Javelin Stamp Manual

Version 1.1

Warranty

Parallax Inc. warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax Inc. will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax. Email: Sales@parallax.com; Support@parallax.com; Tel: (916)624-8333; Fax (916) 624-8003.

14-Day Money-Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax Inc. will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

Copyrights and Trademarks

ISBN: 1928982-15-8

Disclaimer of Liability

Parallax Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax Inc. is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your BASIC Stamp application, no matter how life-threatening it may be.

Internet Access and Discussion Forums

We maintain active web-based discussion forums for people interested in Parallax products. These lists are accessible from www.parallax.com via the Support → Discussion Forums menu. These are the forums that we operate from our web site:

- BASIC Stamp – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- Stamps In Class® – Created for educators and students, subscribers discuss the use of the Stamps in Class educational products in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- Parallax Educators –Exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our educational products and to provide a forum for educators to develop and obtain Teacher's Materials and other resources.
- Robotics – Designed for Parallax robots, this forum is intended to be an open dialogue for robotics enthusiasts. Topics include assembly, source code, expansion, and manual updates. The Boe-Bot®, Toddler®, SumoBot®, HexCrawler and QuadCrawler robots are discussed here.
- SX Microcontrollers and SX-Key – Discussion of programming the SX microcontroller with Parallax assembly language SX – Key® tools and 3rd party BASIC and C compilers.
- Javelin Stamp – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.
- Propeller Chip – Forum for those using the Parallax Propeller™ chip.

Table of Contents

Table of Contents

Table of Contents

Table of Program Listings

Table of Program Listings

Table of Figures

Table of Figures

Table of Tables

Table of Tables

Manual Organization

This manual was written under the assumption that the reader's level of experience could be anywhere between beginner and advanced embedded Java™ aficionado. We recommend that you start from the beginning and work your way through this manual sequentially, especially if you are new to both circuits and Java. Make sure to try all the examples and understand how they work before moving on to the next. For those of you who do not fall at either end of the spectrum, below is a condensed table of contents with comments regarding the intended audience and uses of each chapter.

Preface

General information - discusses Javelin Stamp's features, this manual's format and conventions, resources and acknowledgements.

1: Introduction

General information - about the Javelin Stamp, its uses, equipment it can be used with, specifications, software, etc.

2: Javelin Stamp Quick Start

Recommended for all – includes step by step instructions for software installation, hardware setup, trouble shooting, a couple of example programs, an example circuit, and a software tour.

3: Beginners Guide to Embedded Java™ Programming

Recommended for Java newcomers and BASIC Stamp users - if you've never programmed in Java before, read this, and try the examples!

4: Application Examples – Circuits and Programs

Recommended for embedded newcomers and BASIC Stamp users – provides good examples for BASIC Stamp users to make the transition to Java based hardware design, and helps those new to circuit based programming projects get their feet wet.

5: Using the Javelin Stamp IDE

Recommended for all – the Javelin Stamp IDE is a powerful tool with many useful features.

6: Javelin Stamp Programmers Reference

If you are a Java programmer, pay close attention to the differences between Java for the Javelin Stamp and Java on your PC. For beginners, this is a good way to learn programming in Java.

7: Working with Objects

Recommended if you are still learning Java – by this point, if you were new to Java at the beginning of this manual, you are now well into the learning curve.

8: Object Reference

Recommended for all – whether you are an experienced Java programmer or you just finished Chapter 7, this chapter explains the Java library classes available for use with the Javelin Stamp.

9: Javelin Stamp Hardware Reference

Recommended for all – explains all the hardware related library classes and methods. If it has to do with a VP, a peripheral or an external circuit, the information is here.

10: Technical Details

Appendix material.

Java Programmers – READ THIS

The Javelin Stamp is a small yet powerful controller that makes use of a subset of Java 1.2. The Javelin Stamp has firmware enhancements (called Virtual Peripherals or VPs) that emulate, or virtualize, hardware devices such as UARTs, timers, A/D converters, D/A converters, and more. These VP's have been painstakingly optimized, and they take the form of native methods that make it easy to interface with just about any circuit or peripheral device. Many of these firmware features are similar to those that lead the BASIC Stamp's popularity, and others have long been on BASIC Stamp users' wish lists.

