -1
                        parents[i + 1] = new ch8p1_edge(nodes[0], n, getEdgeLength(
                                        nodes[0], n));
                }

                finalPath = null;
                // while the solved nodes ArrayList is greater than
// zero
                while (solvedConn.size() > 0) {
// Create a minimum shortest path object to find // the shortest path
                        // to all connected points
                ch8p1_minpath msp = getMinSideNode();
                finalPath = msp;

                if (msp.getEdgeLength() == -1)
                        msp.outputPath(nodes[0]);
                else
                        msp.outputPath();

                String node = msp.getLastNode();
                unsolvedConn.add(node);
                setEdgeLength(node);
        }
}

public static String getParent(ch8p1_edge[] parents, String node) {
        if (parents != null) {
                for (ch8p1_edge nd : parents) {
                        if (nd.getChildNode() == node) {
                                return nd.getParentNode();
                        }
                }
        }
        return null;
}

public static void setEdgeLength(String parentNode) {
        if (graph != null && parents != null && solvedConn != null) {
                for (String node : solvedConn) {
                        ch8p1_minpath msp = getMinPath(node);
                        int w1 = msp.getEdgeLength();
                        if (w1 == -1)
                                continue;
                        for (ch8p1_edge n : parents) {
                                if (n.getChildNode() == node) {
                                        if (n.getEdgeLength() == -1 ||
                                        n.getEdgeLength() > w1) {
                                                n.setEdgeLength(w1);
                                                n.setParentNode(parentNode);
                                                break;
                                        }
                                }
                        }
                }
        }
}

public static int getEdgeLength(String parentNode, String childNode) {
        if (graph != null) {
                for (ch8p1_edge s : graph) {
                        if (s.getParentNode() == parentNode
                                        && s.getChildNode() == childNode)
                                        return s.getEdgeLength();
                        }
                }
                return -1;
        }

        public static ch8p1_minpath getMinSideNode() {
                // Create a minimum shortest path object
                ch8p1_minpath minMsp = null;
                // while the solved node ArrayList is greater than zero
                if (solvedConn.size() > 0) {
                        // Create an index value set to zero
                        int index = 0;
                        // for each value in the solved nodes ArrayList
                        for (int j = 0; j < solvedConn.size(); j++) {
                                // Create a shortest path map get the
// MinPath of the solved node
                                ch8p1_minpath msp = getMinPath(solvedConn.get(j));
// if there is no minimum shortest path, // if the minimum shortest
                                // path
                                // does not equal -1
                                // AND the minimum shortest path get
// weight is less than the
                                // minimum shortest path get weight
                                if (minMsp == null || msp.getEdgeLength() != -1
                                                && msp.getEdgeLength() < minMsp.
                                                getEdgeLength()) {
// set the minimum shortest path to // the minimum shortest
                                        // path
                                        minMsp = msp;
                // set the index value equal it the // node value j
                                        index = j;
                                }
                        }
                        // remove the index that you checked in the
// solved nodes
                        solvedConn.remove(index);

                }
                // return the MinShortPath object
                return minMsp;
        }

        public static ch8p1_minpath getMinPath(String node) {
// Create a Minshort Path object that is an ArrayList // and set the take
                // in node as the base
                // in this case will always be zero
                ch8p1_minpath msp = new ch8p1_minpath(node);
                // if the parents array does not equal null and the
// unsolved nodes does
                // not equal null
                if (parents != null && unsolvedConn != null) {
                        for (int i = 0; i < unsolvedConn.size(); i++) {
                                ch8p1_minpath tempMsp = new ch8p1_minpath(node);
                                String parent = unsolvedConn.get(i);
                                String curNode = node;
                                while (parent != null) {
                                        int weight = getEdgeLength(parent, curNode);
                                        if (weight > -1) {
                                                tempMsp.addNode(parent);
                                                tempMsp.addEdgeLength(weight);
                                                curNode = parent;
                                                parent = getParent(parents, parent);
                                        } else
                                                break;
                                }

                                if (msp.getEdgeLength() == -1 || tempMsp.getEdgeLength() != -1
                                                && msp.getEdgeLength() > tempMsp.
                                                getEdgeLength())
                                        msp = tempMsp;
                        }
                }
                return msp;
        }
}

//*********************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch8p1_minpath.java
//the minimum path from starting node to all the other nodes, //including each node
//and the corresponding minimum distance as well
//*********************************************************************************************

import java.util.ArrayList;

public class ch8p1_minpath {
        private ArrayList<String> nodeList;
        private int edgeLength;

        public ch8p1_minpath(String node) {
                nodeList = new ArrayList<String>();
                nodeList.add(node);
                edgeLength = -1;
        }

        public ArrayList<String> getNodeList() {
                return nodeList;
        }

        public void setNodeList(ArrayList<String> nodeList) {
                this.nodeList = nodeList;
        }
        public void addNode(String node) {
                if (nodeList == null)
                        nodeList = new ArrayList<String>();
                nodeList.add(0, node);
        }

        public String getLastNode() {
                int size = nodeList.size();
                return nodeList.get(size - 1);

        }

        public int getEdgeLength() {
                return edgeLength;
        }

        public void setEdgeLength(int edgeLength) {
                this.edgeLength = edgeLength;
        }

        public void outputPath() {
                outputPath(null);
        }

        public void outputPath(String pathNode) {
                String result = "the minimum path of ";
                if (pathNode != null)
                        nodeList.add(pathNode);
                for (int i = 0; i < nodeList.size(); i++) {
                        result += "" + nodeList.get(i);
                        if (i < nodeList.size() - 1)
                                result += "->";
                }
                result += " is:" + edgeLength;
                System.out.println(result);
        }

        public void addEdgeLength(int e) {
                if (edgeLength == -1)
                        edgeLength = e;
                else
                        edgeLength += e;
        }

}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch8p1_edge.java
//edge class including the parent node and child node
//and the edge length between two nodes
//*******************************************************************
```

```
public class ch8p1_edge {
        private String parentNode;
        private String childNode;
        private int edgeLength;

        public ch8p1_edge(String parentNode, String childNode, int edgeLength) {
                this.parentNode = parentNode;
                this.childNode = childNode;
                this.edgeLength = edgeLength;
        }

        public String getParentNode() {
                return parentNode;
        }

        public void setParentNode(String parentNode) {
                this.parentNode = parentNode;
        }

        public String getChildNode() {
                return childNode;
        }

        public void setChildNode(String childNode) {
                this.childNode = childNode;
        }

        public int getEdgeLength() {
                return edgeLength;
        }

        public void setEdgeLength(int edgeLength) {
                this.edgeLength = edgeLength;
        }

}
```

The result of running the program is as follows:

```
The minimum path of A->C is: 2
The minimum path of A->D is: 3
The minimum path of A->C->B is: 5
The minimum path of A->D->E is: 8
The minimum path of A->C->B->F is: 9
The minimum path of A->C->J is: 9
The minimum path of A->D->I is:16
The minimum path of A->C->B->F->G is: 19
The minimum path of A->D->I->H is: 22
```

As you have seen, this result is exactly same as the one shown in Table 8-13.

## leJOS EV3-Based Dijkstra's Algorithm

Referring to Figure 8-5, you will write a program for your robot so it can travel the path between starting Node A and destination Node I covering the minimum distance possible using Dijkstra's algorithm. Assume that you use the Cartesian coordinate system:

The coordinate at A is (0,0).

The coordinate at B is (-10,20).

The coordinate at C is (0,20).

The coordinate at D is (-15,20).

The coordinate at E is (-15,30).

The coordinate at F is (0,30).

The coordinate at G is (10,10).

The coordinate at H is (10,20).

The coordinate at I is (10,30).

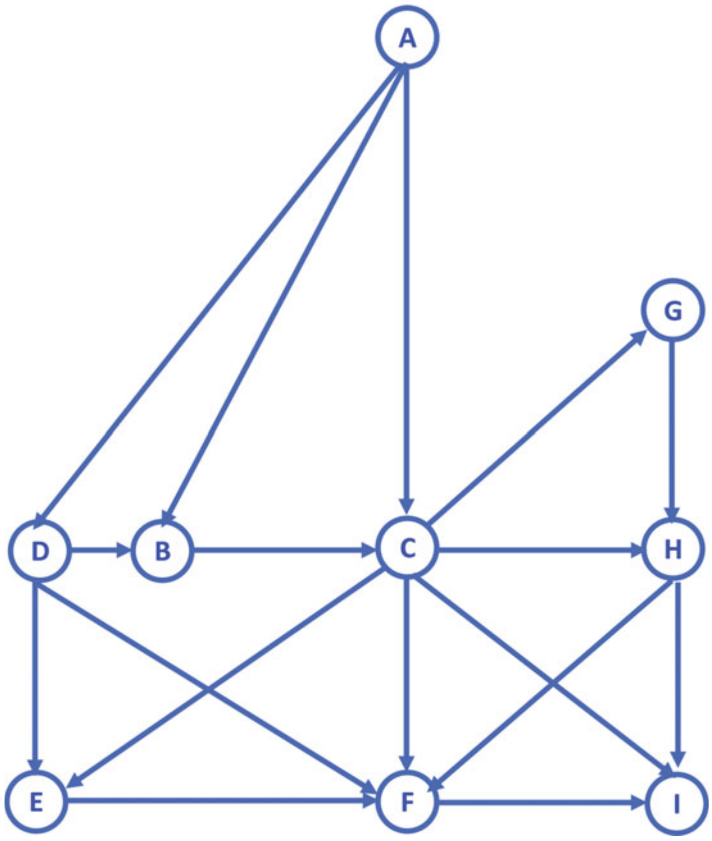


***Figure 8-5.*** *Graphic to be used for conducting the shortest-path search*

You program should present at least the destination's coordinate on the LCD and then show the message "`Press ENTER Key to continue`." When you press Enter, your robot moves to the next node. For instance, suppose that your robot starts from `Node A(0,0)` and you want it to explore a path to `Node G(10,10)`. Assume that the path your robot explores with minimum distance covered using Dijkstra's algorithm is `Node A -> Node C -> Node G`. At the starting point Node A, your program should do the following:

Display the destination's coordinate `Node C(0, 20)` on the LCD.

Display the message "`Press ENTER key to continue.`"

Go to the location with coordinate 0,20.

Display the destination's coordinate `Node G(10, 10)` on LCD.

Display the message "`Press ENTER key to continue.`"

Go to the location with coordinate 10,10.

Furthermore, your problem should have a string called *destination*, so that it's intelligent enough when changing the value of the destination, your robot can explore a new path from starting Node A to the new destination node in which the robot will pass with the minimum distance to reach destination node. (We assume that the starting node is always A, so the from string can be hard-coded.)

The following programs represent the implementation of the leJOS-based Dijkstra's algorithm designed to explore the shortest path from Node A to the destination Node I:

```
//****************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch8p2_main.java
//Driver class to set up map using ch8p2_edge, find the shortest //path from
//starting node A to all the other nodes in a given graph.
//and then robots will follow the path to move from start node A
//to the given destination node I
//****************************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.chassis.Chassis;
import lejos.robotics.chassis.Wheel;
import lejos.robotics.chassis.WheeledChassis;
import lejos.robotics.navigation.MovePilot;
import lejos.robotics.navigation.Navigator;
import lejos.robotics.navigation.Waypoint;
import lejos.robotics.pathfinding.Path;

import java.util.ArrayList;

public class ch8p2_main {
        static ArrayList<ch8p2_edge> graph = null;
        static ch8p2_edge[] parents = null;

        static ArrayList<String> unsolvedConn = null;
        static ArrayList<String> solvedConn = null;
        private static ch8p2_minpath finalPath;

        static EV3LargeRegulatedMotor LEFT_MOTOR = new EV3LargeRegulatedMotor(
                        MotorPort.A);
        static EV3LargeRegulatedMotor RIGHT_MOTOR = new EV3LargeRegulatedMotor(
                        MotorPort.C);

        static Waypoint[] coordinates = { new Waypoint(0, 0),
                        new Waypoint(-10, 20), new Waypoint(0, 20), new Waypoint(-15, 20),
                        new Waypoint(-15, 30), new Waypoint(0, 30), new Waypoint(10, 10),
                        new Waypoint(10, 20), new Waypoint(10, 30),

        };

        public static void main(String[] args) {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

// instantiated LCD class for displaying and Keys class // for buttons
                Keys buttons = ev3brick.getKeys();

// setup the wheel diameter of left (and right) motor // in centimeters,
                // i.e. 2.8 cm
// the offset number is the distance between the center // of wheel to
                // the center of robot, i.e. half of track width
                Wheel wheel1 = WheeledChassis.modelWheel(LEFT_MOTOR, 2.8).offset(-9);
                Wheel wheel2 = WheeledChassis.modelWheel(RIGHT_MOTOR, 2.8).offset(9);

                // set up the chassis type, i.e. Differential pilot
                Chassis chassis = new WheeledChassis(new Wheel[] { wheel1, wheel2 },
                                WheeledChassis.TYPE_DIFFERENTIAL);
                MovePilot ev3robot = new MovePilot(chassis);

                Navigator navbot = new Navigator(ev3robot);

                // initialize the nodes set
                String[] nodes = { "A", "B", "C", "D", "E", "F", "G", "H", "I" };

                // initialize the map with the nodes
                graph = new ArrayList<ch8p2_edge>();

                graph.add(new ch8p2_edge("A", "B", 22));
                graph.add(new ch8p2_edge("A", "C", 20));
                graph.add(new ch8p2_edge("A", "D", 25));

                graph.add(new ch8p2_edge("B", "C", 10));

                graph.add(new ch8p2_edge("C", "E", 18));
                graph.add(new ch8p2_edge("C", "F", 10));
                graph.add(new ch8p2_edge("C", "G", 14));
                graph.add(new ch8p2_edge("C", "H", 10));
                graph.add(new ch8p2_edge("C", "I", 18));

                graph.add(new ch8p2_edge("D", "B", 5));
                graph.add(new ch8p2_edge("D", "E", 10));
                graph.add(new ch8p2_edge("D", "F", 18));

                graph.add(new ch8p2_edge("E", "F", 15));

                graph.add(new ch8p2_edge("F", "I", 10));

                graph.add(new ch8p2_edge("G", "H", 10));

                graph.add(new ch8p2_edge("H", "F", 14));
                graph.add(new ch8p2_edge("H", "I", 10));

                // initialize the unsolved nodes
                unsolvedConn = new ArrayList<String>();

                // sets the parent node in the unsolved connection
// ArrayList to A
                unsolvedConn.add(nodes[0]);

                // initialize the solved nodes
                solvedConn = new ArrayList<String>();

                // Add all nodes to the solved connection tree
                for (int i = 1; i < nodes.length; i++)
                        solvedConn.add(nodes[i]);

                // create a parent array that will store all the edges
                parents = new ch8p2_edge[nodes.length];

// Set the initial node to A and make its parent null // with a weight cost
                // of zero
                parents[0] = new ch8p2_edge(null, nodes[0], 0);

                for (int i = 0; i < solvedConn.size(); i++) {
// get all of the String node names that could be // attached the
                        // root
                        String n = solvedConn.get(i);

                        // Check the weights of all the nodes that are
// attached to the root
                        // A node
                        // If they are attached will return positive
// weight if not will
                        // return -1
                        parents[i + 1] = new ch8p2_edge(nodes[0], n, getEdgeLength(
                                        nodes[0], n));
                }
                finalPath = null;
// while the solved nodes ArrayList is greater than zero
                while (solvedConn.size() > 0) {
// Create a minimum shortest path object to find // the shortest path
                        // to all connected points
                        ch8p2_minpath msp = getMinSideNode();
                        finalPath = msp;

                        String node = msp.getLastNode();
                        unsolvedConn.add(node);
                        setEdgeLength(node);
                }
                Path directions = finalPath.buildPath(coordinates);
                navbot.setPath(directions);
                navbot.singleStep(true);
                while (navbot.getWaypoint() != null) {
                        if (navbot.getWaypoint() != null) {
                                System.out.println("Next destination" + navbot.
                                getWaypoint());

                        }
                        System.out.println("Press Enter Key to Continue");
                        // block the thread until a button is pressed
                        buttons.waitForAnyPress();

