Introduction to GPS, Part 1

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Introduction 2
2. What is GPS? 5
3. Why Java for GPS? 9
4. Build environment 11
5. Sample application 13
6. User interface and options for the sample application 15
7. Summary 23

Section 1. Introduction

Should I take this tutorial?

This tutorial, the first in a three-part series, is designed for developers and product managers who are evaluating the Java programming language as an option for use in Global Positioning System (GPS) applications. More generally, anyone familiar with serial communications in another programming language such as C or BASIC, but who is interested in learning how to achieve serial communications by using the Java programming language, will find the sample application a useful exercise. Familiarity with the Java programming language is recommended. Tutorial resources on page 23 contains links to useful primers on the Java language if you need to brush up your skills.

This tutorial is a prerequisite for subsequent tutorials in this series. Upcoming tutorials discuss GPS concepts in further depth as well as look at the task of porting the sample Java application from the desktop to a portable computing environment.

What is this tutorial about?

This tutorial explores GPS and the Java programming language's applicability to GPS applications. Basic GPS concepts serve as a foundation for the sample application. In addition to the GPS topics, this tutorial presents a high-level discussion of Java architecture with respect to the Java Native Interface (JNI) and platform-specific issues related to physical device communications. The tutorial works with the Java Communications API package for serial communications with a GPS unit. The sample application is also designed to be portable; therefore, the application employs the Java Abstract Windowing Toolkit (AWT) for the user interface.

- What is GPS? on page 5 -- This section discusses the basics of GPS, including information on the official U.S. government agencies that have authority over the system and common applications and devices available for GPS. Subsequent panels examine the data protocols available for GPS (the tutorial's sample application optionally parses out National Marine Electronics Association, or NMEA, sentences), and the merits of Java for development of GPS applications. GPS applications are inherently portable and demand a versatile programming environment such as the Java environment.
- Why Java for GPS? on page 9 -- This section shifts gears and delves into the development environment of choice, the Java programming language. It explores JNI, which enables the Java programming language to leverage native resources and talk to a portable GPS unit, and the `javax.comm` package, which Sun Microsystems provides as a reference implementation of a JNI solution to device communications. The sample application employs this package. This discussion of the package examines the prescribed communications driver architecture.
- Build environment on page 11 -- This section walks through the acquisition, installation, and verification of the proper build environment required for the tutorial's sample application. The Tools on page 4 panel contains links for each of the tools used in the tutorial.
- Sample application on page 13 -- The tutorial's sample application is constructed step-by-step in the this section.
- User interface and options for the sample application on page 15 -- This section describes the user interface and the major options available to the user.

Source code map

This tutorial's sample application demonstrates serial communications, Java graphical user interface programming, and GPS NMEA sentence parsing. The full source code is available for download from Tools on page 4 . Here are the code snippets discussed in this tutorial:

- Verifying the build environment -- The `verifybuildenv`class exercises core functionality of the Java Communications API, verifying the build environment.
- `Ibmdwgps.properties` -- This property file specifies required runtime parameters.
- `Property access` -- This code snippet demonstrates loading a properties file into the System object's properties collection.
- `User interface` -- This code creates the graphical user interface via the Java programming language's AWT.
- `Action handler` -- This routine processes events the user initiates.
- `Start option` -- The Start routine initiates communications with the serial port resource.
- `Port configuration` -- Serial ports have a variety of modes and parameters requiring configuration. This code demonstrates the steps necessary to prepare the port for use.
- `Java I/O streams` -- Java code "talks" to other systems by using a mechanism known as *streams*. This code sets up the streams necessary for interaction with the serial port.
- `Serial event` -- This routine handles data that arrives at the port's input stream.
- `NMEA sentence parsing` -- NMEA sentences represent various data from the GPS unit. This code employs a simple parsing technique to handle the NMEA message.
- `Stop option` -- The `Stop` routine closes the communications resources.
- `Save option` -- The `Save` routine prompts the user for a filename to save information into.

Some helpful terms

- GPS -- Global Positioning System, a networked system providing longitude, latitude, and elevation information to specially designed receivers.
- JDK -- Java Developers Kit, otherwise known as the Java SDK. This is the core Java toolset available from Sun Microsystems.
- Communications port -- Physical connector and circuitry in a computer used for the purpose of transferring information to another computer or device. Common examples include serial and parallel ports on personal computers.
- Portability -- The term given to software characterizing the ease with which you can convert an application from one platform to another; for example, converting a program that was written on the Windows platform and enabling it to run on a UNIX platform.
- Cross-platform development -- The practice of employing one operating system and/or hardware combination to develop applications for another platform. Developing a Palm OS application on a Linux computer is an example of cross-platform development.
- Data protocol -- The rules computers use when talking to one another. Common examples are found in Electronic Data Interchange (EDI), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), X-Modem, etc.