The flip side of the Virtual Peripheral firmware features is that they have been incorporated into the Javelin Stamp at the expense of Java purity. You will find the experience of developing applications with the Javelin Stamp uniquiely different from developing applications on a PC. To get to the rewards of a rapid prototype of your product design or project with minimal stumbling, we recommend above all that you try the many programming and circuit examples in this text. Before getting started on the examples, take a few minutes to review the reading list below. It will acquaint you with the scope of Javelin Stamp projects and help you avoid some of the programming pitfalls you might otherwise encounter.

Suggested reading for Java Programmers:

BASIC Stamp Enthusiasts – READ THIS

As with the Java Programmers who were addressed in the previous section, programming the Javalin Stamp is also likely to be very different from what you, the BASIC Stamp Enthusiast, are expecting. This manual has LOTS of example programs and circuits to help you transition from PBASIC to the Java subset used to program the Javelin Stamp. Especially if you are unfamiliar with Java, we strongly recommend that you work through the examples in this text sequentially. The majority of this manual's organization was established with you in mind, so, if you have not already done so, please take a look at the Manual Organization section at the beginning of this preface. If you are like the rest of us at Parallax, you probably can't wait to get started, so have fun with Chapter 2: Javelin Stamp Quick Start.

Manual Conventions

Below is a list of typographical conventions used in this manual:

`Monospaced` is used for:

- Words that are part of the language syntax when they are part of a sentence.
- Fragments of programs. The code snippet below is an excerpt from a program, but it cannot be run on its own. It has to appear in either a complete program or a complete class file, both of which are discussed next:

```
System.out.println("Not a complete program.");
```

A gray box is used for:

- Complete programs that can be entered into the Javelin Stamp IDE and executed on a Javelin Stamp, for example:

```
import examples.manual_v1_0.*;
public class CompleteProgram{
  public static void main() {
  CompleteClassFile example = new CompleteClassFile();
    System.out.println("Now, it's in a complete program.");
    example.displaySameMessageAgain();
  }
}
```

- Complete class files that can be instantiated by other programs. Here is an example:

```
package examples.manual_v1_0;
public class CompleteClassFile {
  public static void displaySameMessageAgain() {
    System.out.println("Now, it's in a complete class file");
  }
}
```

Preface

Resources and Technical Support

Free Javelin Stamp Technical Support can be obtained from Parallax.com via:

- Telephone: (916) 624-8333
- Toll Free in U.S.: (888) 99-STAMP which is (888) 997-8267
- Fax: (916) 624-8003
- Email: support@parallax.com

For online support: Follow the Javelin Stamp link at www.parallax.com for the latest in tech support contact info, discussion group links, manual errata, answers to frequently asked questions, and more! For a complete list of Parallax Discussion Groups, see the reverse of the title page (page 2).

Free Downloads from www.parallax.com

You can always get the latest revisions and updates of the following items from the Javelin Stamp link at www.parallax.com:

- Javelin Stamp Manual
- Javelin Stamp IDE
- Application Notes
- Library Files

Acknowledgements

Chris Waters and Celsius Research provided the Javelin Stamp firmware and reference design. This manual was developed using information and research provided by Al Williams Consulting. Each and every employee at Parallax has made some contribution to the Javelin Stamp project, so as always, thanks to the entire Parallax staff.

The Javelin Stamp and Its Features

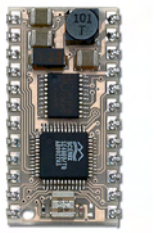


Figure 1.1 Javelin Stamp (top view)

The Javelin Stamp is a single board computer that's designed to function as an easy-to-use programmable brain for electronic products and projects. As shown in Figure 1.1, it's about the size and shape of a commemorative postage stamp. It is programmed using software on a PC and a subset of the Sun Microsystems Java® programming language. After the program is downloaded to the Javelin Stamp, it can run the program without any further help from the PC. The Javelin Stamp can be programmed and re-programmed up to one million times.