                        navbot.followPath();
                        while (navbot.isMoving())
                                ;
                }
                // block the thread until a button is pressed
                buttons.waitForAnyPress();

        }

        public static String getParent(ch8p2_edge[] parents, String node) {
                if (parents != null) {
                        for (ch8p2_edge nd : parents) {
                                if (nd.getChildNode() == node) {
                                        return nd.getParentNode();
                                }
                        }
                }
                return null;
        }

        public static void setEdgeLength(String parentNode) {
                if (graph != null && parents != null && solvedConn != null) {
                        for (String node : solvedConn) {
                                ch8p2_minpath msp = getMinPath(node);
                                int w1 = msp.getEdgeLength();
                                if (w1 == -1)
```

```
                                        continue;
                                for (ch8p2_edge n : parents) {
                                        if (n.getChildNode() == node) {
                                                if (n.getEdgeLength() == -1 ||
n.getEdgeLength() > w1) {
                                                        n.setEdgeLength(w1);
                                                        n.setParentNode(parentNode);
                                                        break;
                                                }
                                        }
                                }
                        }
                }
        }

        public static int getEdgeLength(String parentNode, String childNode) {
                if (graph != null) {
                        for (ch8p2_edge s : graph) {
                                if (s.getParentNode() == parentNode
                                                && s.getChildNode() == childNode)
                                        return s.getEdgeLength();
                        }
                }
                return -1;
        }

        public static ch8p2_minpath getMinSideNode() {
                // Create a minimum shortest path object
                ch8p2_minpath minMsp = null;
                // while the solved node ArrayList is greater than zero
                if (solvedConn.size() > 0) {
                        // Create an index value set to zero
                        int index = 0;
                        // for each value in the solved nodes ArrayList
                        for (int j = 0; j < solvedConn.size(); j++) {
                                // Create a shortest path map get the
// MinPath of the solved node
                                ch8p2_minpath msp = getMinPath(solvedConn.get(j));
                                // if there is no minimum shortest path, // if the minimum
                                   shortest
                                // path
                                // does not equal -1
                                // AND the minimum shortest path get
// weight is less than the
                                // minimum shortest path get weight
                                if (minMsp == null || msp.getEdgeLength() != -1
                                                && msp.getEdgeLength() < minMsp.
                                                getEdgeLength()) {
                                        // set the minimum shortest path to // the minimum
                                           shortest
                                        // path
                                        minMsp = msp;
                                        // set the index value equal it the // node value j
                                        index = j;
                                }
                        }
                        // remove the index that you checked in the
// solved nodes
                        solvedConn.remove(index);

                }
                // return the MinShortPath object
                return minMsp;
        }

        public static ch8p2_minpath getMinPath(String node) {
                // Create a Minshort Path object that is an ArrayList // and set the take
                // in node as the base
                // in this case will always be zero
                ch8p2_minpath msp = new ch8p2_minpath(node);
                // if the parents array does not equal null and the
// unsolved nodes does
                // not equal null
                if (parents != null && unsolvedConn != null) {
                        for (int i = 0; i < unsolvedConn.size(); i++) {
                                ch8p2_minpath tempMsp = new ch8p2_minpath(node);
                                String parent = unsolvedConn.get(i);
                                String curNode = node;
                                while (parent != null) {
                                        int weight = getEdgeLength(parent, curNode);
                                        if (weight > -1) {
                                                tempMsp.addNode(parent);
                                                tempMsp.addEdgeLength(weight);
                                                curNode = parent;
                                                parent = getParent(parents, parent);
                                        } else
                                                break;
                                }

                                if (msp.getEdgeLength() == -1 || tempMsp.getEdgeLength() !=
-1

                                                && msp.getEdgeLength() > tempMsp.
                                                getEdgeLength())
                                        msp = tempMsp;
                        }
                }
                return msp;
        }
}
```

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch8p2_edge.java
//edge class including the parent node and child node
//and the edge length between two nodes
//*******************************************************************

public class ch8p2_edge {
        private String parentNode;
        private String childNode;
        private int edgeLength;

        public ch8p2_edge(String parentNode, String childNode, int edgeLength) {
                this.parentNode = parentNode;
                this.childNode = childNode;
                this.edgeLength = edgeLength;
        }

        public String getParentNode() {
                return parentNode;
        }

        public void setParentNode(String parentNode) {
                this.parentNode = parentNode;
        }

        public String getChildNode() {
                return childNode;
        }

        public void setChildNode(String childNode) {
                this.childNode = childNode;
        }

        public int getEdgeLength() {
                return edgeLength;
        }

        public void setEdgeLength(int edgeLength) {
                this.edgeLength = edgeLength;
        }

}

//*************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch8p2_minpath.java
//the minimum path from starting node to all the other nodes, //including each node
//and the corresponding minimum distance as well
//*************************************************************************************

import java.util.ArrayList;
import lejos.robotics.navigation.Waypoint;
import lejos.robotics.pathfinding.Path;

public class ch8p2_minpath {
        private ArrayList<String> nodeList;
        private int edgeLength;

        public ch8p2_minpath(String node) {
                nodeList = new ArrayList<String>();
                nodeList.add(node);
                edgeLength = -1;
        }

        public ArrayList<String> getNodeList() {
                return nodeList;
        }

        public void setNodeList(ArrayList<String> nodeList) {
                this.nodeList = nodeList;
        }

        public void addNode(String node) {
                if (nodeList == null)
                        nodeList = new ArrayList<String>();
                nodeList.add(0, node);
        }

        public String getLastNode() {
                int size = nodeList.size();
                return nodeList.get(size - 1);

        }

        public int getEdgeLength() {
                return edgeLength;
        }

        public void setEdgeLength(int edgeLength) {
                this.edgeLength = edgeLength;
        }

        public void outputPath() {
                outputPath(null);
        }

        public Path buildPath(Waypoint[] coordinates) {
                Path result = new Path();
                int p = 0;
                for (int i = 0; i < nodeList.size(); i++) {
                        if (nodeList.get(i).equals("A"))
                                p = 0;
```

```
                    else if (nodeList.get(i).equals("A"))
                            p = 0;
                    else if (nodeList.get(i).equals("B"))
                            p = 1;
                    else if (nodeList.get(i).equals("C"))
                            p = 2;
                    else if (nodeList.get(i).equals("D"))
                            p = 3;
                    else if (nodeList.get(i).equals("E"))
                            p = 4;
                    else if (nodeList.get(i).equals("F"))
                            p = 5;
                    else if (nodeList.get(i).equals("G"))
                            p = 6;
                    else if (nodeList.get(i).equals("H"))
                            p = 7;
                    else if (nodeList.get(i).equals("I"))
                            p = 8;
                    result.add(coordinates[p]);
            }

            return result;

    }

    public void outputPath(String pathNode) {
            String result = "the munimum path of ";
            if (pathNode != null)
                    nodeList.add(pathNode);
            for (int i = 0; i < nodeList.size(); i++) {
                    result += "" + nodeList.get(i);
                    if (i < nodeList.size() - 1)
                            result += "->";
            }
            result += " is:" + edgeLength;
            System.out.println(result);
    }

    public void addEdgeLength(int e) {
            if (edgeLength == -1)
                    edgeLength = e;
            else
                    edgeLength += e;
    }

}
```

## Summary

This chapter introduced the basic idea behind and the fundamentals of Dijkstra's search algorithm. You are now able to apply Dijkstra's algorithm to solve path-planning issues in practice. In particular, the chapter presented how to build a problem-solving agent based on Dijkstra's path-planning algorithm and the Navigation class in which the problem-solving agent could find a route path intelligently from the starting point to any destination, covering the shortest distance possible.

In the next chapter, you will see how to build a problem-solving agent based on the A* path-planning algorithm and the Navigation class in which the problem-solving agent can intelligently find a route path from the starting point to any destination in a maze.

CHAPTER 9

# The A* Search Algorithm and Its Implementation with Lego Mindstorms

This chapter will introduce you to the fundamentals of the A* search algorithm. After completing this chapter, you will be able to apply the A* algorithm to solve path-planning issues in practice. You will also see how to build a problem-solving agent based on the A* path-planning algorithm and the Navigation class in which the problem-solving agent can find a route path intelligently from the starting point to any destination in a maze.

In particular, this chapter will cover the following topics:

- What is the A* algorithm?
- The basic idea of the A* searching strategy.
- Programming practice for path planning using the A* algorithm.

## What Is the A* Algorithm?

A-Star (or A*) is a well-known search algorithm that is extremely competitive compared to other search algorithms when solving path findings in terms of efficiency. In particular, the A* algorithm is best used for those problems that can be represented as a state space, for example, exploring a path in a maze. Given a suitable problem, the initial conditions of the problem can be represented with an appropriate *initial state*, and the goal conditions can be represented as the *goal state*.

For each action that you perform, the A* algorithm generates successor states to represent the outcome of that action. If you keep doing this, and at some point one of the generated successor states is the goal state, then the path from the initial state to the goal state is the solution to your problem. Moreover, the A* algorithm generates and processes the successor states in a certain way; that is, whenever it is looking for the next state to continue, the A* algorithm employs a heuristic function to try to pick the best state to process next.

## The Basic Idea of the A* Searching Strategy

As shown in Figure 9-1, assume that you want to get from Node 1 to Node 5 and that there is a wall separating Node 1 and Node 5. The search area is split into a square-based grid, and the path that you will try to find is which squares to take to get from Node 1 to Node 5. There are 15 squares, labeled 1 to 15, in the entire search area. The upper-left corner is Starting Point 1, the upper-right corner is Ending Point 5, and Nodes 3 and 8 are located at the center to represent the wall.

W. Lu, *Beginning Robotics Programming in Java with LEGO Mindstorms*, DOI 10.1007/978-1-4842-2005-4_9

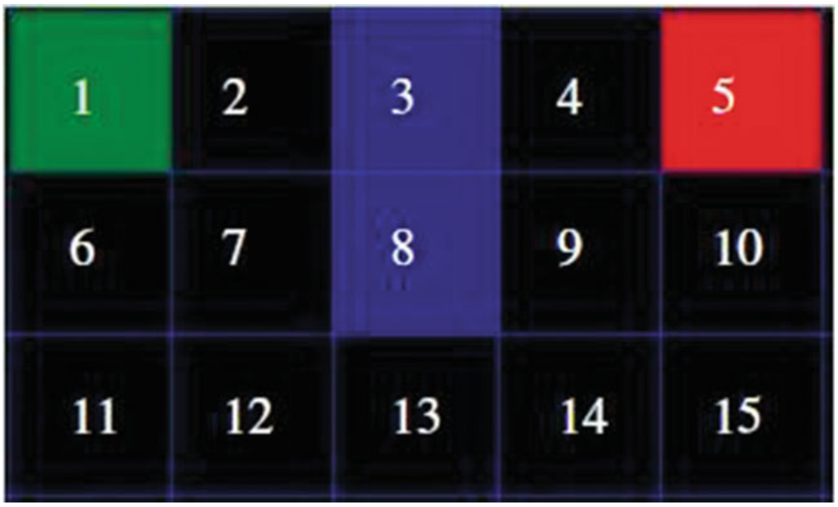


***Figure 9-1.*** *The search area of the A* Algorithm*

Once you have simplified your search area into a predefined number of nodes, the next step is to conduct a search to find the shortest path from Starting Point Node 1 to Ending Point Node 5, the ultimate goal. Starting at Node 1, you check its adjacent squares and then generally search outward until you find target Node 5.

You begin the A* search by doing the following:

1. Begin at Starting Point 1 and add it to an "open list," which contains squares that fall along the path that you want to take. Basically, this is a list of squares that need to be checked out. For example, `OpenList = {1}`.
2. Look at all of the reachable squares adjacent to Starting Point 1, ignoring squares with walls, and also add them to the open list. For each of these squares, save Node 1 as its "parent square": that is, `OpenList = {1,2,6}`.
3. Drop the Starting Point square 1 from the open list, and then add it to a "closed list" of squares, which you don't need to look at again for now. Thus you have `OpenList = {2,6}` and `ClosedList = {1}`.
4. All of the adjacent squares are now on the open list of squares to be checked.

Next you need to choose one of the adjacent squares on the open list to continue. However, there is the question of which square do you choose? The answer is that it is the one with the lowest F cost, where F is Path Scoring, which is the key to determining which squares to use.

F is the sum of G and H (F = G + H) in which G is the cost of moving from starting Node 1 to a given square on the grid, following the path generated to get there, and H is the estimated cost of moving from that square on the grid to the final destination, that is, ending Node 5. The heuristic function H involves generating your path by repeatedly going through your open list and choosing the square with the lowest F score.

First let's look more closely at how you calculate the equation.

$$F = G + H$$

G is the cost of moving from starting Node 1 to the given square using the path generated to get there. In this example, you assign a cost of 1 to each horizontal or vertical square moved. Thus you have the following:

***G(1->2)=1***

***G(1->6)=1***

Heuristic function H can be estimated in a variety of ways. In this example, the Manhattan method is used, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement and any obstacles that may be in the way. As a result, you have heuristic function H:

***H(2->5)=3***

***H(6->5)=5***

F is calculated by adding G and H.

***F(2)=1+3=4***

***F(6)=1+5=6***

Next, as illustrated in Figure 9-2, you simply choose the lowest F score square, which is Node 2, from the existing nodes on the open list. Then you drop Node 2 from the open list and add it to the closed list, that is, `OpenList = {6}` and `ClosedList = {1,2}`.

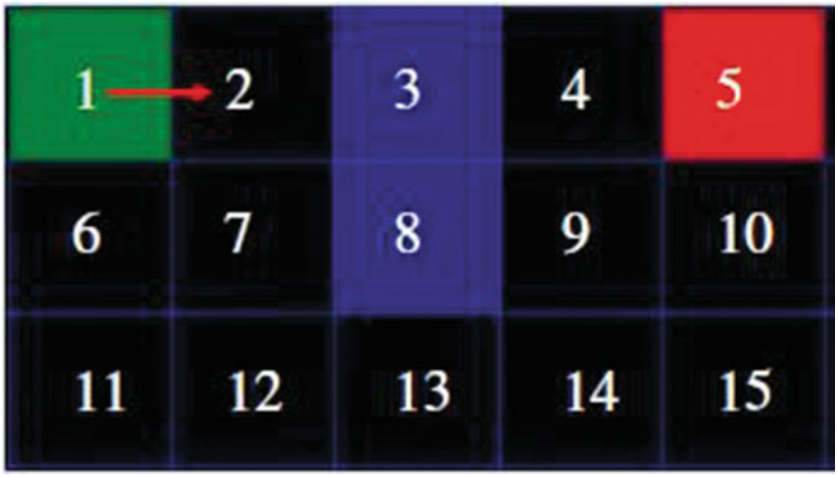


***Figure 9-2.*** *Search from Node 1 to Node 2*

Check all of the adjacent squares of Node 2. Ignoring those that are on the closed list or unreachable (Node 3 is unreachable because it belongs to the wall), add squares to the open list if they are not on this list already. Make the selected square the "parent" of the new squares. As a result, you have the following:

`OpenList = {6,7}` in which 6's parent is Node 1 and 7's parent is Node 2, as shown in Figure 9-3.

`ClosedList = {1,2}`

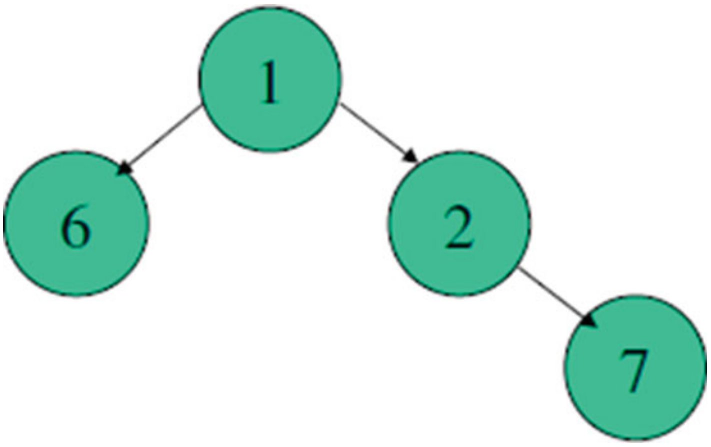


***Figure 9-3.*** *Tree structure of nodes after visiting Node 2*

Now you have two nodes in the open list (Node 6 and Node 7). You then calculate the value of F and then pick the one with the lowest F cost.

```
OpenList = {6,7}
ClosedList = {1,2}
```

In this case, Node 6 and Node 7 have the same F score of 6. So which do you choose? For the purposes of speed, it can be faster to choose the one with the smaller H score, which is Node 7. Then you simply choose the lowest F score square, which is Node 7, from existing nodes on the open list and drop Node 7 from the open list and add it to the closed list, that is, `OpenList = {6}` and `ClosedList = {1,2,7}`.