Tools

The following resources are required to complete this tutorial:

- Java SDK (JDK) -- The Java SDK is the core Java development platform required for any Java application. You must install this to build the sample application. You can find the JDK at *http://java.sun.com/j2se*.
- Java Communications API -- You can find the Java package, `javax.comm`, used in the sample application for communicating with the GPS unit, at *http://java.sun.com/products/javacomm/index.html*. Follow the steps outlined below in the Build environment on page 11 section to ensure proper installation of this package.
- Tutorial sample code -- You can find the complete code to the tutorial's sample application at *http://www.palm-communications.com/ibmdw* .
- GPS unit -- The tutorial's sample application demonstrates communications with a consumer GPS unit available at *http://www.garmin.com/mobile/*. Any unit capable of producing NMEA sentences over a serial communications port is sufficient.

About the author

After his college basketball career came to an end without a multiyear contract to play for the L.A. Lakers, Frank Ableson shifted his focus to computer software design. He enjoys solving complex problems, particularly in the areas of communications and hardware interfacing. When not working, he can be found spending time with his wife Nikki and their children. You can reach Frank at *frank@cfgsolutions.com*.

Section 2. What is GPS?

Background on GPS

GPS is a U.S. government-operated network of earth-orbiting satellites (space vehicles) and ground control stations. This network provides time and position information to receiving stations and devices around the globe. There are approximately 24 active satellites participating in this network at any point in time. Each GPS satellite continually transmits position information via a spread-spectrum signal. To obtain an accurate location and time calculation, four of these satellites must be in range of the receiver.

There are two varieties of positioning service: *Standard* and *Precise*.

- The Standard Positioning Service (SPS) is freely available to the public. The quality of the positioning and timing information may be degraded without notice for security purposes. The information is generally suitable for all but the most demanding navigation exercises.
- The Precise Positioning Service (PPS) is available to select organizations by permission from the U.S. government. The PPS is a highly accurate positioning and timing service suitable for use in military and other applications where the highest degree of accuracy is required. The U.S. government protects this valuable positioning information via cryptography and provides it on an "as-needed" basis.

The Federal Radionavigation Plan (FRP), a product of the U.S. Department of Defense and the U.S. Department of Transportation, outlines GPS's requirements, features, capabilities, and restrictions. This plan is the authoritative document regarding GPS. Tutorial resources on page 23 contains a link to the current plan document and related materials.

Common GPS applications

The most common GPS application category is navigation. You can find GPS receivers in watercraft, airplanes, and automobiles. Often, these applications combine GPS location information with a mapping service to provide precise, easy-to-follow navigation guidance. For example, many rental cars include a navigation system based on GPS. These in-car navigation systems even offer drivers auditory cues, informing them of upcoming turns, exits, and intersections.

GPS receivers are well on their way to becoming a popular sportsman staple. In addition to providing assistance in navigation, GPS units allow a user to mark a specific location, recording it for later reference. For example, a fisherman can mark the spot where he "caught the big fish" in the middle of the lake. On a subsequent fishing expedition, the fisherman can then relocate this recorded "waypoint" in hopes of landing yet another prize catch.

A hiker can prepare for an upcoming trip by loading a "route" into a personal GPS device, marking key landmarks of interest. During the course of the hiking expedition, the GPS unit displays Time of Arrival (TOA) information along with other pertinent positional information such as bearing and elevation.

Beyond navigation, you can use GPS for precise measurement, such as surveying, or the measurement of velocity and position of various objects, such as fault lines. GPS technology

is ideal for management and tracking of mobile assets, such as vehicles and construction equipment.

Tutorial resources on page 23 provides links to popular GPS applications and studies.

GPS devices

In only a few years' time, the GPS receiver has gone from a high-end tech gadget to a moderately priced commodity available in most electronics stores. The devices range from simple position indicators to full-featured, color navigation assistants, complete with street maps and PC interfaces. Early generations of GPS/mapping solutions operated only with the assistance of a personal computer, as the GPS receiver itself had no built-in user interface.

Unlike cell phones and pagers, a GPS receiver cannot operate without a view of the sky, as the GPS signals do not penetrate the walls and ceilings of homes and office buildings. For this reason, where you decide to mount a GPS receiver is crucial to its proper operation.

While automotive navigation GPS units can draw power from the car's electrical system, portable GPS receivers, such as ones a hiker or backpacker uses out on a hike, require the use of batteries. This requirement is important and you should consider it whenever taking a trip and relying on the availability of the GPS receiver.

The next panel examines some of the GPS devices available on the market.

Real-world GPS devices

Garmin and Magellan are arguably the most well-known GPS manufacturers. Their product lines include devices for aviation, automotive, and marine vehicles; handheld units; telephones; and even OEM offerings. OEM products are suitable for embedding GPS functionality into a custom design, without all of the molded plastic and software functionality found in a consumer product. The devices from Garmin, Magellan, and others provide for basic GPS navigation facilities such as waypoints, routes, tracks, elevation, bearing, speed, and more.

The Earthmate GPS receiver, available from DeLorme, is geared for in-vehicle navigation. The Earthmate product is an example of a device offering PC interfacing capabilities. This model is designed for use with software running on a laptop or Palm OS device.