We hope you enjoy working with your new Javelin Stamp as much as we have while preparing this manual. The Javelin Stamp is somewhat of a departure from Parallax's BASIC Stamps. Most notably, the Javelin Stamp is programmed using a subset of the Java programming language. Some of the other features that set the Javelin Stamp apart from BASIC Stamps are:

- The instruction codes for the Javelin Stamp are fetched and executed from a parallel SRAM instead of a serial EEPROM.
- The Javelin Stamp has 32k of RAM/program memory with a flat architecture. No more program banks, and no more tight squeezes with variable space.
- The Javelin Stamp has built in Virtual Peripherals (VPs) that take care of serial communication, pulse width modulation and tracking time in the background.
- Serial communication is buffered as a background process. When writing programs, all you have to do is periodically check the buffer.
- The Javelin Stamp Integrated Development Environment (Javelin Stamp IDE) software is a significant departure from a simple Editor and messages window combination. When used with the Javelin Stamp connected to a PC by a serial cable, this software can be used as a highly integrated in-circuit debugging system that allows you to run code, set breakpoints and view variable values, memory usage, I/O pin states and more. There is also no need for emulators; the Javelin Stamp can be placed directly into the circuit and debugged there.
- Sigma-Delta A/D conversion.
- D/A conversion is accomplished in the background as a continuous pulse train delivered by an I/O pin. The pulse width modulation VP can also be used for generating pulse trains, frequencies, and D/A conversions in the background while your foreground code is free to perform other tasks.

1: Introduction

Those of you who appreciate the simplicity and ease of use of the BASIC Stamps need not worry; the Javelin Stamp has many features that BASIC Stamp users have come to depend on in their projects and designs. Here is a list of features built into the Javelin Stamp with BASIC Stamp users in mind:

- Synchronous serial communication (shiftIn/shiftOut)
- The ability to both send and measure discrete pulses (pulseIn/pulseOut)
- Frequency counting (count)
- Simple and intuitive methods for reading from and writing to I/O pins
- Measurement of RC charge and discharge times (rcTime)

BASIC Stamps have been used for everything from lessons in basic computer programming and electronics, all the way up to aerospace subsystem designs. We expect to see the Javelin Stamp used in a similar manner. However, by making use of the Javelin Stamp's new features, it can be used to tackle some more demanding designs that used to require larger processors.

Programming Language - Java™ for the Javelin Stamp

The Javelin Stamp's programming language supports many of the Java languages most useful features:

- Object Orientation - Inheritance, method overloading, polymorphism and static initializers.
- Exceptions - Try-catch-finally blocks and the ability to catch exceptions with a super-class.
- Strings – Programmed using many familiar Java commands.
- Custom Library Support - For many popular peripherals such as LCDs, temperature, AD, communication ICs, and common Internet protocols such as ARP, UDP, and PPP.

Java Differences	There are some differences between writing applications for your PC using Java 1.2 and the subset of Java used by the Javelin Stamp. Experienced Java programmers should consult the Summary of Java Differences section in Chapter 10.

Javelin Stamp Integrated Development Environment

Javelin Stamp Integrated Development Environment (Javelin Stamp IDE) offers the features that you would commonly expect from a source-level debugger:

- Multiple breakpoints
- Stack backtrace
- Inspection of all variables and objects, both static and dynamically allocated
- Single-step, run, stop, reset
- Built-in bi-directional serial message terminal for **`System.out.println()`** and **`Terminal.getChar()`** type debugging

The Javelin Stamp IDE is introduced in Chapter 2, and then discussed in more detail in Chapter 5. This IDE makes real-time debugging so easy that a PC emulator is completely unnecessary. It is just as easy to develop and debug on the Javelin Stamp module itself.

Virtual Peripherals

The Javelin Stamp firmware supports a variety of Virtual Peripherals (VPs). The VPs are separated into two categories, foreground and background. The background processes allow you to create UARTs, pulse trains, and a timer. Once created, background VP objects run independently from the program. Since time-sensitive tasks are taken care of by the VPs in the background, designs that used to be difficult become easy. For example, serial communication does not stop just because the Javelin Stamp is measuring the duration of an incoming pulse. The programmer simply needs to periodically check the serial buffer in the foreground code. Below is a list of background and foreground VPs.

Background VPs

- UART (Full duplex, optional HW flow control, buffered)
- PWM
- 32-bit Timer
- 8-bit PWM DAC
- 8-bit Sigma-Delta ADC

Foreground VPs

- Pulse count
- Pulse width measurement
- Pulse generation
- RC Timer
- SPI master

> *These Virtual Peripherals are built into the Javelin Stamp's firmware. Although you can write library classes that make use of these VPs, the VPs themselves cannot be modified or rewritten.*

How the Javelin Stamp Works

The Javelin Stamp's hardware architecture is shown in Figure 1.2. Programming and debugging is done via communication with the serial port. The COM circuit takes care of the voltage conversions necessary for a TTL device to talk with an RS232 port. The Java interpreter processes all serial port/COM circuit information. Whether it's byte codes, debugging data or serial messages, the interpreter processes the data and decides what to do with it.