Next you check all of the adjacent squares of Node 7. Ignoring those that are on the closed list or unreachable (Node 8 is unreachable), add squares to the open list if they are not on this list already. Make the selected square the "parent" of the new squares, as illustrated in Figure 9-4. As a result, you have the following:

```
OpenList = {6,12}
ClosedList = {1,2,7}
```

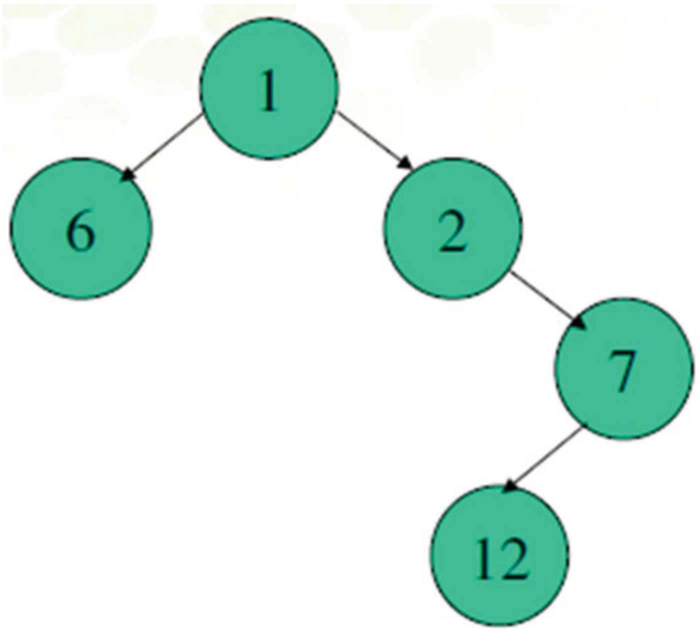


***Figure 9-4.*** *Tree structure of nodes after visiting Node 7*

In this case, Node 6 is already on the open list. You then need to check to see if the current path to that Node 6 is a better one. Using the current path `1->2->7`, you can calculate current G score of Node 6, `G(1->2->7->6)`, which is 3 and then compare it with the previous G score, `G(1->6)`, which is 1. If the current G score is lower than the previous one, you change the parent node of Node 6 to Node 7. If it's not lower than the previous one, do nothing on the list. Since the previous G score `(1->6)` is smaller than the current G score `(1->2->7->6)`, you don't have to do anything on the list.

Next you calculate the function F for Node 6 and Node 12:

- You have known `F(6) = 6`
- `H(12->5)=5`
- `G(1->12)=3`

Thus you have `F(12) = 8`, which is bigger than `F(6)`. Node 6 is then chosen as the next node, and you can drop Node 6 from the open list and add it to the closed list as follows:

```
OpenList = {12}

ClosedList = {1,2,7,6}
```

Node 6's parent is 1, as illustrated in Figure 9-5.

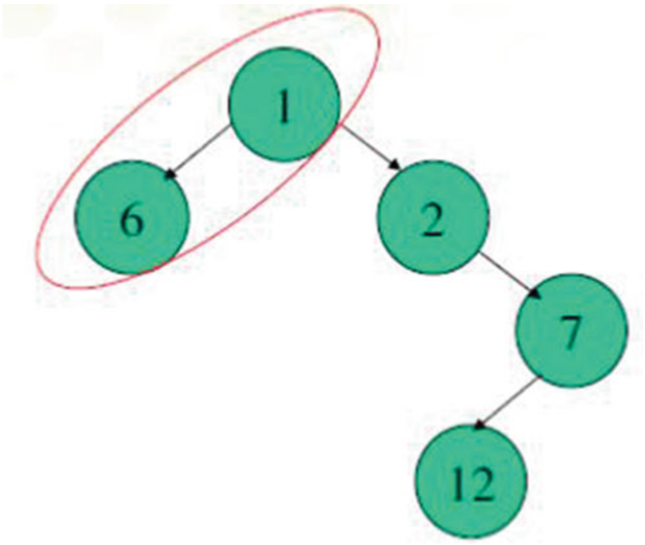


***Figure 9-5.*** *Tree structure of nodes in which Node 6 is chosen as the next node*

Check all of the adjacent squares of Node 6. Ignoring those that are on the closed list or unreachable, add squares to the open list if they are not on this list already. Make the selected square the "parent" of the new squares, as illustrated in Figure 9-6. As a result, you have the following:

```
OpenList = {12,11}
ClosedList = {1,2,7,6}
```

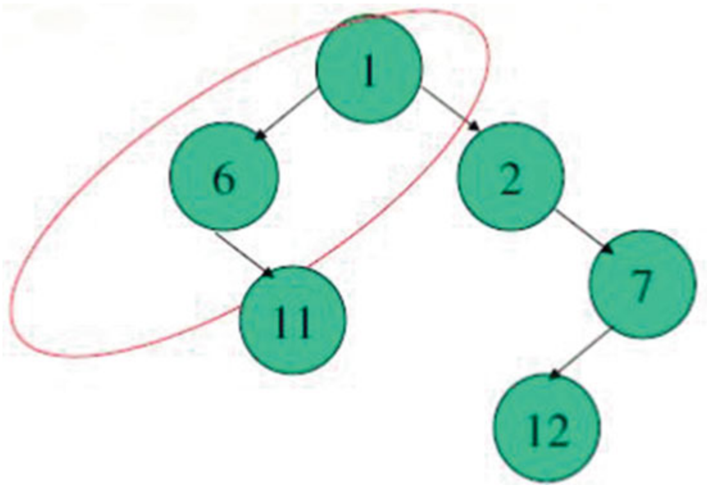


***Figure 9-6.*** *Tree structure of nodes after visiting Node 6*

Next you calculate the function F for Node 11 and Node 12:

- You have known `F(12) = 8`
- `H(11->5)=6`
- `G(1->11)=2`

Thus you have `F(12) = 8`, which is same with `F(11)`. So which do you choose? Similarly, for the purposes of speed, it can be faster to choose the one with the smaller H score, which is Node 12.

Node 12 is then chosen as the next node, and you drop Node 12 from the open list and add it to the closed list, thus you have the following:

```
OpenList = {11}

ClosedList = {1,2,7,6,12}
```

Node 12's parent is 7, as illustrated in Figure 9-7.

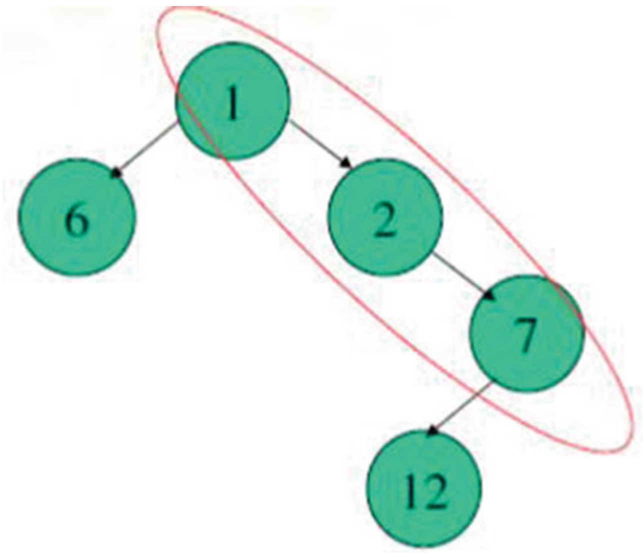


***Figure 9-7.*** *Tree structure of nodes in which Node 12 is chosen as the next node*

Check all of the adjacent squares of Node 12. Ignoring those that are on the closed list or unreachable, add squares to the open list if they are not on this list already. Make the selected square the "parent" of the new squares, as illustrated in Figure 9-8.

```
OpenList = {11,13}

ClosedList = {1,2,7,6,12}
```

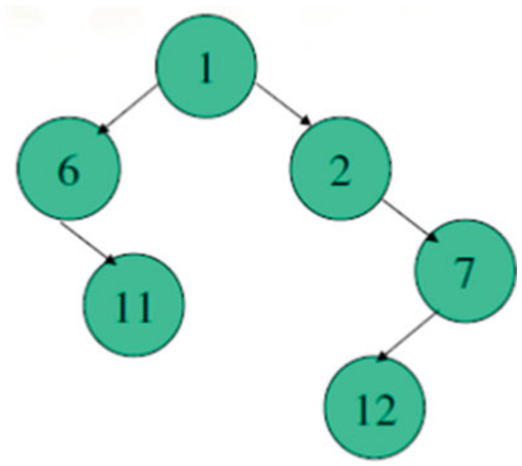


***Figure 9-8.*** *Tree structure of nodes after visiting Node 12*

Next you calculate the function F for Node 11 and Node 13:

- You have known `F(11) = 8`
- `H(13->5)=4`
- `G(1->13)=4`

Thus you have `F(13) = 8`, which is same with `F(11)`. So which do you choose? Similarly, for the purposes of speed, it can be faster to choose the one with the smaller H score, which is Node 13.

Node 13 is then chosen as the next node. You drop Node 13 from the open list and add it to the closed list as follows:

```
OpenList = {11}
ClosedList = {1,2,7,6,12,13}
```

Node 13's parent is 12, as illustrated in Figure 9-9.

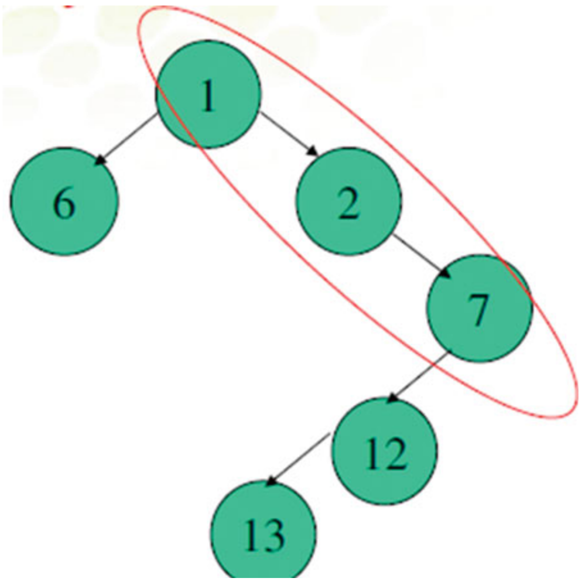


***Figure 9-9.*** *Tree structure of nodes in which Node 13 is chosen as the next node*

Check all of the adjacent squares of Node 13. Ignoring those that are on the closed list or unreachable, add squares to the open list if they are not on this list already. Make the selected square the "parent" of the new squares, as illustrated in Figure 9-10.

```
OpenList = {11,14}
ClosedList = {1,2,7,6,12,13}
```

Next you calculate the function F for Node 11 and Node 14:

- You have known `F(11)=8`
- `H(14->5)=5`
- `G(1->14)=3`

Thus you have `F(14) = 8`, which is the same with `F(11)`. So which do you choose? Similarly, for the purposes of speed, it can be faster to choose the one with the smaller H score, which is Node 14.

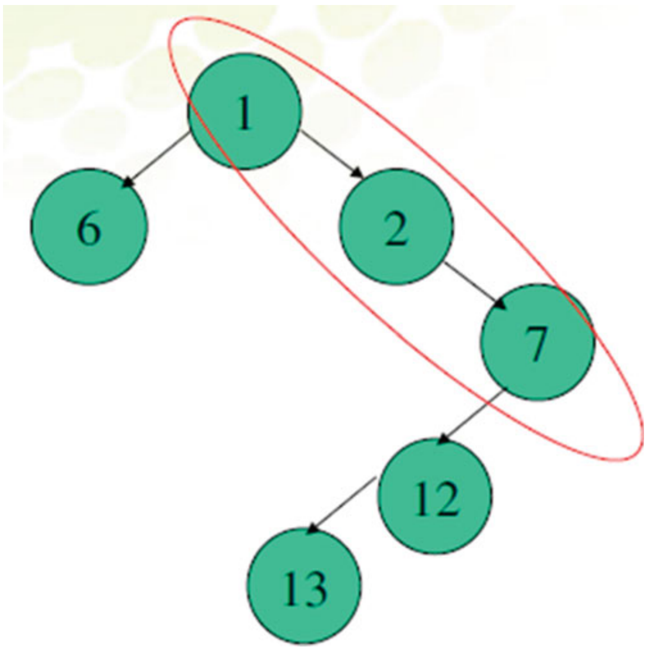


***Figure 9-10.*** *Tree structure of nodes after visiting Node 13*

Node 14 is then chosen as the next node. You drop Node 14 from the open list and add it to the closed list as follows:

```
OpenList = {11}
ClosedList = {1,2,7,6,12,13,14}
```

Node 14's parent is 13.

Check all of the adjacent squares of Node 14. Ignoring those that are on the closed list or unreachable, add squares 9 and 15 to the open list if they are not on this list already. Make the selected square the "parent" of the new squares.

```
OpenList = {11,9,15}
ClosedList = {1,2,7,6,12,13,14}
```

Next you calculate the function F for Node 11, Node 9, and Node 15.

- You have known `F(11) = 8`
- Following our previous thinking, you can calculate `F(9) = 8` and `F(15) = 8`, which are same with `F(11)`.

Which do you choose? Similarly, for the purposes of speed, it can be faster to choose the one with the smaller H score. Node 15 is then chosen as the next node, you drop Node 15 from the open list, and add it to the closed list as follows:

`OpenList = {11,9}`

`ClosedList = {1,2,7,6,12,13,14,15}`, 15's parent is 14

Check all of the adjacent squares of Node 15. Ignoring those that are on the closed list or unreachable, add squares to the open list if they are not on this list already. Make the selected square the "parent" of the new squares.

`OpenList = {11,9,10}`

`ClosedList = {1,2,7,6,12,13,14,15}`

Next you calculate the function F for Node 11, Node 9, and Node 10.

You have `F(11)=F(9)=F(10)=8`

Similarly, for the purposes of speed, it can be faster to choose the one with the smaller H score, which is Node 10. Node 10 is then chosen as the next node, you drop Node 10 from the open list, and add it to the closed list as follows:

`OpenList = {11,9,5}`

`ClosedList = {1,2,7,6,12,13,14,15,10}`, 10's parent is 15

The goal state Node 5 is found, and you stop it there. Thus the shortest path is in the closed list according to the parent node, which is the following:

```
1->2->7->12->13->14->15->10->5
```

## Practice for Path Planning Using the A* Algorithm

Pathfinding on a maze is an interesting problem, which has largely been conquered though computing. The goal in this practice is to solve a simple maze in the shortest time possible, starting from one point to another, regardless of the obstacles that stand between these two points. As shown in Figure 9-11, your robot will explore the path starting from the blue circle and ending at the green oval. A flat surface is a square with a dimension of 90 inches by 90 inches: that is, the distance between H and I is 90 inches.