While the Earthmate offers a proprietary software interface only, many GPS devices support both a manufacturer custom data protocol and industry-standard protocols. The device used in the sample application was selected precisely for its communications capabilities. These capabilities include an easy-to-use serial interface with a DB-9 connector and support for a variety of data protocols. This "open" design makes the device perfectly suited for portable applications including laptops, PDAs, and even embedded controllers for custom-purpose applications.

GPS data protocols

The primary activity of a GPS receiver is the acquisition of positional information from the GPS network. However, there are additional resources beyond the network of space satellites available to the GPS receiver. *Differential GPS* (DGPS) information provides additional positional information that the receiver uses to gain a more precise location measurement. *Beacons* provide receivers with DGPS signals. Maritime applications commonly deploy these beacons near the coastline and in harbors to promote safe passage of seagoing vessels. These beacons are essentially high-tech lighthouses.

GPS receivers acquire beacon signals via a protocol known as the Radio Technical Commission for Maritime Services (RTCM). RTCM is a DGPS standard protocol, used as an input data source to a GPS receiver. The GPS receiver typically receives this information from another device, known as a *beacon receiver*. Another form of differential data designed to augment the accuracy of a GPS receiver is the Wide Area Augmentation System (WAAS). WAAS is gaining acceptance in the civil aviation industry.

Once the GPS receiver has processed/processes the data sources, it is often necessary to disseminate the standard positional information to other applications. Each GPS receiver vendor may offer a special communication and protocol feature set; however, there is a common data format -- NMEA.

NMEA sentence format, part 1

The NMEA protocol calls for information to be transmitted in "sentences" at 4800 baud. Here is an example NMEA sentence:

```
$GPGLL,4338.581,N,07015.101,W,170110,A*3D
```

The NMEA sentence is broken into the following format:

"$" marks the beginning of the sentence. The characters *GP* indicate that the message is from the GPS unit. In NMEA parlance, these two characters identify the "talker" in the conversation. The next three letters indicate the type of NMEA sentence. For example, *GLL* above refers to geographic position, longitude, and latitude.

This format, $GPXXX, represents the standard prefix to NMEA sentences, but not the only data format available.

NMEA sentence format, part 2

In addition to standard GPS messages, the NMEA format accommodates the use of manufacturer-specific messages as well.

A sentence beginning with a *P* indicates a manufacturer's proprietary message, not part of the NMEA protocol. The next three letters following the *P* indicate the manufacturer:

```
$PGRMZ,4,f,3*1F
```

This sentence is in proprietary format, from manufacturer Garmin. The *Z* indicates this message contains altitude information.

Comma-separated values follow the sentence type indicator. After the series of data values, an asterisk (*) represents the end of the sentence, followed by a two-digit hexadecimal checksum and a carriage return line feed.

The tutorial's sample application optionally parses out NMEA sentences. The next section moves closer toward the sample application with a discussion of the Java programming language and its applicability to GPS.

Section 3. Why Java for GPS?

A portable language for a portable application

GPS is inherently a portable application, used in a variety of industries and roles. The diversity of applications communicating with a GPS receiver cannot be confined to a PC or UNIX workstation. There is a mandate for portable and even embedded environments to interact with a GPS receiver.

The Java language's earliest claims were "write once, run everywhere." Its ability to run on multiple platforms with a binary compatibility offers a compelling case for selection when building GPS receiver applications. Applications written in Java language are compiled into *byte code*. The byte codes run in a Java Virtual Machine (JVM) on the target processor. The JVM interprets each set of byte codes and performs the appropriate action as a native instruction. Java architecture permits easy substitution of libraries to provide enhanced functionality as required.

At the lowest levels, Java function calls, known as *methods*, must interact with the host operating system to obtain and use operating system resources. For example, when a Java application opens a TCP socket to communicate with a server application, it must request the socket from the host operating system's TCP stack. This is accomplished through the use of native methods. The next panel discusses the role of the JNI and its significance to GPS.

Java Native Interface

Java language excels at core computing activities such as string manipulation, numeric calculations, sorting, hashing, file handling, network I/O, etc. However, it does not support native communications port interactions very well. Communicating with a serial or parallel port on a PC differs from the same activity on a UNIX workstation or a PDA. In fact, the physical differences can be vast. For example, a PDA does not have a parallel port. These distinctions present a challenge to the Java "write once, run everywhere" stance. Luckily, the JNI can resolve these hardware platform-specific issues.

The JNI requires the use of *shared libraries*. The Java runtime classes load these shared libraries dynamically via the `System.loadLibrary()` method. Once the Java runtime classes have loaded a JNI library, the methods exposed in that library are available to the Java environment. The Tutorial resources on page 23 section contains a link to an informative tutorial discussing the JNI.

Serial communications options for the Java programming language are very limited. A few commercial options are available for communicating with a serial device, and an offering from Sun Microsystems known as `javax.comm`. This tutorial demonstrates the `javax.comm` package.