When a program is downloaded, the interpreter buffers the program bytecodes and writes them to the EEPROM. Upon reset (or a power interruption), all the Javelin Stamp's I/O pins are set to input. The interpreter copies the bytecodes to the SRAM, then starts fetching bytecodes from the SRAM and executing them. The bytecode instructions can be executed very rapidly because all data is transmitted along parallel data busses instead of synchronous serial lines. A typical fetch and execute cycle involves a couple of read/write cycles. During a read/write cycle, the interpreter loads some of the 15 bit address information into an address latch and writes the other portion directly to the SRAM. When the SRAM address is set, then the data is read or written by the interpreter as needed.

1: Introduction

The Javelin Stamp's internal voltage regulation is done using a switching regulator. The switching regulator runs cooler and is significantly more efficient than a linear regulator. It accepts voltages between 6 and 24 V, and makes 5 V available for the Javelin Stamp with a total current budget of 150 mA. The passive components including the input and output capacitors, switching diode and inductor are on the top side, and the switching IC is on the bottom side of the board next to the EEPROM. The switching IC monitors the output voltage and adjusts the switching duty cycle to the passive components to maintain a constant 5 V output.

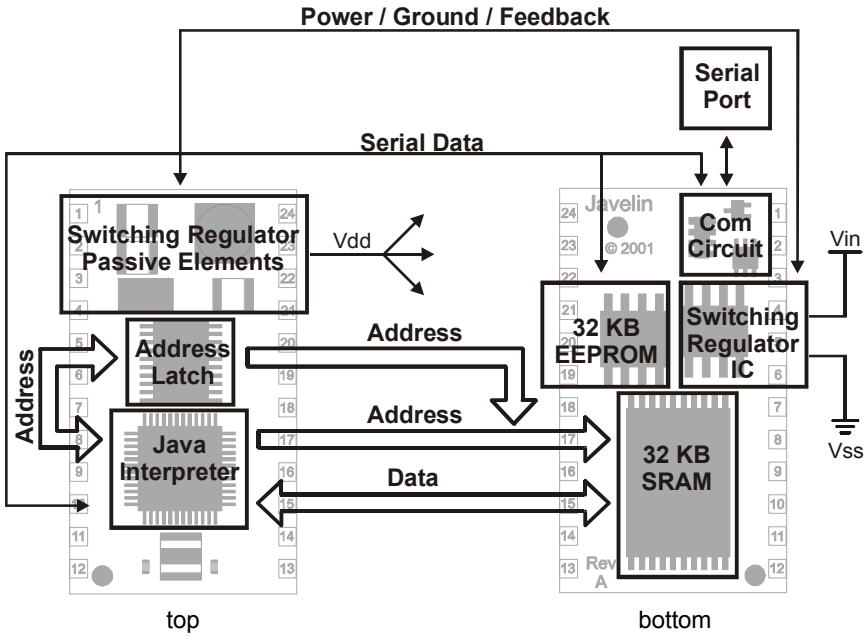


Figure 1.2
Javelin Stamp
Block Diagram

Javelin Stamp Hardware

Table 1.1 shows the Javelin Stamp's specifications. Note that the onboard voltage regulator can accept between 6 and 24 V_{DC} and output up to 150 mA of current. Since the Javelin Stamp consumes approximately 60 mA, you have 90 mA available for other uses. Keep in mind that if you are utilizing the full 60 mA of total I/O pin source/sink that only 30 mA is left over for powering peripheral devices using the Javelin Stamp's Vdd pin. On the other hand, if all the I/O pins are being used for input, 90 mA can be used drawn from the Javelin Stamp's voltage regulator output (Vdd) for peripherals. If in doubt, use an external 5 V regulator for your peripherals.