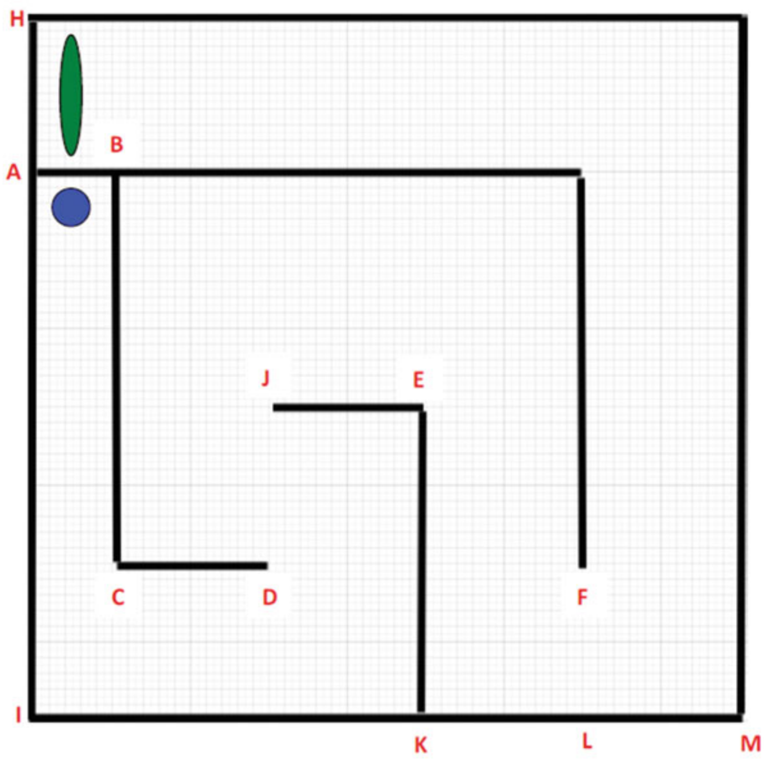


***Figure 9-11.*** *Maze map*

From the map in Figure 9-11, you can see the following:

The distance between points A and B is 10 inches.

The distance between points A and H is 20 inches.

The distance between points A and I is 70 inches.

The distance between points B and C is 50 inches.

The distance between points C and D is 20 inches.

The distance between points J and E is 20 inches.

The distance between points E and K is 40 inches.

The distance between points K and L is 20 inches.

The distance between points L and M is 20 inches.

To allow for a wide range of maze-solving methods, the starting and ending points of the maze will be known in advance, as displayed in Figure 9-11, and there will be no loops. In the real demonstration, you will mark A using a blue color to identify point A as the starting point (in the case of the Cartesian Coordinates system, you can, for example, say that A's coordinator is `(0,-20)`). Furthermore, you are going to use a green color to mark line A to H so that your robot can stop when it reaches the green color destination, B). In the map itself, all intersections will be at right angles. The black color line represents the wall surroundings.

To summarize, in this practice you will be creating a maze-traveling robot that can explore a path starting from a blue color area and ending at a green color area on the map. Moreover, your robot can detect the green color line of the ending area and then stop by identifying it when it finds the final destination.

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch9p1.java
//simple line map and grid using A* search build into LeJOS
//*******************************************************************

import lejos.geom.*; //used for rectangle
import lejos.robotics.RegulatedMotor; //motor controller
import lejos.robotics.localization.*; //numbers
import lejos.robotics.mapping.LineMap; //mapping
import lejos.robotics.navigation.*; //navigation used for the
// waypoints
import lejos.robotics.pathfinding.*; //A* search algorithm
import lejos.util.PilotProps; //not used really
import lejos.nxt.Sound;
import lejos.robotics.navigation.DifferentialPilot;

public class ch9p1 {

          private static final short[] note = { 2349, (115 / 3), 0, (5 / 3), 1760,
                    (165 / 3), 0, (35 / 3), 1760, (28 / 3), 0, (13 / 3), 1976,
                    (23 / 3), 0, (18 / 3), 1760, (18 / 3), 0, (23 / 3), 1568, (15
                    / 3),
                    0, (25 / 3), 1480, (103 / 3), 0, (18 / 3), 1175, (180 / 3), 0,
                    (20 / 3), 1760, (18 / 3), 0, (23 / 3), 1976, (20 / 3), 0, (20
                    / 3),
                    1760, (15 / 3), 0, (25 / 3), 1568, (15 / 3), 0, (25 / 3),
                    2217,
                    (98 / 3), 0, (23 / 3), 1760, (88 / 3), 0, (33 / 3), 1760, (75
                    / 3),
                    0, (5 / 3), 1760, (20 / 3), 0, (20 / 3), 1760, (20 / 3), 0,
                    (20 / 3), 1976, (18 / 3), 0, (23 / 3), 1760, (18 / 3), 0, (23
                    / 3),
                    2217, (225 / 3), 0, (15 / 3), 2217, (218 / 3) };

          static RegulatedMotor leftMotor; // motors
          static RegulatedMotor rightMotor;

          public static void main(String[] args) {

                    // set up the robot
                    PilotProps pp = new PilotProps();
                    float wheelDiameter = Float.parseFloat(pp.getProperty(
                              PilotProps.KEY_WHEELDIAMETER, "2.11"));
                    float trackWidth = Float.parseFloat(pp.getProperty(
                              PilotProps.KEY_TRACKWIDTH, "5.45"));

                    RegulatedMotor leftMotor = PilotProps.getMotor(pp.getProperty(
                                    PilotProps.KEY_LEFTMOTOR, "C"));

                    RegulatedMotor rightMotor = PilotProps.getMotor(pp.getProperty(
                                    PilotProps.KEY_RIGHTMOTOR, "A"));
                    leftMotor.setSpeed(750);
                    rightMotor.setSpeed(750);
                    leftMotor.setAcceleration(1000);
                    rightMotor.setAcceleration(1000);
                    boolean reverse = Boolean.parseBoolean(pp.getProperty(
                                    PilotProps.KEY_REVERSE, "false"));

                    // new robot object using the setup
                    DifferentialPilot robot = new DifferentialPilot(wheelDiameter,
                                    trackWidth, leftMotor, rightMotor, reverse);

                    // make the robot move faster this is over max
                    // robot.setTravelSpeed(500);
                    robot.setRotateSpeed(600);

                    // Create final map:
                    Line[] lines = new Line[8]; // six lines inside the map
                    lines[0] = new Line(-2.5f, -2.5f, -2.5f, 67.5f);
// line AG
                    lines[1] = new Line(-5.0f, -2.5f, -5.0f, 67.5f);
                    lines[2] = new Line(-2.5f, 67.5f, 47.5f, 67.5f);
// line GF
                    lines[3] = new Line(-2.5f, 7.5f, 47.5f, 7.5f);
// line BC
                    lines[4] = new Line(47.5f, 7.5f, 47.5f, 27.5f);
// line cd
                    lines[5] = new Line(44f, 7.5f, 44f, 27.5f);
// cd broader
                    lines[6] = new Line(27.5f, 27.5f, 27.5f, 47.5f);
// line je
                    lines[7] = new Line(27.5f, 47.5f, 67.5f, 47.5f);
// line ek

                    lejos.geom.Rectangle bounds = new Rectangle(-22.5f, -2.5f, 90f, 90f);
                    LineMap myMap = new LineMap(lines, bounds); // add the //bounds to the
                    map

                    // Use a regular grid of node points. Grid space = 20. //Clearance = 15:
                    FourWayGridMesh grid = new FourWayGridMesh(myMap, 10, 2f);

                    // Use A* search:
                    AstarSearchAlgorithm alg = new AstarSearchAlgorithm();

                    // Give the A* search alg and grid to the PathFinder:
                    PathFinder pf = new NodePathFinder(alg, grid);

                    // store the location of the robot at a given time
```

```
                PoseProvider posep = new OdometryPoseProvider(robot);

                // new navigator loaded with the robot position, and //path
                NavPathController nav = new NavPathController(robot, posep, pf);

                System.out.println("Planning path…"); // displays as //the path is
                calculated
                nav.goTo(-12, 0); // goto the end location.

                for (int i = 0; i < note.length; i += 2) {
                        final short w = note[i + 1];
                        final int n = note[i];
                        if (n != 0)
                                Sound.playTone(n, w * 10);
                        try {
                                Thread.sleep(w * 10);
                        } catch (InterruptedException e) {
                        }
                }

        }
}
```

## Summary

This chapter introduces the fundamentals of the A* search algorithm. Upon completing the chapter, you were able to apply the A* algorithm to solve path-planning issues in practice. Specifically, this chapter presented how to build a problem-solving agent based on the A* path-planning algorithm and the Navigation class in which the problem-solving agent found a route path intelligently from the starting point to any destination and with the shortest distance possible.

In the next chapter, you will learn how to apply a set of sensors to perform informed movements. In particular, you will take the wheeled robot that you create and add the two sensors — touch sensor and ultrasonic sensor — to make it more aware of its surroundings as it moves around.

CHAPTER 10

# Introducing the Touch Sensor and Ultrasonic Sensor

As you saw in Chapter 3, motors are the most important component for performing movement. In this chapter, you will learn about a set of sensors that are applied to perform informed movements.

Think about a robot traveling around your house not equipped with any sensors. Such a travel would be short lived as the robot ran into a wall without the bumper activating or blocking of the wheels by some low-level obstacles. However, if you have installed a distance sensor in the robot, you might imagine that the robot could wander around your house for hours by avoiding obstacles using this distance sensor.

The LEGO Mindstorms EV3/NXT kit comes with four common sensors: the touch sensor, the color sensor, the light sensor, and the ultrasonic sensor. In this chapter, you take the wheeled robot that you create and add two of these four sensors — the touch sensor and the ultrasonic sensor — to make it more aware of its surroundings as it moves around a room. In the next chapter, you will see how to apply the light sensor and color sensor to explore the environment surrounding your robot.

In particular, this chapter will cover the following topics:

- Introduction to the touch sensor
- Introduction to the ultrasonic sensor
- Programming practice with the touch sensor
- Programming practice with the ultrasonic sensor

## Sensor Classes

The leJOS EV3 provides software abstractions for all types of sensors. In this case, a sensor must be physically connected to a port in the EV3 brick, and the sensor object must know which port this is. In order to identify this information, you create an instance of the sensor and then pass this information in its constructor, such as `SensorPort.S1`, `SensorPort.S2`, `SensorPort.S3`, and `SensorPort.S4`. In your code, the sensor classes must indicate which port they are plugged into. This can be done by using the Port class `lejos.hardware.port.Port`.

The Port class is similar to `MotorPort`. There are four static Port objects in this class. Generally, these are used in the constructors for sensors as follows:

```
Port portS1 = ev3brick.getPort("S1");
EV3TouchSensor touchSensor = new EV3TouchSensor(portS1);
```

## Touch Sensor

The touch sensor is the most basic sensor in the EV3 robotics kit. It has a simple switch activated by a red button on the front, as shown in Figure 10-1:

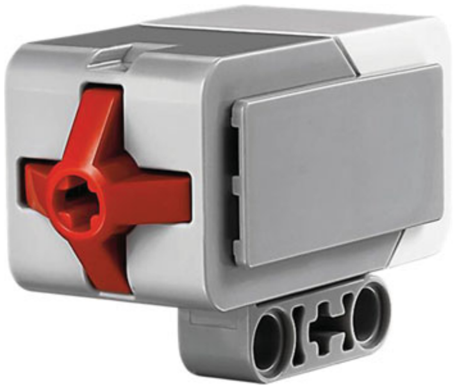


***Figure 10-1.*** *The Lego EV3 touch sensor*

As seen in Figure 10-1, the touch button has an axle hole, allowing a LEGO axle to attach directly to the switch. The touch sensor detects when it is being pressed by something and when it is released again, and it will return a simple float value indicating if the sensor is pressed or not. The `EV3TouchSensor` class implements the Touch interface, which contains a simple method, `getTouchMode ()`. The `getTouchMode()` method detects when its front button is pressed in which the sample contains one element. A value of 0 indicates that the button is not pressed, and a value of 1 indicates that the button is pressed.

In summary, in order to use a touch sensor, you can create an instance of it using the following constructor:

```
EV3TouchSensor touchSensor = new EV3TouchSensor(portS1);
SensorMode toucher = touchSensor.getTouchMode();
float[] sample = new float[toucher.sampleSize()];
```

To test if the touch sensor is pressed, you can use the `fetchSample ()` method:

```
toucher.fetchSample(sample, 0);
if (sample[0]==1) then the button is pressed
```

The following program waits for the touch sensor to be pressed. After you press the touch sensor, the value of the touch sensor will be displayed in the LED screen for 50 seconds. The program is terminated until you press the ESCAPE button on the EV3 brick. The source code follows for this testing program in which port S1 is used.

```
//********************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch10p1.java
//an example for touch sensor testing
//********************************************************************
```

```
import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.TextLCD;
import lejos.hardware.port.Port;
import lejos.hardware.sensor.EV3TouchSensor;
import lejos.hardware.sensor.SensorMode;

public class ch10p1 {
        public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

                // LCD class for displaying and Keys class for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // get a port instance
                Port portS1 = ev3brick.getPort("S1");

                // Get an instance of the touch EV3 sensor
EV3TouchSensor touchSensor = new EV3TouchSensor(portS1);

                // get an instance of this sensor in measurement mode
                SensorMode toucher = touchSensor.getTouchMode();

                // initialize an array of floats for fetching samples
                float[] samplevalue = new float[toucher.sampleSize()];

                lcddisplay.clear();

                while (buttons.getButtons() != Keys.ID_ESCAPE) {

                        // fetch a sample
                        toucher.fetchSample(samplevalue, 0);

                        lcddisplay.drawString("value: " + samplevalue[0], 0, 0);
                        Thread.sleep(50);
                }

                lcddisplay.clear();
                System.out.println("EXIT");
                System.exit(0);
        }

}
```

## Ultrasonic Sensor

The ultrasonic sensor sends out a sound signal that is nearly inaudible to humans, and then it measures how long it takes for the reflection to return. Since the speed of sound is well known, the sensor can then calculate the distance that the signal has traveled. The ultrasonic sensor is often used to avoid obstacles, navigate between locations, and even build maps in between different positions. The ultrasonic sensor measures distances to a solid object in centimeters, and it can measure distances of up to 255 centimeters. The ultrasonic sensor is accurate from 6 to 180 centimeters. Empirical experiments show that the objects beyond 180 centimeters are not reliably located. Also, it has an accuracy of plus or minus 3 centimeters, even though accuracy is improved for closer objects.

As illustrated in Figure 10-2, the ultrasonic sensor often produces a sonar cone, which means that it detects objects in front of it within a cone shape. The cone opens at an angle of about 30 degrees, which means that at a distance of 180 centimeters, the cone is about 90 centimeters in diameter. The cone shape is good for robots because it's better to scan a larger area in front of the robots for possible collisions.

***Figure 10-2.*** *The Lego EV3 ultrasonic sensor*

In leJOS NXJ, the ultrasonic sensor is a range sensor that implements the *RangeFinder* interface. The following method is used to obtain distances using the RangeFinder interface.

```
double         distance  = rf.getRange();
```

This method will return the distance to an object in centimeters. When there are multiple objects detected from a single scan, the following method can be used to obtain the corresponding distance:

```
float [] distances  = rf.getRanges();
```

To create an instance, you can use the following constructor:

```
UltrasonicSensor(Port SensorPort)
```

In leJOS NXJ, the ultrasonic sensor operates in two modes: continuous mode by default and ping mode. In ping mode, the sensor sends a single ping. When operating in continuous mode, the sensor sends out pings as often as it can and the most-recently obtained result is available via a call to the following:

```
int getDistance()
```

The return value is measured in centimeters. If no echo is found, the returned value will be 255.

The following program will test if your ultrasonic sensor works; that is, it is able to identify the distance in centimeters between your robot and the obstacle in front of it. In the program, we use port number S1 and terminate the program after pressing the ESCAPE key.

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch10p2.java
//an example for ultrasonic sensor testing
//*******************************************************************

import lejos.hardware.BrickFinder;
import lejos.hardware.Keys;
import lejos.hardware.ev3.EV3;
import lejos.hardware.lcd.TextLCD;
import lejos.hardware.port.Port;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.SampleProvider;

public class ch10p2 {
        public static void main(String[] args) throws Exception {

                // get EV3 brick
                EV3 ev3brick = (EV3) BrickFinder.getLocal();

                // LCD class for displaying and Keys class for buttons
                Keys buttons = ev3brick.getKeys();
                TextLCD lcddisplay = ev3brick.getTextLCD();

                // block the thread until a button is pressed
                buttons.waitForAnyPress();

                // get a port instance
                Port portS1 = ev3brick.getPort("S1");

                // Get an instance of the Ultrasonic EV3 sensor
EV3UltrasonicSensor sonicSensor = new EV3UltrasonicSensor(portS1);

                // get an instance of this sensor in measurement mode
SampleProvider sonicdistance = sonicSensor.getDistanceMode();

                // initialize an array of floats for fetching samples
                float[] sample = new float[sonicdistance.sampleSize()];

                // fetch a sample
                sonicdistance.fetchSample(sample, 0);

                while (buttons.getButtons() != Keys.ID_ESCAPE) {
                        lcddisplay.clear();
                        lcddisplay.drawString("distance: " + sample[0], 0, 1);
                        try {
                                //Thread.sleep(10000);
                        } catch (Exception e) {
                        }

                }
        }
}
```

In the above code, to use the Ultrasonic Sensor, you have to create an instance of `UltrasonicSensor`:

```
// get a port instance
        Port portS1 = ev3brick.getPort("S1");
        // Get an instance of the Ultrasonic EV3 sensor
EV3UltrasonicSensor sonicSensor =
new EV3UltrasonicSensor(portS1);
```

You have to indicate into which port you have plugged the sensor. In the example, Ultrasonic is plugged into Port 1. If your program needs to know the distance from the sensor to any object, then you need to use the method getDistanceMode():

```
                // get an instance of this sensor in measurement mode
                SampleProvider sonicdistance =
                sonicSensor.getDistanceMode();
                // initialize an array of floats for fetching samples
                float[] sample = new float[sonicdistance.sampleSize()];
                // fetch a sample
                sonicdistance.fetchSample(sample, 0);
```

In summary, the ultrasonic sensor helps your EV3 robot measure distances and observe where the objects are located. As you know, the ultrasonic sensor uses the same scientific principle as bats; that is, it measures distances by calculating the time it takes for a sound wave to detect an object and then return. Usually, the larger-sized objects with hard surfaces return the best readings compared with those objects made of soft fabric or those with curved shapes, as the latter are difficult for the sensor to detect.