The javax.comm package

The Java Communications API, for example, available from Sun Microsystems in the `javax.comm` package, is one that permits "platform-independent" access to communications resources. This package is a reference implementation available for the Solaris and Win32

platforms. The package offers support for both serial and parallel ports including support for typical activities such as configuring, sending, and receiving data. The API is event-driven, including events pertaining to the ownership of communication ports and other port conditions. This port-monitoring feature allows an application to wait for a port to become available for use; the API sends an event to the application, when the port is no longer in use by another application.

The implementation includes a platform-specific shared library containing the routines required for the JNI native methods of the package's classes. For example, on the Win32 platform, the reference implementation ships with a file named `win32com.dll`. Here are a few of the functions exported from the DLL:

```
_Java_com_sun_comm_Win32Driver_readRegistrySerial@12
_Java_com_sun_comm_Win32SerialPort_available@8
_Java_com_sun_comm_Win32SerialPort_nativeEnableFraming@12
_Java_com_sun_comm_Win32SerialPort_nativeSetDTR@12
_Java_com_sun_comm_Win32SerialPort_nativeSetFlowcontrolMode@12
_Java_com_sun_comm_Win32SerialPort_nnotifyOnBreakInterrupt@12
_Java_com_sun_comm_Win32SerialPort_nnotifyOnCTS@12
_Java_com_sun_comm_Win32SerialPort_nwrite@20
```

Javax.comm.properties file

The native shared library is not the only file in the `javax.comm` solution. The package also relies on a Java properties file to specify which class contains the actual driver. Once the file has identified the driver, the appropriate library is loaded, providing the required native methods for port interactions. Here is the `javax.comm.properties` file contents:

```
#
# Drivers loaded by the Java Communications API standard extension
# at initialization time
#
# Format:
#   Each line must contain ONE driver definition only
#   Each line must be of the form:
#     driver=<ClassName>
#      No spaces or tabs in the line.
#       ClassName must implement the interface
#        javax.comm.CommDriver
#         example: driver=Win32Serial
#
# The hash(#) character indicates comment till end of line.
#
# Windows Serial Driver
Driver=com.sun.comm.Win32Driver
```

The `com.sun.comm.Win32Driver` contains the `System.loadLibrary` ("win32com") to link the Java code with the native code.

The next section begins the hands-on portion of the tutorial: establishing the build environment and constructing the sample GPS application.

Section 4. Build environment

Obtaining and installing the Java SDK

Download the latest Java SDK and perform the installation, if a Java development environment is not already available.

The `javax.comm` package installation instructions indicate that the package has been tested with JDK 1.1.6; however, this tutorial utilizes the latest JDK available. See Tools on page 4 for more information on downloading the latest JDK. You cannot build some of the sample applications shipping with the `javax.comm` package with Java 2 versions, as those applications utilize deprecated methods. This is not a concern for this tutorial's sample application.

Note that the terms Java SDK and JDK are used synonymously.

Installing the javax.comm package

A link for downloading the Java Communications API package is available in Tools on page 4 . The site provides specific download instructions for each supported platform. Regardless of the platform you choose, here are the important steps in the installation:

1. Copy the platform-specific library (`win32com.dll` for Win32) to the `bin` folder of the JDK. For example, if the Java SDK is located in the `C:\jdk2` directory, copy `win32com.dll` into `C:\jdk2\bin`. This ensures that the native methods found in this library are available at runtime.
2. Copy the file `comm.jar` into the lib folder of the JDK. Continuing the example above, copy `comm.jar` into `C:\jdk2\lib`. You must include this file in the classpath during both compilation of the tutorial's application as well as during the runtime execution of the application.
3. You must also copy the `javax.comm.properties` file into the `lib` folder, `C:\jdk2\lib`. If the entry in this file is not available to the application, the communications driver class will fail to load and no device operations, such as enumerating or opening ports, will succeed.

Verifying the build environment

Now that you've installed the JDK and the Java Communications API, it's time to verify the installation. The following program exercises the port-enumeration capabilities of the API and serves to verify that the `javax.comm` package is properly installed and operational:

```
import javax.comm.*;
import java.util.Enumeration;
public class verifybuildenv
{
 public static void main(String args[])
  {
   Enumeration ports;
   System.out.println("Verifying Build Environment");
```

```
   try
    {
      ports = CommPortIdentifier.getPortIdentifiers();
      if (ports == null)
      {
        System.out.println("No comm ports found!");
        return;
        }
        while (ports.hasMoreElements())
        {
        System.out.println("Here is a port [" +
        ((CommPortIdentifier)ports.nextElement
        ()).getName() + "]");
        }
      }
      catch (Exception e)
      {
       System.out.println("Failed to enumerate ports
        [" + e.getMessage() + "]");
        e.printStackTrace();
      }
      System.out.println("Complete.");
   }
}
```

The next panel demonstrates how to build the `verifybuildenv` application using the JDK.

Build and run the application from the command line

This panel provides step-by-step instructions for building the first test application to verify the build environment.