Table 1.1: Javelin Stamp Hardware Specifications

Attribute	Value
Module Footprint	24-pin DIP module
Package Measurements (LxWxH)	1.2"x0.6"x0.4" (3.0x1.5x1.0 cm)
Operating Environment	0º - 70º C (32º - 158º F)
Microcontroller	Ubicom SX48AC
RAM	32 kilobytes
EEPROM	32 kilobytes
Number of I/O pins	16
Voltage Supply	6 – 24 VDC (unregulated) - or - 5 VDC (regulated)
Voltage regulator current output	$0 < I_{out} < 180$ mA
Current Consumption	60 mA / 13 mA nap
Sink/Source Current per I/O	30 mA / 30 mA
Sink/Source Current per module	60 mA / 60 mA per 8 I/O pins
Sink/Source Current per Bank Pins (0 – 7) and (8 - 15)	30 mA / 30 mA
Windows Editor/Debugger	Javelin Stamp IDE

1: Introduction

Equipment and System Requirements

To run the IDE and program the Javelin Stamp, you will need an IBM PC or compatible computer with the following:

- Windows 95, 98, ME, 2000, or XP.
- Internet connection.
- An available 9-pin serial port
  Or – A USB port with an approved USB to serial adaptor. See www.parallax.com for information on products that have been tested and approved.
  Or – A 25-pin serial port with a 25 to 9-pin adaptor.

The Javelin Stamp Starter Kit is discussed in detail in the following section: *Useful Hardware*. If you do not have a Javelin Stamp Starter kit, you will need to acquire at least the following.

- Recommended DC Power Supply: 7.5 VDC, 1000 mA 2.1 mm, center-positive
  Acceptable battery/DC Power Supply values range between 6 and 24 VDC. Minimum output current rating depends on voltage. A 6 V supply can have an output current rating as low as 100 mA while higher voltage supplies may need higher output current ratings.
- Serial programming cable
  Be sure to use a straight-through serial cable or adaptor. Do not try to use a null modem cable or adaptor for downloading programs to the Javelin Stamp.
- Carrier board or serial cable and power supply connections
  Parallax makes a variety of carrier boards for BASIC Stamps. The Javelin Stamp can be powered and programmed using any of these carrier boards. You can also make your own connections for supply voltage and serial cables. See the Hardware Setup section in Chapter 2.

Useful Hardware

The Javelin Stamp Starter kit is a great way to get started, especially if this is your first adventure into Javelin Stamp based projects. Projects featured in Chapters 2, 4, and 9 make use of the carrier board and parts in this starter kit. The Javelin Stamp Demo Board is the carrier board included in the kit, and its features are shown in Figure 1.3 and listed below.

The Javelin Stamp Demo board (Figure 1.3) has the following features:

- Socket for the Javelin Stamp (Labeled U1).
- JIDE port for debugging, messages, and downloading programs from the PC into the Javelin Stamp.
- A power jack that can accept input voltage ranging from 6 to 24 V_{DC}.
- A COM port that can be used to connect the Javelin Stamp to other computers. Alternately, you can attach a null modem adaptor to this COM port and then connect the Javelin Stamp to peripherals such as serial GPS units, mice, etc.
- Linear voltage regulator for prototype circuits.

- Small breadboard area for building, testing and prototyping circuits.
- A power header (supplied by the linear voltage regulator). This header can be used to supply circuits with power.
- A Javelin Stamp I/O header to connect your Javelin Stamp I/O pins to your circuit.
- COM Port I/O header. You can use jumper wires to connect Javelin Stamp I/O pins to the COM port I/O header. Then you can write code to communicate with another serial device such as a computer or peripheral that's connected to the COM port.
- LED power indicator (labeled PWR).
- Reset pushbutton. Press and release to restart the program from its beginning.
- A servo port for connecting and controlling servo motors.