## Programming Practice with Touch Sensor

So far, your robots haven't known anything about their environment. Now you want to begin practicing with some sensors so that the robot can make some judgments on how best to move. Your task in this section is to add touch and ultrasonic sensors to your robot so that it moves about its environment performing obstacle avoidance maneuvers as necessary. Upon starting, the robot should proceed in a straight line until it is confronted by an obstacle. Once the obstacle is detected, your robot should attempt an avoidance maneuver. The simplest maneuver is to back up, rotate away from the object, and then proceed again. Your robot should continue moving and avoiding objects until the program is terminated by a user (for example, by pressing the ESCAPE button). You should actually produce two different programs: one using the touch sensor only and the other using the ultrasonic sensor only.

In this practice, you will develop a program to place the touch sensor at the front of your robot and write a program so that your robot can move around obstacles. The pseudocode could be the following in which there is a loop continuously asking the sensor if there is a problem and if evasive actions should be taken. In this practice, the version of leJOS that we are using is leJOS NXJ.

```
while true
        Move forward
        if (touch sensor is pressed)
                move backward
                appropriate delay
                turn right
```

The following example program, `ch10p3.java`, illustrates how to achieve this goal.

```
//*********************************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch10p3.java
//This program has the robot move forward until the touch sensor is //activated.
//When the sensor is activated the robot will back up, rotate 90 degrees and continue
moving.
//*********************************************************************************************

import lejos.nxt.Button;
import lejos.nxt.Motor;
import lejos.nxt.SensorPort;
import lejos.nxt.TouchSensor;
import lejos.robotics.navigation.DifferentialPilot;

public class ch10p3 {
        public static void main(String[] args) {
// set up differential pilot and nav path controller to // use for
                // navigation
                DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f, Motor.A,
                                Motor.C);
pilot.setRotateSpeed(60); // rotate speed set slower to // prevent slipping

                // set up touch sensor
                TouchSensor touch = new TouchSensor(SensorPort.S1);

                // wait to begin
                Button.waitForPress();

                // move forward until touch sensor is pressed
                while (!Button.ESCAPE.isPressed()) {
                        pilot.forward();

// if sensor is pressed, stop, rotate 90 degrees // then continue
                        if (touch.isPressed()) {
                                pilot.stop();
                                pilot.travel(-10);
                                pilot.rotate(90);
                        }
                }
        }

}
```

## Programming Practice with Ultrasonic Sensor

In this practice, you will write a program that allows your robot to wander around in an area, avoiding collisions with walls using the ultrasonic sensor. It should start when you press the ENTER key and stop when you press the ESCAPE key, constantly showing the distance covered in its LCD. Also, the code for this program should be able to instruct the robot to back up a certain distance if the range finder detects an obstacle, rotate away from the object, and then proceed again. An example program, `ch10p4.java`, follows, showing you how to achieve this objective.

```
//********************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3 ch10p4.java
//This program will move the robot forward until the ultrasonic //sensor detects an
//object within 25cm. When the object is detected the robot will //back up to 30cm,
//rotate 90 degrees, then continue moving.
//********************************************************************************

import lejos.nxt.Button;
import lejos.nxt.Motor;
import lejos.nxt.SensorPort;
import lejos.nxt.UltrasonicSensor;
import lejos.robotics.navigation.DifferentialPilot;

public class ch10p4 {

        public static void main(String[] args) {

                int distance;

// set up differential pilot and nav path controller to // use for
                // navigation
                DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f, Motor.A,
                                Motor.C);

                // rotate speed set slower to prevent slipping
                pilot.setRotateSpeed(180);

                // set up ultrasonic sensor
                UltrasonicSensor ultraSonic = new UltrasonicSensor(SensorPort.S1);

                // wait to begin
                Button.waitForPress();

                // move forward until distance from object is 25 cm
                while (!Button.ESCAPE.isPressed()) {
                        distance = ultraSonic.getDistance();
                        while (distance > 25) {
                                pilot.forward();
                                distance = ultraSonic.getDistance();
                        }
                        // if object is closer than 25 cm backup to 30 cm
                        if (distance <= 25) {
                                while (distance <= 30) {
                                        pilot.backward();
                                        distance = ultraSonic.getDistance();
                                }

                                // rotate 90 degrees and continue loop
                                pilot.rotate(90);
                        }
                }
        }
}
```

## Summary

In this chapter, you learned about the features of the touch and ultrasonic sensors and how they work. You also became familiar with the usage of the ultrasonic sensor and touch sensor and learned how to apply the leJOS NXJ Java programming language to control and operate the touch sensor and ultrasonic sensor to give the robot the capability of interacting with its environment.

In the next chapter, you will take the wheeled robot that you create and add the light sensor and color sensor to make your robot more aware of its surroundings as it moves around the room.

CHAPTER 11

# Introducing the Light Sensor and Color Sensor

The LEGO Mindstorms EV3/NXT kit comes with four common sensors: the touch sensor, the color sensor, the light sensor, and the ultrasonic sensor. Following through on the coverage of the touch sensor and the ultrasonic sensor in Chapter 10, in this chapter you will add the light sensor and the color sensor to the wheeled robot to increase its awareness of its surroundings as it moves around a room.

In particular, this chapter will cover the following topics:

- Introduction to the light sensor
- Introduction to the color sensor
- Programming practice with the color and light sensors

## Light Sensor

The light sensor and the color sensor are integrated into to the same piece of hardware in NXT 2.0. The light sensor specifically consists of a single tiny lens on the front of the sensor, as seen in Figure 11-1.

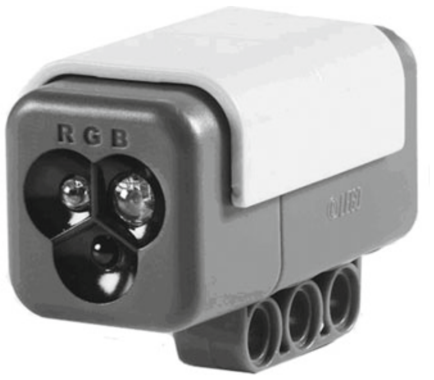


***Figure 11-1.*** *Lego NXT 2.0 light sensor*

The *light sensor* measures the intensity of light captured by the lens and then executes a set of functions. By pointing the light sensor downward, the robot can follow a black line or a line consisting of any other color. By using the sensor, you can also prevent your robot from moving off the edge of the line because light values decrease significantly when an object is far away. In other words, faraway objects do not reflect as much light as nearby objects. Moreover, the light sensor can distinguish dark objects from light objects because less light is reflected back by dark objects.

To use the light sensor, you first create an instance of it using the following constructor:

```
public ColorSensor(SensorPort port);
```

Then, once you have an instance of ColorSensor, use method setFloodlight to turn off color detection so that the ColorSensor detects only the ambient light by using Color.NONE, as follows here. Then use the method getLightValue() to return a number between 0 and 100 in which the brighter the light, the higher the light value.

```
setFloodlight(Color.NONE)
int lightvalue  = getLightValue()
```

The following is an example program that reads the current light value using the light sensor, which is attached to port S3.

```
//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch11p1.java
//an example for light sensor testing
//*******************************************************************

import lejos.nxt.*;
import lejos.robotics.Color;

public class ch11p1 {
        public static void main(String[] args) throws Exception {

                // Get an instance of the color/light sensor of NXT 2.0
ColorSensor lightsensor = new ColorSensor(SensorPort.S1);

                // turn off the color detection, detecting light only
                lightsensor.setFloodlight(Color.NONE);

                LCD.clear();

// keep receiving light value until an Escape button is // pressed
                while (!Button.ESCAPE.isPressed()) {
LCD.drawInt(lightsensor.getLightValue(), 4, 0, 0);
                }

                // clean out the LCD screen
                LCD.clear();
        }
}
```

## Color Sensor

As stated earlier, both the color sensor and light sensor are integrated into the same piece of hardware in Lego NXT 2.0. Similar to the light sensor, the color sensor can also be used to detect different colors. As shown in Figure 11-1, there is a built-in lamp emitting red, blue, or green light from multicolored LEDs on the side of the hardware sensor. By implementing the `ColorDetector` interface, the color sensor can read red, green, and blue color values, or it can identify colors from a palette. To identify simple predefined colors, you can use the `getColorID()` method, which returns an integer representing a color constant.

```
int getColor()
```

These values are predefined in the `lejos.robotics.Color` class. For example, `Color.GREEN` represents a color value in the green spectrum. You can also retrieve RGB values by using the following code:

```
ColorPick cp = new ColorSensor(SensorPort.S1);
Color colorvalue = cp.getColor();
int green = colorvalue.getGreen();
```

The *Color* class can further identify the color spectrum by using the `Color.getColor()` method, which returns a color constant from a palette of colors.

The following is an example program that tests if your color sensor works by being able to identify the different colors in the light.

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch11p2.java
//an example for color sensor testing
//*****************************************************************

import java.util.ArrayList;
import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.nxt.SensorPort;
import lejos.robotics.Color;
import lejos.util.Stopwatch;
import sensors.ColorStruct;
import sensors.SensorControl;
import sensors.UnsupportedSensorException;

/**
 * Demonstrates the use of color sensors
 */

public class ch11p2 {

        // The port to use for color sensing
        private static SensorPort _colorPort = SensorPort.S1;

// Stores the number of characters that fit on one line of the // LCD screen.
        private static int LCD_DISP_WIDTH = 16;
        public static void main(String[] args) {
                try {
                        runColorSenseDemo();
                } catch (UnsupportedOperationException e) {
                        handleError(e, true);
                } catch (Throwable e) {
                        handleError(e, false);
                }
        }

        /**
         * Demonstrate color sensing
         *
         * @throws UnsupportedSensorException
 *             Thrown if _colorPort does not specify a valid
 *  color sending
         *             port
         */
        private static void runColorSenseDemo() throws UnsupportedSensorException {
                LCD.drawString("Press to begin…", 0, 0);
                Button.waitForPress();
                SensorControl sControl = new SensorControl(null, null, _colorPort, null);
                while (!Button.ESCAPE.isPressed()) {
                        ColorStruct cvalue = sControl.getSensedColor();
                        Color cv = sControl.getRGBColor();
                        LCD.clear();
                        LCD.drawString("Color: " + cvalue, 0, 0);
                        LCD.drawString("Sensed:", 0, 1);
                        LCD.drawString(
                                        cv.getRed() + " " + cv.getGreen() + " " +
                                        cv.getBlue(), 0,2);

                }
        }

        /**
         * Prints an error on the LCD screen so that it does not run
 * off the screen,
         * then waits for escape to be pressed to continue.
         *
         * @param message
         *            The message to display on the LCD screen.
         * @param expected
 *            Flag indicating if the exception was expected or
 *        caught as
         *            part of a blanket-catch.
         */
        private static void handleError(Throwable ex, boolean expected) {
                String message;
                if (expected) {
                        message = "ERROR: " + ex.getClass().toString() + ex.getMessage();
                } else {
```

```
                message = "UNEXPECTED ERR: " + ex.getClass().toString() +
                ex.getMessage();
            }

            LCD.clear();
            ArrayList<String> messageSplit = new ArrayList<String>();

            // split the message in 16-character segments
            while (message.length() > LCD_DISP_WIDTH) {
                messageSplit.add(new String(message.substring(0, LCD_DISP_WIDTH)));
                message = message.substring(LCD_DISP_WIDTH, message.length());
            }

            int printRow = 0;
            Stopwatch sw = new Stopwatch();
            // print all of the messages
            for (String msg : messageSplit) {
                LCD.drawString("Here", 0, 4);
                LCD.drawString("CLS:" + ex.getClass().toString(), 0, 5);
                sw.reset();
                while (sw.elapsed() < 1000)
                    Thread.yield();
                LCD.drawString(msg, 0, printRow);
                printRow++;
            }
            LCD.refresh();
            LCD.drawString("Press to exit…", 0, printRow);
            Button.waitForPress();
        }
}
```

## Programming Practice with the Color and Light Sensors

In this programming practice, your task is to design and program a Mindstorms NXT 2.0 robot to navigate around two predefined courses. The first course is an area defined by a black line on a white background, and the second course is equipped with obstacles – that is, two boxes on the black line. Your robot should recognize when an obstacle is blocking its path, go around the obstacle, and then rejoin the path and continue onto the end of the path. Figure 11-2 illustrates the two courses that your robot will explore, in which the red boxes indicate obstacles.

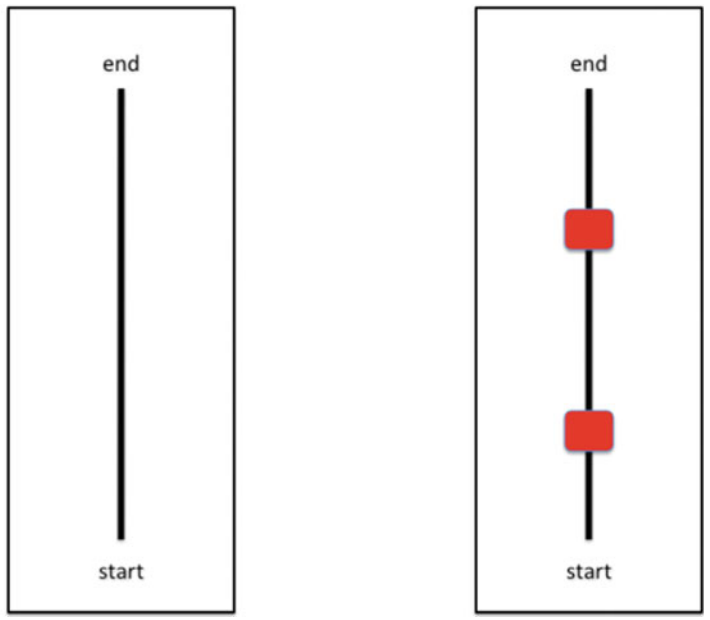


***Figure 11-2.*** *Two courses for programming practice with color/light sensor*

In this programming practice, you will be creating a line-following robot that has the following functionalities:

1. Your robot can follow the black line straightforward from the starting point to the ending point of the line. You don't have to hard-code the color to indicate whether you are following a black track on a white background or a white track on a black background. Instead, your program allows you to pick up the line and background colors at startup and then wait for a button push to start running.
2. Your robot can avoid the obstacles on the way to the ending point, and it can then rejoin the original line smoothly until it reaches the destination point.

The following program, `ch11p3.java`is designed to follow the first course without any obstacles.

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch11p3.java
//This program allows your robot to follow a line.
//If line is lost the robot will rotate left and right at increasing //angles
//until the line is found again.
//*****************************************************************

import lejos.nxt.Button;
import lejos.nxt.ColorSensor;
import lejos.nxt.LCD;
import lejos.nxt.Motor;
import lejos.nxt.SensorPort;
import lejos.robotics.navigation.DifferentialPilot;
```

```
public class ch11p3
{
        public static void main(String[] args)
        {
// set up differential pilot and nav path controller to // use for navigation
                DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f, Motor.A,
                Motor.C);
                pilot.setTravelSpeed(4);

                //set up color sensor
                ColorSensor colorSense = new ColorSensor(SensorPort.S1);
                colorSense.setFloodlight(false);

                //used to store values returned by color sensor
//follow is color robot is to follow, search is value //returned by sensor when searching
                int follow, search;

                //degrees robot will rotate when searching for line
                int rotation;

                //calibrate sensor
                LCD.drawString("Place color sensor\nabove color to follow", 0, 0);
                Button.waitForPress();
                follow = colorSense.getColorID();

                // place robot on start and wait for button press to
// begin main loop
                LCD.clear();
                LCD.drawString("Place robot", 0, 0);
                Button.waitForPress();

                // main loop
// follow line. if line is lost turn left and right to // search for it
                while(!Button.ESCAPE.isPressed())
                {
                        rotation = 5;
search = colorSense.getColorID(); //make sure we // are still on the line

                        //line is found continue forward
                        while(search == follow)
                        {
                                pilot.forward();
                                search = colorSense.getColorID();
                        }

                        //line lost
                        while(search != follow)
                        {
                                pilot.rotate(rotation); //rotate right
                                search = colorSense.getColorID();
                                if(search == follow)
                                break; //found line again exit loop
                        else
                        {
                                pilot.rotate(-rotation * 2); //rotate left back to
                                start then to left position
                                search = colorSense.getColorID();
                                if(search == follow)
                                        break;
//found line again exit loop

                                pilot.rotate(rotation);
//rotate back to center
                        }
                        rotation+=5; //increase angle of rotation // and continue
                        search
                }//end search
        }//end main loop
    }//end main()
}//end ch11p3
```