Build this application with the following sequence from the command line:

```
javac -classpath \jdk2\lib\comm.jar verifybuildenv.java
```

Run the application with this command:

```
java -cp .;\jdk2\lib\comm.jar;  verifybuildenv
```

The expected output looks like this:

```
Verifying Build Environment
Here is a port [COM1]
Here is a port [COM3]
Here is a port [LPT1]
Here is a port [LPT2]
Complete.
```

If you do not achieve similar results, refer to the detailed instructions in the `Readme.html` file found in the `javax.comm` installation files.

Section 5. Sample application

Motivation and purpose of sample

The sample application, `ibmdwgps`, demonstrates the use of the `javax.comm` API in conjunction with a portable GPS unit. The purpose of the application is to create a baseline Java data collection program, capable of communicating with a GPS unit via serial communications. Subsequent tutorials expand on GPS-specific features and protocols. Once complete, the application has the following features:

- Java AWT user interface
- Serial port communications via the Java Communications API
- Java file system I/O including file dialog interfaces for saving traces captured from the GPS unit
- GPS/NMEA sentence capture
- Java property file use allowing a highly configurable application

Tools on page 4 contains a link to the complete source code for this application.

The sample application was tested against a Garmin eTrex Venture model handheld GPS unit communicating NMEA sentences at 4800 baud in "demo mode."

Configuration and properties file

A common feature of communications applications is parameterization or configuration to switch between different ports, speeds, settings, etc. This application utilizes a Java properties file, `ibmdwgps.properties`, to provide these values. This file contains the port number, the baud rate, and a setting indicating whether to expect NMEA sentences.

```
## ibmdwgps.properties
version=Version 1.0\r\n
welcomemessage=Welcome to IBM developerWorks :
Wireless: Introduction to Java and GPS\r\n
commportnumber=COM1
baudrate=4800
protocol=NMEA
```

In addition to the property file, a Java application can receive properties on the command line of the form:

```
java -cp <someclasspath> -Dname=value
-Dname=value <classtorun>
```

Having multiple means to provide runtime parameters enhances the flexibility and portability of the Java environment.

Property access

This panel examines the sample application's approach to accessing the properties at

runtime, regardless of the manner in which they are provided to the program.

Any properties passed on the command line are accessible from the system object's properties collection. This application loads the properties found in the `ibmdwgps.properties` file into the system object's properties for simplified property access at any point in the program. This way, there is no guessing as to where you might find a given property.

```
// load properties
try
{
 config = new Properties(System.getProperties());
 config.load(new FileInputStream
 ("ibmdwgps.properties"));
 System.setProperties(config);
}
catch (Exception e)
{
 System.err.println("Failed to retrieve
 properties [" + e.getMessage() + "]");
 e.printStackTrace();
 System.exit(1);
}
trace.append(System.getProperty("welcomemessage"));
trace.append(System.getProperty("version"));
commportnumber = System.getProperty("commportnumber");
baudrate = System.getProperty("baudrate");
if (commportnumber == null)
{
 trace.append("Failed to load commportnumber
 from properties");
 enabled = false;
}
```

The next section describes the user interface and the major options available to the user.

Section 6. User interface and options for the sample application

User interface

The sample application presents a very simple user interface based on the `Frame` class of the AWT. The application also implements the *ActionListener* interface, allowing it to respond to user actions such as menu selections. For more information on Java user interface programming, see Tutorial resources on page 23 . Here is the class declaration specifying the specific class properties:

```
public class ibmdwgps extends Frame implements
ActionListener,SerialPortEventListener
```

The client area of the frame contains a scrollable *TextArea* used for displaying status information and any data retrieved from the GPS unit. The following code is a representative subset of the user interface code:

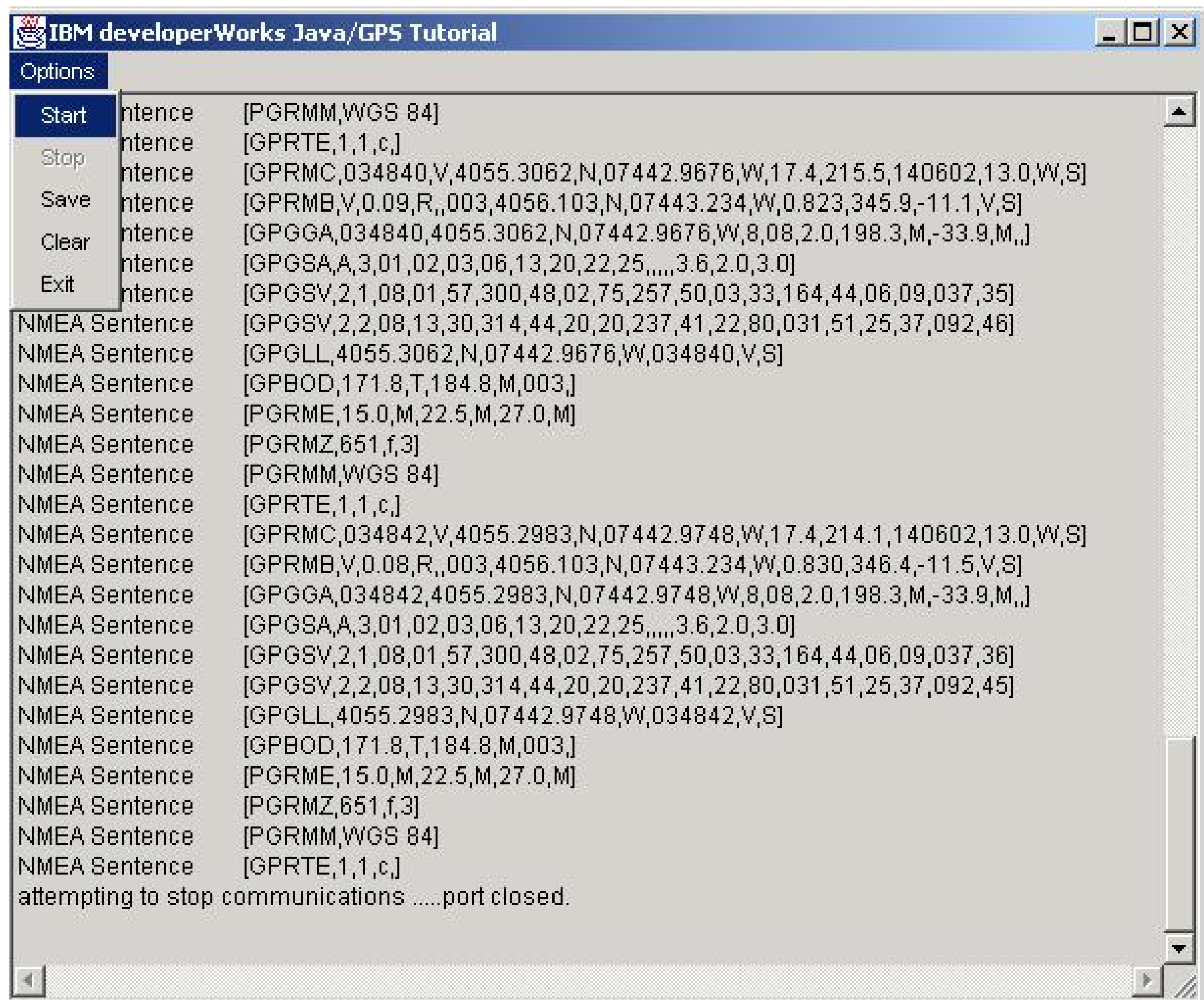
```
private Panel    tracewindow;
private TextArea trace;
final int HEIGHT = 480;
final int WIDTH  = 600;
private Panel    tracewindow;
private TextArea trace;
private MenuBar barMain;
private Menu mnuMain;
private MenuItem miStart;


super("IBM developerWorks Java/GPS Tutorial");
tracewindow = new Panel();
tracewindow.setLayout(new GridLayout(1, 1));
trace = new TextArea();
trace.setEditable(false);
tracewindow.add(trace);
add(tracewindow, "Center");
barMain = new MenuBar();
mnuMain = new Menu("Options");
miStart= new MenuItem("Start");
miStart.addActionListener(this);
mnuMain.add(miStart);
...
barMain.add(mnuMain);
setMenuBar(barMain);
Dimension screenSize = Toolkit.getDefaultToolkit()
 .getScreenSize();
setLocation((screenSize.width - WIDTH)/2,
(screenSize.height - HEIGHT)/2);
setSize(WIDTH, HEIGHT);
```

All of this declarative Java code results in a graphical user interface, discussed in the next panel.

Main application window

The primary user interface window provides a scrolling view of all data received along with user-selectable menu choices for controlling the application.



The following menu choices correspond to the available actions the sample takes:

- Start -- This option commences data collection via the serial port.
- Stop -- This option closes the serial port.
- Save -- This option prompts the user for a filename and location to save the contents of the captured data in the TextArea.
- Clear -- This option clears all text from the TextArea.
- Exit -- This menu terminates the application.

The next few panels drill down into each of these options with sample source code.

Action handler

The `ibmdwgps` class implements the ActionListener interface, allowing it to respond to user actions. To implement this interface, a class must provide a method named `actionPerformed()`, taking a single `Event` parameter. Here is the action handler code:

```
public void actionPerformed(ActionEvent e)
{
   String cmd = e.getActionCommand();
   System.err.println(cmd);
   if (cmd.equals("Exit"))
   {
      shutdown();
   }
   if (cmd.equals("Start"))
   {
      startupcomms();
   }
   if (cmd.equals("Stop"))
   {
      stopcomms();
   }
   if (cmd.equals("Save"))
   {
      savetrace();
   }
   if (cmd.equals("Clear"))
   {
      trace.setText("");
   }
}
```

This method essentially acts as a distributor of events. For example, when the method performs the Start event, it transfers control to the `startupcomms()` method.