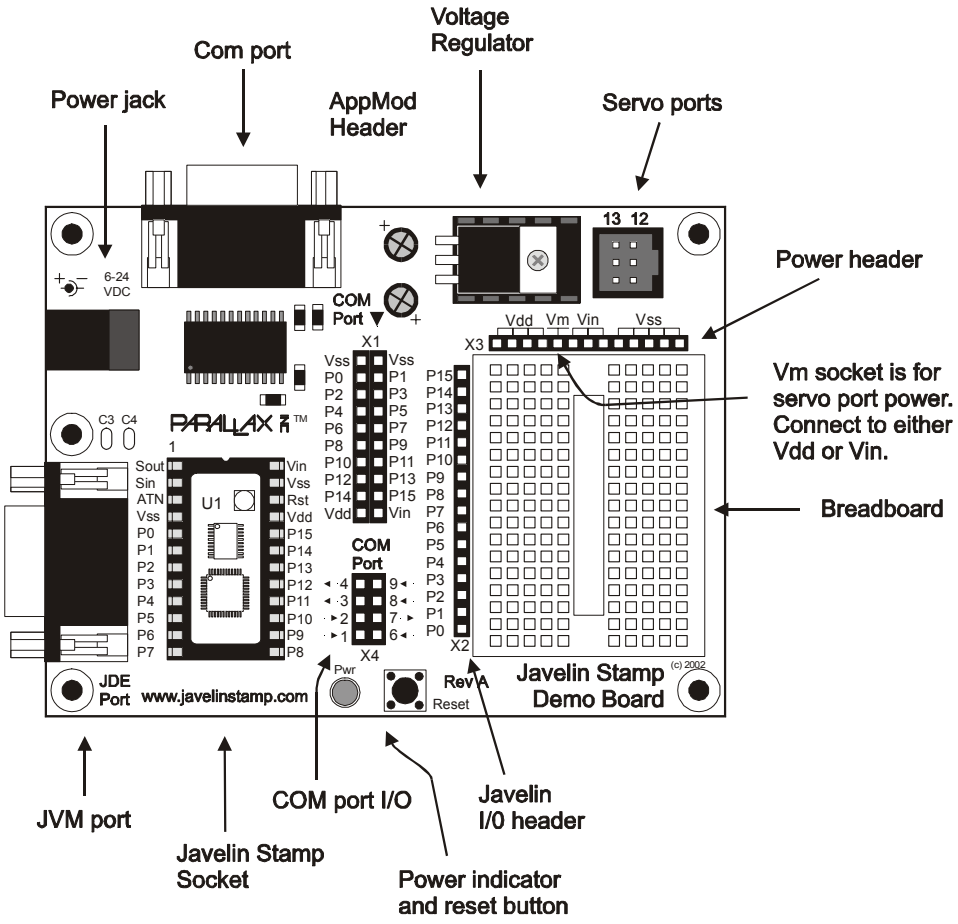


Figure 1.3 Javelin Stamp Demo Board Features

As mentioned earlier, the circuit examples in this manual feature parts you can find in the Javelin Stamp Starter Kit. The parts are listed in Table 1.2. Table 1.3 lists parts that are also recommended but not included in the kit.

Table 1.2: Javelin Stamp Starter Kit (#27237)

Quantity	Part Number	Part Description
1	550-00019	Javelin Stamp Demo Board
1	JS1-IC	Javelin Stamp Module
1	27957	Javelin Stamp Manual
1	800-00003	Serial Cable
1	800-00002	DB9 Null Modem Adapter Male to Male
1	604-00002	DS1620 Digital Thermometer
1	350-00009	Photoresistor
1	900-00001	Piezo Speaker
1	602-00009	74HC595 Output Shift Register
1	602-00010	74HC165 Input Shift Regster
3	400-00002	Tact Switch (Pushbutton)
2	350-00006	LED - Red - T1 3/4
8	350-00001	LED - Green - T 3/4
1	150-02210	RES - 220 - ¼ W - 5%
8	150-04710	RES - 470 - ¼ W - 5%
1	150-01020	RES - 1 k - ¼ W - 5%
3	150-01030	RES - 10 k - ¼ W - 5%
2	156-02230	RES - 22 k - ¼ W - 5%
2	200-01040	CAP - 0.1 µF - MonRad
2	201-01050	CAP - 1 µF - Elect.
1	201-01062	CAP - 10 µF - 25V - Elect.
1	800-00016	3" Jumper Wires (1 Bag of 10)

Table 1.3: Recommended Parts not Included in the Javelin Stamp Starter Kit

Quantity	Part Number	Part Description
1	750-00009	7.5 VDC Power Supply
1	900-00005	Parallax Standard Servo

2: Javelin Stamp Quick Start

This chapter will guide you through *getting started quickly with the Javelin Stamp*. Later chapters will show you more details about each feature you work with here. The easiest way to get started is to use the Javelin Stamp Demo Board. However, if you want, you can use a carrier board of your own design using the schematics in this chapter. This chapter's topics include:

- Connecting the Javelin Stamp Hardware
- Installing the Javelin Stamp IDE
- "Hello World" program for the Javelin Stamp
- Online documentation
- An IDE Debugger example
- A "Hello Circuit" program for the Javelin Stamp
- Trouble-shooting tips

Hardware Setup

If you are using the Javelin Stamp Starter Kit or the Javelin Stamp Demo Board, getting the hardware set up takes just a few steps:

- ✓ Plug your serial cable into an available COM port or COM port adaptor on your PC or laptop.
- ✓ Plug the 7.5 V DC Power Supply into a wall socket. DO NOT PLUG THE OTHER END INTO THE CARRIER BOARD YET.