The following program, ch11p4.java, is designed to follow the second course and avoid any obstacles.

```
//******************************************************************
// Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch11p4.java
// Program that allows your robot to follow a line. It uses the
// ultrasonic
// sensor to detect if an object is in its path while following the // line.
// If an object is detected the robot will leave the line and travel // around
// the object. It will then search for the line and continue
// traveling.
//******************************************************************

import lejos.nxt.Button;
import lejos.nxt.ColorSensor;
import lejos.nxt.LCD;
import lejos.nxt.Motor;
import lejos.nxt.SensorPort;
import lejos.nxt.UltrasonicSensor;
import lejos.robotics.navigation.DifferentialPilot;

public class ch11p4 {

        public static void main(String[] args) {
// set up differential pilot and nav path controller to // use for
                // navigation
                DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f, Motor.A,
                                Motor.C);
                pilot.setTravelSpeed(5);

                UltrasonicSensor ultra = new UltrasonicSensor(SensorPort.S4);
                // set up color sensor
                ColorSensor colorSense = new ColorSensor(SensorPort.S1);
                colorSense.setFloodlight(false);

                // used to store values returned by color sensor
// follow is color robot is to follow, search is value // returned by
                // sensor when searching
                int follow, search;

                // degrees robot will rotate when searching for line
                int rotation;

                // calibrate sensor
                LCD.drawString("Place color sensor\nabove color to follow", 0, 0);
                Button.waitForPress();
                follow = colorSense.getColorID();

                // place robot on start and wait for button press to
// begin main loop
                LCD.clear();
                LCD.drawString("Place robot", 0, 0);
                Button.waitForPress();

                // main loop
// follow line. if line is lost turn left and right to // search for it
                while (!Button.ESCAPE.isPressed()) {
                        rotation = 5;
                        search = colorSense.getColorID();
// make sure we // are still on the// line

                        // line is found continue forward
                        while (search == follow) {
                                // object detected
                                if (ultra.getDistance() <= 10) {
                                        // execute 90 degree turn to
// navigate around object
                                        pilot.rotate(90);
                                        pilot.travel(10);
                                        pilot.rotate(-90);
                                        pilot.travel(30);
                                        pilot.rotate(-90);
                                        search = colorSense.getColorID();

                                        // find line again
                                        while (search != follow) {
                                                pilot.forward();
                                                search = colorSense.getColorID();

// line found, rotate until // following again
                                                if (search == follow) {
                                                        pilot.rotate(90);
                                                        break;
                                                }
                                        }
                                }

                                // continue following line
                                pilot.forward();
                                search = colorSense.getColorID();
                        }

                        // line lost
                        while (search != follow) {
                                pilot.rotate(rotation); // rotate right
                                search = colorSense.getColorID();
                                if (search == follow)
                                        break; // found line again exit loop
                                else {
                                        pilot.rotate(-rotation * 2);
// rotate left back to start

   // then to left position
                                        search = colorSense.getColorID();
                                        if (search == follow)
                                                break;
// found line again exit loop

                                        pilot.rotate(rotation);
// rotate back to center
                                }
                                rotation += 5; // increase angle of
// rotation and continue search
                        }// end search
                }// end main loop
        }// end main()
}
```

## Summary

In this chapter, you learned about the features of the light sensor and color sensor and how these two sensors work. You also became familiar with the use of the light senor and the color sensor and learned how to apply leJOS NXJ JAVA programming to control and operate them to give your robot the capability to interact with its surrounding environment.

In the next chapter, you will begin to learn about behavior programming on the basis of subsumption architecture defined in leJOS. You will also look into how to apply an arbitrator in leJOS Java programming to control and operate the touch sensor, ultrasonic sensor, and color sensor in the order of a set of behaviors designed to give your robot the capability to interact with its surrounding environment.

CHAPTER 12

# Introduction to Behavior Programming

Robots have the ability to achieve a set of human-like behaviors, such as sensing, planning, and acting through using behavior programming. In this chapter, you will learn behavior programming on the basis of the subsumption architecture defined in leJOS. You will also look into how to apply Arbitrator in leJOS JAVA programming to control and operate the touch sensor, ultrasonic sensor, and color sensor in a certain order of a set of behaviors to give the robot the capability to interact with its surrounding environment.

In particular, this chapter will cover the following topics:

- Introduction to behavior programming
- The behavior API functions
- Design pattern of behavior programming
- Programming practice with behaviors

## Introduction to Behavior Programming

*Behavior programming* is based on the *Sense-Plan-Act control model*, and it requires a little more planning before you begin coding. Through sensing, a robot takes in information about its environment via sight, touch, sound, and distance. Through planning, a robot uses sensory information to decide upon an action. Through acting, a robot uses its moving parts to complete the plan. By implementing the Sense-Plan-Act model, it will theoretically make your code easier to understand by other programmers who are familiar with the behavior control model. Moreover, through behavior programming, it becomes easier for you to add or remove specific behaviors from the overall structure without negative repercussion to the rest of your code.

*Behavior-based robot programming* is the act of setting up a series of behaviors to be carried out or goals to be accomplished. Several simple behaviors can be assembled that can theoretically form a very complex, logical model in the real world. Behavior-based robot programming theory was originally inspired by insects and called "hard-wired thinking," as proposed by Rodney Brooks of the MIT Artificial Intelligence Lab in 1983, as illustrated in Figure 12-1.

W. Lu, *Beginning Robotics Programming in Java with LEGO Mindstorms*, DOI 10.1007/978-1-4842-2005-4_12

***Figure 12-1.*** *Insects "hard-wired thinking"*

*Subsumption architecture* is a methodology widely used for programming robots that is heavily associated with behavior-based robotics. Subsumption architecture is a method for decomposing complicated intelligent behavior into many simpler behavioral modules. In turn, these are organized into different layers. Each layer implements a specific goal of the agent, and the higher layers are increasingly more abstract. For instance, a robot's lower layer could be to "avoid an object." By contrast, on top of that layer might be the one to "wander around." The next higher layer could be the one "explore the world," and so on. Each of these layers can access all of the sensor data and generate a set of actions for the actuators. Using this method, the lower layers can function as quickly adapting mechanisms, while the higher layers work to achieve the overall goal.

The concept of behavior programming implemented in Java leJOS follow these four rules:

1. There should be only one behavior active and under the control of the robot at any time.
2. Each behavior has a predefined, fixed priority number.
3. Each behavior has the ability to determine if it should take control or not.
4. The active behavior with a higher priority than any of the other behaviors should take control.

In general, behavior programming of your robot is a problem-solving process, and you could follow these steps in the order shown when designing the program:

1. What do you want the robot to do?
2. How must the robot behave to complete the task?
3. Decompose the complex task into several simple behaviors, including assigning a priority to each behavior, and a behavior with higher priority may stop the current behavior.
4. Create the program using Java programming language.
5. Run the program.

6. Did the robot behave as required? (Did it perform the task correctly?) If not, do the following:
    a. Check the robot first. If there's a problem, can you fix it?
    b. Next, check the program. Problem? Can you fix it?
    c. Finally, go back to the beginning and reread the task. Does your program really tell the robot what it's supposed to do?

## The Behavior API Functions

The Behavior API has one interface and one class, namely, the Behavior interface and Arbitrator class, respectively. The Behavior interface defines the individual behavior classes, including three public methods: `takeControl()`, `action()`, and `suppress()`. Once all of the behaviors are created, an Arbitrator is used to regulate all of these behaviors to determine which one should be activated and at what time. The Arbitrator class and the Behavior interface are located in the `lejos.subsumption` package. Figure 12-2 illustrates how behavior programming works in Java leJOS. As shown in the figure, the three methods in the Behavior interface are very simple. Suppose that your robot has three discrete behaviors, you then need to create three individual classes with each class implementing the Behavior interface. Once these classes are finished, your code should hand the behavior objects off to the Arbitrator. The Arbitrator is found in the package `lejos.subsumption.Arbitrator`, and its constructor is in the following:

```
public Arbitrator(Behavior[] behaviors, boolean returnWhenInactive)
```

This creates an Arbitrator object that regulates when each of the behaviors will become active, where the parameter `behaviors` are used to index the priority of each behavior in the array (for example, `behaviors[0]` has the lowest priority) and if parameter `boolean returnWhenInactive` has a value of true, the program exits when there is no behavior that wants to take control. Otherwise, the program runs until being shut down by pressing the Enter and Escape buttons on the NXT brick. Simply by using a public method `start()`, you can start the arbitration system.

When an Arbitrator object is instantiated, it is given an array of Behavior objects. Once it has these, the `start()` method is called and it begins arbitrating and determining which behavior will become active. The Arbitrator calls the `takeControl()` method on each Behavior object, starting with the object with the highest index number in the behavior array. It works in decreasing priority order until it finds a behavior that wants to take control. If the priority index of this behavior is greater than that of the currently active behavior, the active behavior is then suppressed. After that, the `action()` method is called on the behavior of this index. Therefore, if several behaviors wish to take control, only the highest priority behavior will become active.

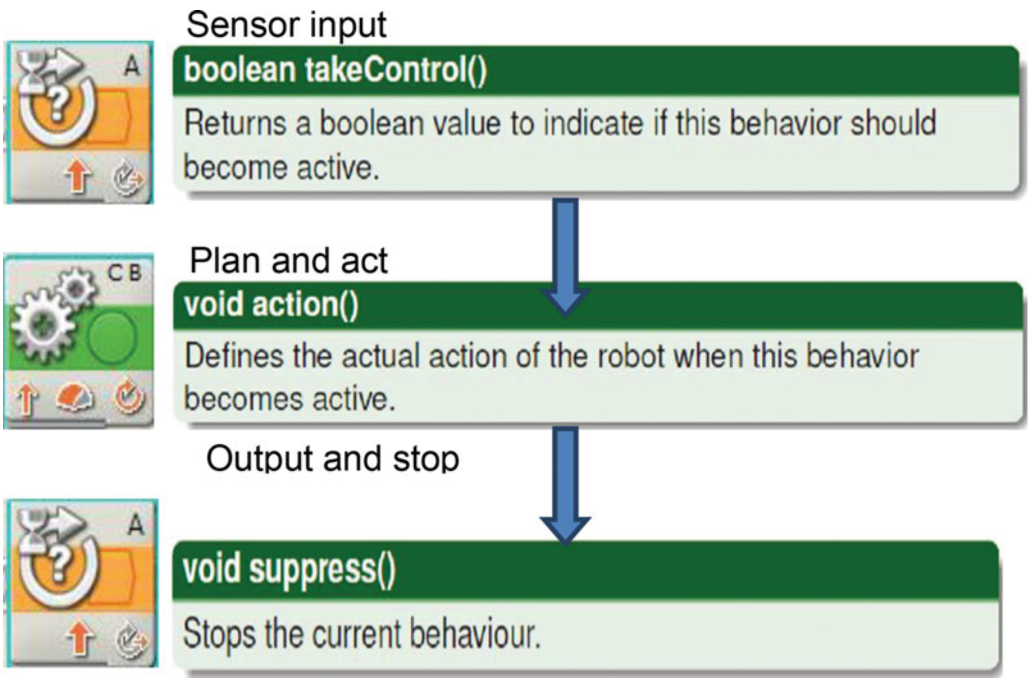

Sensor input
A
boolean takeControl()
Returns a boolean value to indicate if this behavior should become active.
Plan and act
CB
void action()
Defines the actual action of the robot when this behavior becomes active.
Output and stop
A
void suppress()
Stops the current behaviour.

***Figure 12-2.*** *Behavior programming*

## Design Pattern of Behavior Programming

For controlling the performance of the behaviors control system, it is very important that the `action()` method terminates promptly when `suppress()` is called. To achieve this, leJOS actually defines a design pattern for behavior programming. In this design pattern, the Arbitrator contains a monitor thread to cycle through each of the behaviors, checking the `takeControl()` method to see if the behavior should become active or not. It starts with the behavior with the highest index number and works its way down the array. Once it comes across a behavior that should take control, the monitor thread executes `suppress()` on this active behavior and then starts checking each behavior from the top again.

A real-world example and how to use the design pattern concept to solve a problem follows.

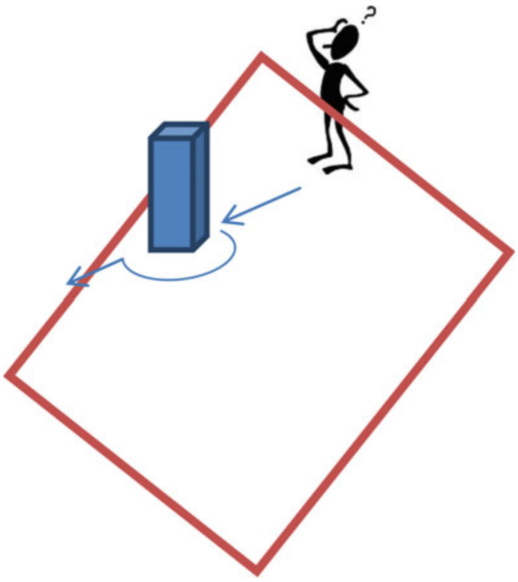
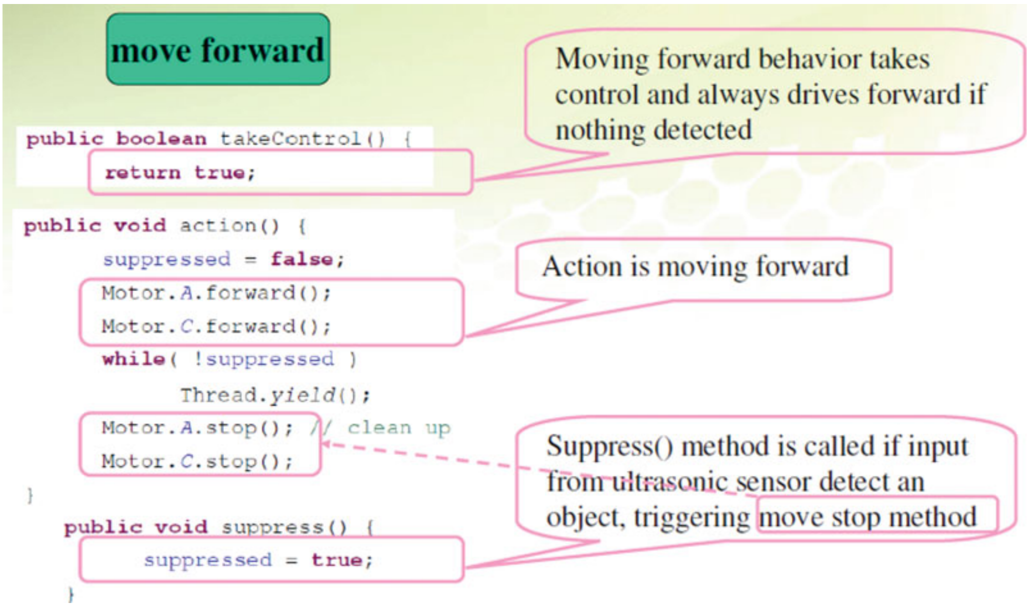


***Figure 12-3.*** *Scenario for design pattern of behavior programming*

As illustrated in Figure 12-3, you want to move forward on a straight line, and when there is an object in front of the robot, you can bypass this object. Given this scenario you have two behaviors:

> *Behavior 1:* Move forward when there is an object detected in front of the robot. Stop the current behavior and start Behavior 2.
>
> *Behavior 2:* Rotate the robot and bypass the object.