Start option

The `startupcomms` method performs the majority of interaction with the serial port via the `javax.comm` API. The actions this method takes include obtaining a port identifier and attempting to open the port:

```
try
{
 portId = CommPortIdentifier.getPortIdentifier
 (commportnumber);
}
catch (NoSuchPortException e)
{
   trace.append("Failed to get port identifier!
   [" + e.getMessage() + "]");
   e.printStackTrace();
   return;
}

try
{
   sPort = (SerialPort)portId.open("ibmdwgps", 1000);
}
catch (PortInUseException e)
{
 trace.append("Port in use??
  [" + e.getMessage() + "]");
   e.printStackTrace();
   return;
```

```
}
```

Note the use of the try/catch mechanism. Java code requires that every exception be caught. In this case, the application is concerned with opening an invalid or busy port.

If there are no errors in opening the port, the application next attempts to prepare the port for operation by assigning port-specific parameters.

Port configuration

Once the port is open, it must be configured for the specific communications parameters required to talk to the GPS device:

```
try
{
 trace.append("Setting parameters\r\n");
 sPort.setSerialPortParams(4800,SerialPort.DATABITS_8,
 SerialPort.STOPBITS_1,SerialPort.PARITY_NONE);
}
catch (UnsupportedCommOperationException e)
{
 trace.append("Failed to set port parameters
 [" + e.getMessage() + "]");
 e.printStackTrace();
 sPort.close();
 return;
}

// Set flow control.
try
{
 sPort.setFlowControlMode(SerialPort.FLOWCONTROL_NONE);
}
catch (UnsupportedCommOperationException e)
{
   trace.append("Setting flow control failed
   [" + e.getMessage() + "]");
   e.printStackTrace();
   sPort.close();
   return;
}
```

The port is now open and set up for the required serial data format. The next step is to prepare the Java-specific I/O mechanisms required for proper data handling.

Java I/O streams

The Java programming language has a remarkably powerful, but often underutilized, I/O architecture. The `SerialPort` class provides both an input and an output stream for application use.

The balance of the `startupcomms` method pertains to I/O and notification settings of the port:

```
try
{
   os = sPort.getOutputStream();
   is = sPort.getInputStream();
}
catch (IOException e)
{
   sPort.close();
   trace.append("Error opening i/o streams
   [" + e.getMessage() + "]");
   e.printStackTrace();
   return;
}
try
{
   sPort.addEventListener(this);
}
catch (TooManyListenersException e)
{
   sPort.close();
   trace.append("Cannot add this
   class as a listener!
   [" + e.getMessage() + "]");
   e.printStackTrace();
   return;
}
try
{
   sPort.enableReceiveTimeout(1);
}
catch (Exception e)
{
   trace.append("Cannot set enableReceiveTimeout
    [" + e.getMessage() + "]\r\n");
   e.printStackTrace();
}
// Set notifyOnDataAvailable to true to allow event
    driven input.
sPort.notifyOnDataAvailable(true);
```

The last action the method takes is setting a class-level variable and toggling the menu options.

```
portisopen = true;
miStop.setEnabled(portisopen);
miStart.setEnabled(!portisopen);
```

Serial event

In addition to the ActionListener interface, the `ibmdwgps` class also implements the SerialPortEventListener interface by providing the `serialEvent()` method. This method is invoked whenever an event on the port occurs. The most common example is the arrival of data at the port. The first portion of `serialEvent` simply dumps all received data to the TextArea.

```
public void serialEvent(SerialPortEvent e)
{
```

```
//System.err.println("Serial Event
[" + e.getEventType() + "]");
   int newData = 0;

switch (e.getEventType())
  {

 // Read data until -1 is returned.
   If \r is received substitute
     // \n for correct newline handling.
       case SerialPortEvent.DATA_AVAILABLE:
    if (!nmea)
     {
        inputBuffer = new StringBuffer();
        while (newData != -1)
        {
           try
           {
              newData = is.read();
               if (newData == -1)
              {
                 break;
              }
              inputBuffer.append((char)newData);
           }
           catch (IOException ex)
           {
              System.err.println(ex.getMessage());
              return;
           }
           }
           // Append received data to messageAreaIn.
           trace.append(new String(inputBuffer));
        }
```

This code handles basic data receipt; however, it does not attempt to interpret the data at all. The next panel examines NMEA sentence parsing.

NMEA sentence parsing

If the NMEA option is selected via the properties file (protocol=NMEA), the data is buffered and parsed into NMEA sentences as it receives it:

```
        {
           // parse NMEA sentence
           boolean done = false;
           boolean gotsentence = false;
           boolean sentenceactive = false;
           boolean eatchecksum = false;
           while (!done)
           {
              try
              {
               newData = is.read();
               switch (newData)
               {
                  case -1:
                     done = true;
                     break;
```

```
                    case '$':
                       sentenceactive = true;
                       break;
                    case '*':
                       eatchecksum = true;
                       break;
                    case 0x0d:
                       break;
                    case 0x0a:
                       sentenceactive = false;
                       gotsentence = true;
                       done = true;
                       break;
                    default:
                       if (!eatchecksum)
                       {
                          inputBuffer.append((char)newData);
                       }
                       break;
                 }
                {     catch (IOException ex)
                {
                   System.err.println(ex.getMessage());
                }
             }
             if (gotsentence)
             {
                trace.append("NMEA Sentence\t
                [" + new String(inputBuffer) + "]\r\n");
                inputBuffer = new StringBuffer();
             }
          }
          break;

       // If break event append BREAK RECEIVED message.
       case SerialPortEvent.BI:
          trace.append("\n--- BREAK RECEIVED ---\n");
          break;
   }
}
```

Regardless of the option chosen, `serialEvent` handles the incoming data and properly displays it to the user.

Stop option

The `stopcomms()` method closes the communications port and resets the user interface menu options. It does not remove any captured text from the TextArea. Here is the code:

```
public void stopcomms()
{
   trace.append("attempting to stop communications...");
   portisopen = false;
   sPort.close();
   miStop.setEnabled(portisopen);
   miStart.setEnabled(enabled);
   trace.append("port closed.\r\n");
}
```

Save option

This option gives the application the ability to save the collected GPS data. It prompts the user for a filename and then writes the contents of the TextArea to the selected file.

```
public void savetrace()
{
   Calendar cal;
   NumberFormat nf;

   nf = NumberFormat.getInstance();
   nf.setMinimumIntegerDigits(2);
   cal = Calendar.getInstance();
   System.err.println("savetrace()");
   System.err.println(trace.getText());
   FileDialog fd = new FileDialog
   (this, "Save Trace",FileDialog.SAVE);
   fd.setFile("gpstrace_" + Integer.toString
   (cal.get(Calendar.YEAR)) +
   nf.format(cal.get(Calendar.MONTH)+1) +
   nf.format(cal.get(Calendar.DAY_OF_MONTH)) + ".txt");
   fd.setVisible(true);
   String fileName = fd.getFile();
   String directory = fd.getDirectory();
   System.err.println(directory + fileName);
   try
   {
      FileOutputStream fos = new FileOutputStream
      (directory + fileName);
      fos.write(trace.getText().getBytes());
      fos.close();
   }
   catch (Exception e)
   {
      System.err.println("failed to save trace
      [" + e.getMessage() + "]");
      e.printStackTrace();
   }
}
```

This code provides a default filename utilizing the `Calendar` class. Once the code selects a filename, the code converts the text found in the TextArea to a byte array and then writes it to disk.

Note: The action of converting between a Java String and a byte array can lead to unexpected and undesirable results if the original input data contains nonprintable characters. These characters may not be available in the current Locale. However, storing them as a byte array to begin with would be preferable to the chosen approach of a String and StringBuffer. This application uses text because all of the data in NMEA format is simple text.

Section 7. Summary

Tutorial summary

This tutorial demonstrated the Java Communications API in the context of collecting basic Global Positioning Data. The JNI, a crucial portion of the Java programming language's architecture, provides a foundation for all physical device communication. The sample application demonstrates a reference implementation of a JNI solution in the form of the Java Communications API, or `javax.comm`. The sample application uses basic stream-oriented communications to communicate with a physical device, including the parsing of basic NMEA sentences. Using this application represents a simple but useful exercise in the examination of GPS protocols and the Java programming language. It is the starting point for subsequent functionality and exploration of GPS protocols and Java programming language techniques in subsequent tutorials. The next tutorial, entitled *GPS data concepts*, builds upon the lessons learned in this introductory tutorial by drilling down further into a conversation between the GPS unit and a sample Java application. A third tutorial examines the task of porting the enhanced application to a PDA.

Tutorial resources

- The U.S. Coast Guard's Navigation Center at *http://www.navcen.uscg.gov/gps/default.htm* is a good starting place for learning more about GPS .
- This research project (*http://www.ctre.iastate.edu/mtc/projects/gpsintegration.htm*) discusses the use of GPS for the purpose of tracking mobile assets.
- The FRP is the authoritative document on GPS (*http://www.navcen.uscg.gov/pubs/frp2001*).
- OnStar (*http://www.onstar.com*) is an in-vehicle service delivering a variety of services, all based on the principle that your vehicle is easily located via GPS technology.
- This tutorial by Scott Stricker discusses the basics of the JNI (*http://www-105.ibm.com/developerworks/education.nsf/java-onlinecourse-bytitle/52FFDF7067EAA64986*
- Download the latest Java SDK from *http://java.sun.com/j2se*.
- Download the `javax.comm` package from *http://java.sun.com/products/javacomm/index.html*. This package is required for this tutorial's sample application.
- The Palm Web site at *http://www.palm-communications.com/ibmdw* contains all of the source code referenced in this tutorial.
- "Java 3D Joyride" at *http://www-105.ibm.com/developerworks/education.nsf/java-onlinecourse-bytitle/8BFBD0EBD772D2EF86* discusses Java graphing, an interesting avenue to explore for representing GPS data.
- *http://www.delorme.com/earthmate/default.asp* -- DeLorme has specially designed its GPS offering for interfacing with laptop and Palm OS software, providing comprehensive mapping for in-vehicle navigation.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.