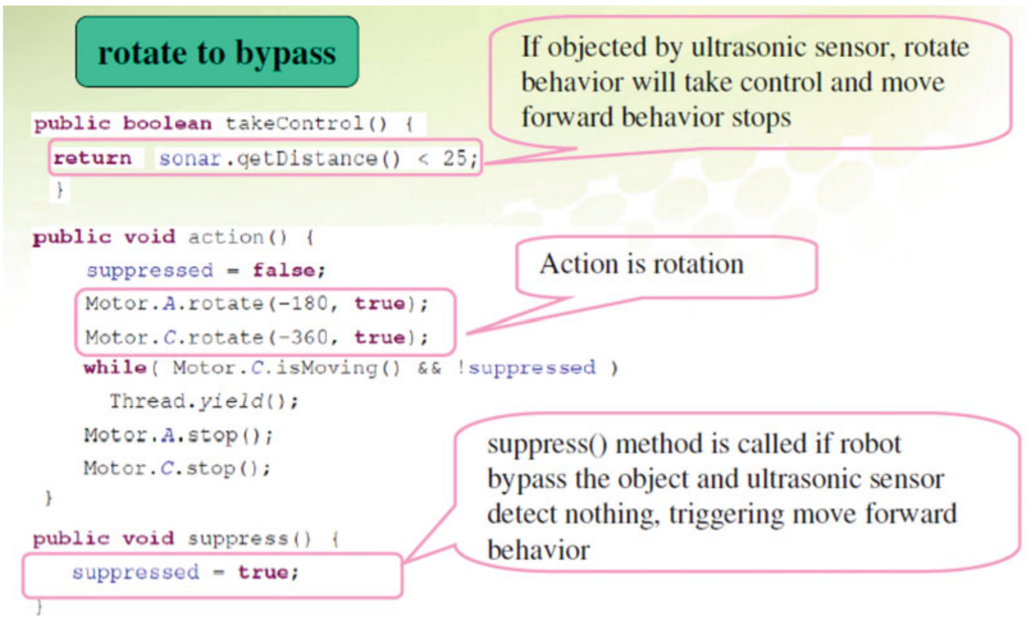
Figure 12-4 illustrates the design pattern of Behavior 1 moving forward.


move forward
Moving forward behavior takes control and always drives forward if nothing detected
Action is moving forward
Suppress() method is called if input from ultrasonic sensor detect an object, triggering move stop method

```
public boolean takeControl() {
      return true;

public void action() {
        suppressed = false;
        Motor.A.forward();
        Motor.C.forward();
        while( !suppressed )
             Thread.yield();
        Motor.A.stop(); // clean up
        Motor.C.stop();
}
   public void suppress() {
        suppressed = true;
   }
```

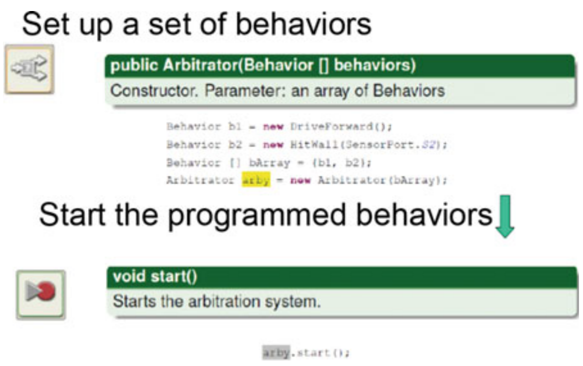
***Figure 12-4.*** *Design pattern of Behavior 1 moving forward*

Figure 12-5 illustrates the design pattern of Behavior 2 rotating to bypass. Figure 12-6 then illustrates how to manage these two behaviors using the design pattern method, including how to set up a set of behaviors and how to start the programmed behaviors.


rotate to bypass
If objected by ultrasonic sensor, rotate behavior will take control and move forward behavior stops
Action is rotation
suppress() method is called if robot bypass the object and ultrasonic sensor detect nothing, triggering move forward behavior

```
public boolean takeControl() {
  return  sonar.getDistance() < 25;
 }

public void action() {
     suppressed = false;
     Motor.A.rotate(-180, true);
     Motor.C.rotate(-360, true);
     while( Motor.C.isMoving() && !suppressed )
       Thread.yield();
     Motor.A.stop();
     Motor.C.stop();
 }

public void suppress() {
    suppressed = true;
}
```

*Figure 12-5. Design pattern of Behavior 2 rotating to bypass*


Set up a set of behaviors
public Arbitrator(Behavior [] behaviors)
Constructor. Parameter: an array of Behaviors
Start the programmed behaviors
void start()
Starts the arbitration system.

```
Behavior b1 = new DriveForward();
Behavior b2 = new HitWall(SensorPort.S2);
Behavior [] bArray = {b1, b2};
Arbitrator arby = new Arbitrator(bArray);
```

```
arby.start();
```

*Figure 12-6. Managing behaviors using design pattern*

To sum up, the idea of using a design pattern to manage behaviors is illustrated in the following example:

1. When an Arbitrator object is instantiated, it is given a set of behaviors in an array.
2. The `start()` method is then called, and the Arbitrator begins determining which behaviors should become active.

3. The Arbitrator calls the `takeControl()` method on each behavior, starting with the object with the highest index number in the array.

4. The Arbitrator then works its way through each of the behaviors until it finds a behavior that wants to take control. When it encounters one, it executes the `action()` method of that behavior once and only once.

5. If two behaviors both want to take control, then only the higher level behavior will be allowed.

## Programming Practice with Behavior Programming

For this programming practice, you will be creating a line-following robot that meets the following with the behavior programming based on subsumption architecture:

*Behavior 1*: Your robot can follow the green line straight from the starting point until it reaches the first obstacle box.

*Behavior 2*: Your robot can then apply the touch sensor to detect this obstacle and then move around the first obstacle; that is, you can have your robot move backward, make an appropriate delay, and then turn right.

*Behavior 3*: After your robot bypasses the first obstacle, it will continue and then go back to the line in which it will follow a blue-color line straight from the first obstacle it encounters until it reaches the second obstacle box.

*Behavior 4*: Your robot then avoids collisions with the second obstacle using the ultrasonic sensor. Once the range finder detects the second obstacle, the robot rotates away from the object and then proceeds.

*Behavior 5*: After avoiding the obstacles on its way to the ending point, the robot then rejoins the original blue line smoothly until it reaches the ending point.

Figure 12-7 illustrates the course for this programming practice.

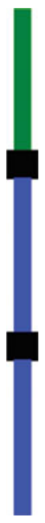


***Figure 12-7.*** *Moving area for your programing practice*

```
//*******************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch12p1.java
//This program implements the leJOS behavior capabilities.
//The robot will begin following a green line. When the touch
//sensor is activated it will move around an object and search for a
//blue line. Then when it travels within 15cm of an object it will //avoid it
//and return to the blue line again.
//*******************************************************************************

import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class ch12p1 {
        public static void main(String[] args) {
                // create a sensors object to send to the behavior
// classes
                Sensors sensors = new Sensors();

                // prompt to begin
                LCD.clear();
                LCD.drawString("Press to begin", 0, 0);
                Button.waitForPress();

                // set up behavior classes
                Behavior b1 = new FollowGreen(sensors);
                Behavior b2 = new TouchAvoid(sensors);
                Behavior b3 = new FollowBlue(sensors);
                Behavior b4 = new SonicAvoid(sensors);

                // create behavior array
                Behavior[] bArray = { b1, b3, b2, b4 };

                // send array to arbitrator and begin
                Arbitrator arby = new Arbitrator(bArray);
                arby.start();
        }
}
//*********************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0
// FollowGreen.java
// This behavior follows a green line
//*********************************************************

import lejos.nxt.Motor;
import lejos.robotics.Color;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.robotics.subsumption.Behavior;

public class FollowGreen implements Behavior {
        private boolean suppressed;
        // used to store values returned by color sensor
        // follow is color robot is to follow, search is value //returned by sensor
        // when searching
        private int follow, search;

        // degrees robot will rotate when searching for line
        int rotation;
        // set up differential pilot and nav path controller to use //for navigation
        private DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f,
                        Motor.A, Motor.C);

        Sensors sensors;

        public FollowGreen(Sensors globalSensors) {
                sensors = globalSensors;
                suppressed = false;
                pilot.setTravelSpeed(4);
                sensors.colorSense.setFloodlight(false);
                follow = Color.GREEN;
        }

        @Override
        public boolean takeControl() {
                return true;
        }

        @Override
        public void action() {
                suppressed = false;
                search = sensors.colorSense.getColorID();

                // move forward while line is green
                while (!suppressed) {
                        Thread.yield();
                        while (search == follow && !suppressed) {
                                pilot.forward();
                                search = sensors.colorSense.getColorID();
                                Thread.yield();
                        }

                        // line lost
                        while (search != follow && !suppressed) {
                                pilot.rotate(rotation); // rotate right
                                search = sensors.colorSense.getColorID();
                                Thread.yield();
                                if (search == follow)
                                        break; // found line again exit loop
                                else {
                                        pilot.rotate(-rotation * 2);
// rotate left back to start
                                                                                // then to left position
                                        search = sensors.colorSense.getColorID();
                                        Thread.yield();
                                        if (search == follow)
                                                break; // found line again //exit loop

                                        pilot.rotate(rotation); // rotate //back to center
                                }
                                rotation += 5; // increase angle of //rotation and continue
                                search
                                Thread.yield();
                        }// end search
                        Thread.yield();
                }
        }

        @Override
        public void suppress() {
                suppressed = true;
        }
}// end FollowGreen

//*********************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0
//      Sensors.java
//Class to store sensor objects to make them available in all //behavior classes
//*********************************************************************************

import lejos.nxt.ColorSensor;
import lejos.nxt.SensorPort;
import lejos.nxt.TouchSensor;
import lejos.nxt.UltrasonicSensor;

public class Sensors {
        ColorSensor colorSense;
        TouchSensor touch;
        UltrasonicSensor sonic;
        boolean touchPressed = false; // used for take control method //in followBlue
                                                                        //behavior

        public Sensors() {
                // set up color, touch, and sonic sensors
                colorSense = new ColorSensor(SensorPort.S1);
                touch = new TouchSensor(SensorPort.S2);
                sonic = new UltrasonicSensor(SensorPort.S4);

        }
}

//***************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 //SonicAvoid.java
//This behavior avoids objects using the ultra sonic sensor
//***************************************************************************
```

```
import lejos.nxt.LCD;
import lejos.nxt.Motor;
import lejos.robotics.Color;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.robotics.subsumption.Behavior;

public class SonicAvoid implements Behavior {
        // set up differential pilot
        private DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f,
                        Motor.A, Motor.C);
        private boolean suppressed;
        private int follow, search;
        Sensors sensors;

        public SonicAvoid(Sensors globalSensors) {
                pilot.setTravelSpeed(4);
                suppressed = false;
                follow = Color.WHITE;
                sensors = globalSensors;
        }

        // take control when within 15 cm of an object and on a blue //line
        @Override
        public boolean takeControl() {
                return (sensors.sonic.getDistance() < 15 && sensors.colorSense
                                .getColorID() == Color.WHITE);
        }

        @Override
        public void action() {
                LCD.drawString("Sonic triggered", 0, 0);
                suppressed = false;

                // stop and move around object
                pilot.stop();
                // execute 90 degree turn to navigate around object
                pilot.rotate(90);
                pilot.travel(10);
                pilot.rotate(-90);
                pilot.travel(30);
                pilot.rotate(-90);

                search = sensors.colorSense.getColorID();

                // find line again
                while (search != follow && !suppressed) {
                        LCD.drawString("HERE I AM", 0, 0);
                        pilot.forward();
                        search = sensors.colorSense.getColorID();
                        // line found, rotate until following again
                        if (search == follow) {
                                pilot.rotate(90);
                                suppressed = true;
                                break;
                        }
                        Thread.yield();
                }
        }

        @Override
        public void suppress() {
                suppressed = true;
        }

}

//***************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 //TouchAvoid.java
//This class represents a robotic behavior when the touch sensor
//is activated
//***************************************************************************

import lejos.nxt.LCD;
import lejos.nxt.Motor;
import lejos.robotics.Color;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.robotics.subsumption.Behavior;

public class TouchAvoid implements Behavior {
        // set up differential pilot
        private DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f,
                        Motor.A, Motor.C);
        private boolean suppressed;
        private int follow, search;
        Sensors sensors;

        public TouchAvoid(Sensors globalSensors) {
                pilot.setTravelSpeed(4);
                suppressed = false;
                follow = Color.WHITE; // color to follow when
// relocating line
                sensors = globalSensors;
        }

        @Override
        public boolean takeControl() {
                return sensors.touch.isPressed(); // behavior takes
// control when touch
                                                                                              //
sensor is pressed
        }
        @Override
        public void action() // avoid the object
        {
                LCD.drawString("BUTTON IS PRESSED", 0, 0);
                sensors.touchPressed = true; // set flag in sensor //class so robot knows

// to follow different color later
                suppressed = false;

                // moves to avoid object
                pilot.stop();
                pilot.travel(-15);
                // execute 90 degree turn to navigate around object
                pilot.rotate(90);
                pilot.travel(10);
                pilot.rotate(-90);
                pilot.travel(30);
                pilot.rotate(-90);

                search = sensors.colorSense.getColorID();

                // find line again
                while (search != follow && !suppressed) {
                        // LCD.clear();
                        pilot.forward();
                        search = sensors.colorSense.getColorID();
                        LCD.drawString("search: " + Integer.toString(search), 0, 0);
                        LCD.drawString("follow: " + follow, 0, 1);

                        // line found, rotate until following again
                        if (search == follow) {
                                pilot.rotate(90);
                                suppressed = true;
                                break;
                        }
                        Thread.yield();
                }
        }

        @Override
        public void suppress() {
                suppressed = true;
        }

}

//*******************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0
//FollowBlue.java
//This behavior follows a blue line
//*******************************************************************
```

```
import lejos.nxt.ColorSensor;
import lejos.nxt.Motor;
import lejos.nxt.SensorPort;
import lejos.robotics.Color;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.robotics.subsumption.Behavior;

public class FollowBlue implements Behavior {
        private boolean suppressed;
        // used to store values returned by color sensor
        // follow is color robot is to follow, search is value //returned by sensor
        // when searching
        private int follow, search;

        // degrees robot will rotate when searching for line
        int rotation;
        // set up differential pilot and nav path controller to use //for navigation
        private DifferentialPilot pilot = new DifferentialPilot(4.32f, 12.2f,
                        Motor.A, Motor.C);

        Sensors sensors;

        public FollowBlue(Sensors globalSensors) {
                sensors = globalSensors;
                suppressed = false;
                pilot.setTravelSpeed(4);
                sensors.colorSense.setFloodlight(false);
                follow = Color.WHITE;
        }

        @Override
        public boolean takeControl() {
                return sensors.touchPressed; // we want this to be the default behavior

// after the touch sensor has been

// pressed
        }

        @Override
        public void action() {
                suppressed = false;
                search = sensors.colorSense.getColorID();

                // move forward if the line is the right color
                while (!suppressed) {
                        Thread.yield();
                        while (search == follow && !suppressed) {
                                pilot.forward();
                                search = sensors.colorSense.getColorID();
                                Thread.yield();
                        }
                        // line lost
                        while (search != follow && !suppressed) {
                                pilot.rotate(rotation); // rotate right
                                search = sensors.colorSense.getColorID();
                                Thread.yield();
                                if (search == follow)
                                        break; // found line again exit loop
                                else {
                                        pilot.rotate(-rotation * 2);
// rotate left back to start

// then to left position
                                        search = sensors.colorSense.getColorID();
                                        Thread.yield();
                                        if (search == follow)
                                                break;
// found line again exit loop

                                        pilot.rotate(rotation);
// rotate back to center
                                }
                                rotation += 5; // increase angle of //rotation and continue
search
                                Thread.yield();
                        }// end search
                        Thread.yield();
                }
        }

        @Override
        public void suppress() {
                suppressed = true;
        }
}// end FollowBlue
```

## Summary

In this chapter, you learned how to do behavior programming using the subsumption architecture defined in leJOS. You also learned how to apply Arbitrator in the leJOS Java programming language to control and operate the touch sensor, ultrasonic sensor, and the color sensor in in a certain order of a set of behaviors to give the robot the capability to interact with its surrounding environment.

In the next chapter, you will learn the basic concepts of Java leJOS multithreading programming and then study how to apply it to control and operate the color sensor, touch sensor, and ultrasonic sensor to give your robot the capability to interact with its surrounding environment.

CHAPTER 13

# Multithreading Programming with Java leJOS

Multithreading is a very well-known programming feature, which allows you to execute multiple jobs at the same time. When developing programs for robots, you need to consider this programming feature as the basis of your programming architecture.

In this chapter, you will learn the basic concepts of Java leJOS multithreading programming and then study how to apply it to control and operate the color sensor, touch sensor, and ultrasonic sensor to give your robot the capability of interacting with its surrounding environment.

In particular, this chapter will cover the following topics:

- Introduction to the thread concept
- How to use the class thread in leJOS
- Programming practice with multithreading in Java leJOS

## The Thread Concept

Normally, when you start developing with Java and leJOS, you create programs that execute a set of operations in order, one by one. For example, see the following program structure where the program completes Task 1 first, then Task 2, Task 3, and so on until it reaches the last program task, Task 10:

```
Program:
        Task 1
        Task 2
                Task 3
                ...
                Task 10
```

If you use Java multithreading, however, then you could execute the robot's tasks in parallel as follows, where the program executes all 10 tasks at the same time:

```
Task 1  Task 2  Task 3  Task 4  Task 5  Task 6  Task 7  Task 8  Task 9  Task 10
```

W. Lu, *Beginning Robotics Programming in Java with LEGO Mindstorms*, DOI 10.1007/978-1-4842-2005-4_13

This is very similar to the operations that our bodies execute in parallel, such as respiration, blood circulation, digestion, thinking, and walking. In addition, the human brain processes information from your body sensors in parallel, including seeing, touching, smelling, tasting, and hearing.

Many current programming languages allow programmers to specify activities to execute concurrently. This is achieved by using a thread. A *thread* in Java is the minimum unit in terms of parallel processing. Therefore, when designing a robot program using threads, you should consider the following ideas:

1. A thread to manage wireless communication
2. A thread to manage the locomotion subsystem
3. A thread to manage the robot arming
4. A thread to manage the robot's senses

The robot's design process is iterative, because it is a complex task. This is different from traditional a design process that performs one action at a time, proceeding to the next action only after the previous one has finished. Here is where threading technology is very useful. When you want to incorporate a new feature in the future, you simply need to add a new task, a thread, or select the old thread and update it with new capabilities.

## Using Threads in leJOS

In the following example, `ch13p1.java`, you will see how to develop your first thread, which is managed by a main program. This program can print the message "`Hello World`," and the value of the variable `i` on the NXT 2.0 LCD. (`i` counts how many messages are printed on the screen.) This program executes the thread `HelloWorldThread` until you push the ESCAPE button on the NXT 2.0 brick.

```
//****************************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch13p1.java
//This program executes the thread HelloWorldThread to print the //message
//"Hello World" and the value of a counter to count how
//many messages printed on the screen.
//****************************************************************************

import lejos.nxt.Button;
import lejos.nxt.LCD;

public class ch13p1 {
        private static HelloWorldThread hwt;

        public static void main(String[] args) {
                int i = 0;

                hwt = new HelloWorldThread();
                hwt.start();
                try {
                        while (!Button.ESCAPE.isPressed()) {
                                LCD.drawString("Hello World: " + i, 0, 0);
                                ++i;
                        }
                } catch (Exception ex) {
                } finally {
                        System.exit(0);
                }
        }
}

import lejos.nxt.*;

public class HelloWorldThread extends Thread {
        private int i = 0;

        public HelloWorldThread() {
        }

        public void run() {
                while (true) {
                        LCD.drawString("Hello World:", 0, 0);
                        LCD.drawInt(i, 0, 1);
                        LCD.refresh();
                        i++;
                }
        }
}
```

To understand multithreading code, it is very important that you know about the life cycle of a thread. The possible states of a thread include the following:

1. Start
2. Sleep
3. Interrupted
4. Yield
5. Join
6. Interrupt

Typically, thread class in Java leJOS has three methods: `Start()`, `IsAlive()`, and `Sleep()`. I will now introduce each method individually.

## Method start()

When you define a thread to execute a task with your robot, you simply need to start it by using a piece of code like the following:

```
private static HelloWorldThread hwt;

hwt = new HelloWorldThread();
hwt.start();
```

## Method isAlive ()

In many cases, when working with multiple threads, it is necessary to know if a thread is still alive or not by using the following code:

```
public void run() {
        while (true) {
                LCD.drawString("Hello World:", 0, 0);
                LCD.drawInt(i, 0, 1);
                LCD.drawString("" + this.isAlive(), 0, 2);
                LCD.refresh();
        }
}
```

## Method sleep ()

In many other cases, it is really useful to generate a timed pause to make sure that other tasks have completed by using the sleep method in the following code:

```
import lejos.nxt.*;
public class SleepDemo {
private static String messages[] = {
        "Java leJOS NXJ",
        "Java leJOS PC API",
        "Java leJOS Mobile API",
        };
public static void main(String[] args) throws InterruptedException {
        for(int i=0;i< messages.length; i++) {
Thread.sleep(1000);
                System.out.println(Messages[i]);
                }
        }
}
```

## Practice with Multithreading in Java leJOS

In this example program, `ch13p2.java`, you will finish a multithreading program called *Music* so that your program can play the music and print a sequence number on the LCD screen at the same time.

```
//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 ch13p2.java
//This program demonstrates a threaded song/counter program
//*****************************************************************

import lejos.nxt.Button;

public class ch13p2 {
        private static Music mThread;
        private static Counter cThread;

        public static void main(String[] args) {
                mThread = new Music();
                cThread = new Counter();

                mThread.start();
                cThread.start();
                try {
                        while (!Button.ESCAPE.isPressed()) {
                                Thread.yield();
                        }
                        System.exit(0);
                } catch (Exception e) {
                }
        }
}

//*****************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 Music.java
//Play a "song" consisting of predefined beeps on a thread
//*****************************************************************

import lejos.nxt.Sound;

class Music extends Thread {
        public void run() {
                short Low_G = 392, B_Flat = 470, D = 588, C = 523, E_Flat = 627, F = 697, G
= 784, A_Flat = 830;
```

```
                short[] note = {a C, 600, G, 100, F, 100, G, 400, C, 400, 0, 600,
                                A_Flat, 100, G, 100, A_Flat, 200, G, 200, F, 400, B_Flat, 600,
                                F, 100, E_Flat, 100, F, 400, B_Flat, 400, Low_G, 600, G, 100,
                                F, 100, G, 200, F, 200, E_Flat, 200, D, 200, E_Flat, 600, D,
                                100, E_Flat, 100, F, 600, E_Flat, 100, F, 100, G, 200, F, 200,
                                E_Flat, 200, D, 200, C, 400, A_Flat, 400, G, 1400, A_Flat, 100,
                                G, 100, F, 100, G, 1400
                };

                for (int i = 0; i < note.length; i += 2) {
                        short w = note[i + 1];
                        Sound.playTone(note[i], w);
                        Sound.pause(w);
                }
        }
}

//*********************************************************************
//Wei Lu Java Robotics Programming with Lego EV3/NXT2.0 Counter.java
//Continuously write an incrementing integer to the LCD
//*********************************************************************

import lejos.nxt.LCD;

class Counter extends Thread {
        public void run() {
                for (int i = 0; i < 1000; ++i) {
                        LCD.drawInt(i, 0, 2);
                        LCD.refresh();
                        try {
                                Thread.sleep(1000);
                        } catch (Exception e) {
                        }
                }
        }
}
```

In Chapter 11, you created a line-following robot that could follow the green line straight forward from the starting point to the ending point of a line. The robot that you created also could avoid obstacles along the way to the ending point and could rejoin the original line smoothly until it reached the ending point.

In this second practice with multithreading programming, you create a very similar robot to the one that you created in Chapter 11. The main goal of the robot in this practice is to follow a green line, as illustrated in Figure 13-1, and if it discovers an obstacle, it stops. You have to use a leJOS multithreading programming to implement this line-following robot.

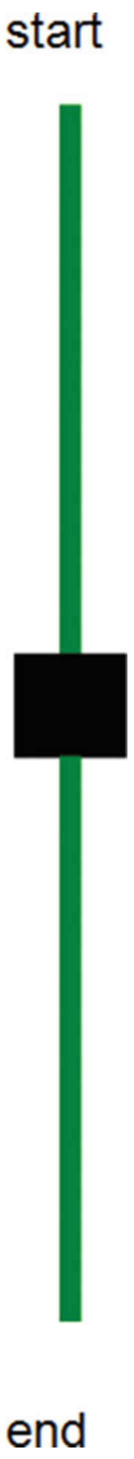


***Figure 13-1.*** *Programming the line-following robot with multithreading*

In analyzing the tasks involved, you will find that there are two major processes in this program: (1) follow green lines and (2) discover obstacles en route. Based on this analysis, you can design and develop a scalable solution with threads: implementing a separate thread used to follow a green line, implementing another separate thread used to detect obstacles, and implementing one more class to exchange data among threads.

```
import lejos.nxt.*;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.robotics.navigation.NavPathController;

public class ch13p3 {

        private static data DE;
        private static line LFObj;
        private static object ODObj;
        static double diam = 2.8;
        static double trackwidth = 15.5;
        static DifferentialPilot pilot = new DifferentialPilot(diam / 2.54,
                        trackwidth / 2.54, Motor.A, Motor.C);
        static UltrasonicSensor sonic = new UltrasonicSensor(SensorPort.S4);
        NavPathController nav = new NavPathController(pilot);// Attaches NavPath

        public static void main(String[] args) {
                DE = new data();
                ODObj = new object(DE, pilot);
                LFObj = new line(DE, pilot);
                ODObj.start();
                LFObj.start();
                while (!Button.ESCAPE.isPressed()) {
                }
                LCD.drawString("Finished", 0, 7);
                LCD.refresh();
        }
}

public class data {
        // ObstacleDetector
        private boolean obstacleDetected = false;
        // Robot has the following commands: Follow Line, Stop
        private int CMD = 1;

        public data() {
        }

        public boolean isObstacleDetected() {
                return obstacleDetected;
        }

        public void setObstacleDetected(boolean obstacleDetected) {
                this.obstacleDetected = obstacleDetected;
        }
        public int getCMD() {
                return CMD;
        }

        public void setCMD(int cMD) {
                CMD = cMD;
        }
}

import lejos.nxt.*;
import lejos.robotics.Color;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.util.Stopwatch;

public class line extends Thread {
        data DEObj;
        private ColorSensor cs;
        private UltrasonicSensor us;
        private DifferentialPilot pilot;
        static boolean leftfirst = true;

        public line(data DE, DifferentialPilot pi) {
                DEObj = DE;
                cs = new ColorSensor(SensorPort.S3);
                us = new UltrasonicSensor(SensorPort.S4);
                pilot = pi;
        }

        public void run() {
                if (DEObj.getCMD() == 1) {
                        if (cs.getColorID() == Color.GREEN) {
                                pilot.forward();
                        } else {
                                pilot.travel(1.5);
                                int found = 0;
                                LCD.clear(6);
                                double degrees = 10;
                                double look = 0, i = 1;
                                if (leftfirst != true) {
                                        i = 1 * i;
                                }
                                LCD.drawString("Looking", 0, 6);
                                Stopwatch clock = new Stopwatch();
                                clock.reset();
                                while (clock.elapsed() < 30000) {
                                        look = degrees * i;
                                        pilot.rotate(look);
                                        pilot.stop();
                                        int batman = cs.getColor().getColor();
                                        if (batman == Color.GREEN) {
                                                LCD.clear(6);
                                                LCD.drawString("Found it!", 0, 6);
                                                found = 1;
                                                break;
                                        } else if (batman == Color.BLACK) {
                                                found = 2;
                                                break;
                                        } else {
                                                look = look * 2 * 1;
                                                pilot.rotate(look);
                                                batman = cs.getColor().getColor();
                                                if (batman == Color.GREEN) {
                                                        pilot.stop();
                                                        LCD.clear(6);
                                                        LCD.drawString("Found it!", 0, 6);
                                                        found = 1;
                                                        break;
                                                } else if (batman == Color.BLACK) {
                                                        found = 2;
                                                        break;
                                                }
                                                pilot.rotate(look * 1);
                                        }
                                        i++;
                                }
                                if (look < 0) {
                                        leftfirst = false;
                                } else {
                                        leftfirst = true;
                                }
                        }
                } else {
                        pilot.stop();
                }
        }
}

import lejos.nxt.*;
import lejos.nxt.ColorSensor.Color;
import lejos.robotics.navigation.DifferentialPilot;

public class object extends Thread {
        private data DEObj;
        private UltrasonicSensor us;
        private ColorSensor cs;
        private final int securityDistance = 15;
        private DifferentialPilot pilot;

        public object(data DE, DifferentialPilot pi) {
                DEObj = DE;
                us = new UltrasonicSensor(SensorPort.S4);
                cs = new ColorSensor(SensorPort.S3);
                pilot = pi;
        }

        public void run() {
                while (true) {
                        if (us.getDistance() > securityDistance) {
                                DEObj.setCMD(1);
                        } else {
                                DEObj.setCMD(0);
                                pilot.rotate(80);// left
                                Motor.B.rotateTo(90);
                                while (us.getDistance() < 15) {
                                        pilot.forward();
                                }
                                pilot.travel(7);// offline
                                pilot.rotate(85);// back on track
                                pilot.travel(2);//
                                while (us.getDistance() < 30) {
                                        pilot.forward();
                                }
                                pilot.travel(8);
                                pilot.rotate(85);
                                pilot.setTravelSpeed(1);
                                while (cs.getColor().getColor() != Color.GREEN) {
                                        pilot.forward();
                                }
                                pilot.stop();
                                pilot.setTravelSpeed(2.25);
                                pilot.travel(2);
                                pilot.rotate(85);// left
                                Motor.B.rotateTo(0);
                                DEObj.setCMD(1);
                        }
                }
        }
}
```

## Summary

In this chapter, you learned about the basic concepts of Java leJOS multithreading programming and how to apply these concepts to control and operate the color sensor, touch sensor, and ultrasonic sensor to give your robot the capability of interacting with its surrounding environment.

After reading this book and completing all of the programming practice, take your robot programming to the next level. For example, try directing your robot remotely by developing Android applications to manage and control a Lego EV3 device and implementing Java applications that make wireless connections among various Lego EV3 devices.

# Index

W. Lu, *Beginning Robotics Programming in Java with LEGO Mindstorms*, DOI 10.1007/978-1-4842-2005-4