Next, use Figure 2.1 as your guide to the following:

- ✓ Plug your Javelin Stamp into the Javelin Stamp Demo Board. Double check the figure to make sure you did not plug it in upside down. Once the Javelin Stamp's pins are all lined up with the holes in the socket, press down firmly with your thumb to make sure the Javelin Stamp is properly seated in its socket.
- ✓ Plug the serial cable into the DB9 connector labeled JIDE port on your Javelin Stamp Demo Board.
- ✓ Plug the 7.5 V DC Power Supply's barrel jack into the 6-24 VDC plug on the Javelin Stamp Demo Board.

2: Javelin Quick Start

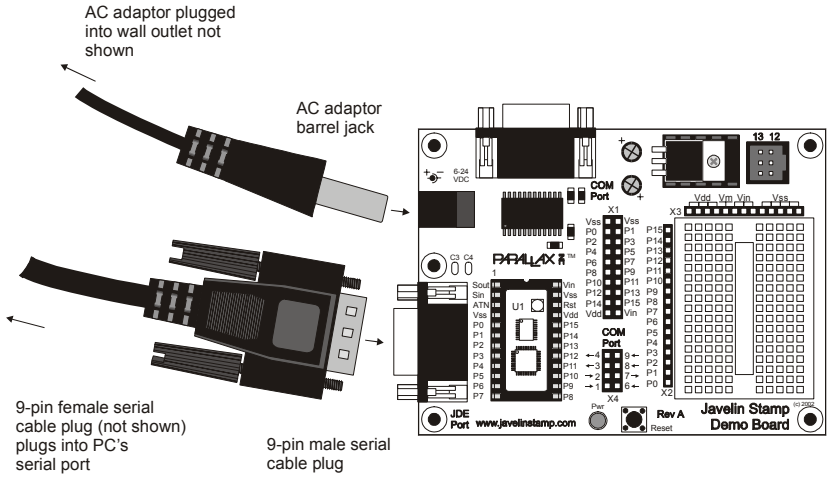


Figure 2.1 Connecting Power and Serial Cable to Javelin Stamp Demo Board

NOTE: Serial cable is a "straight-through" cable. **Do not use a null-modem cable!**

Done?	When you are done with this, you can skip to the Installing the Javelin Stamp IDE section. The remaining material in this section details the electrical connections required for powering the Javelin Stamp and connecting the serial cable to the communications pins without a carrier board.

The Javelin Stamp's pin map and mechanical drawing is shown in Figure 2.2. Throughout this text, the Javelin Stamp's pin labels will be referred to as shown on this diagram. Keep in mind that pin labels correspond to numbered pins on the module. For example, the pins labeled Vin, Vss, and Vdd are used for connecting power to the Javelin Stamp. You can use this pin map to discover that Vin, Vss, and Vdd are pins 24, 23, and 21 respectively. Likewise, the general-purpose input/output pins (I/O pins) P0 through P15 correspond to pin numbers 5 through 20 in the figure. The active-low reset pin, RES, is pin 22, and the COM pins, SOUT, SIN, and ATN are pins 1 through 3 respectively.

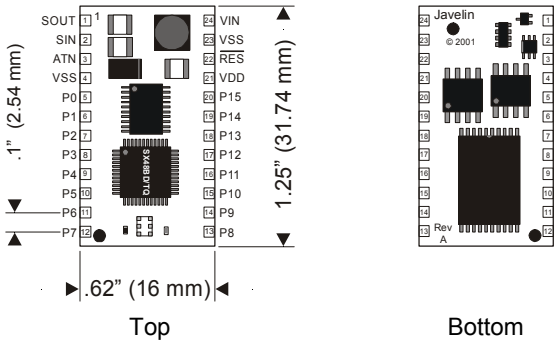


Figure 2.2
Javelin Stamp Mechanical Drawings and Pin Map

Figure 2.3 shows the recommended power supply circuit along with the recommended serial port wiring and reset switch. The power supply connections involve Vin, Vss, and Vdd (pins 24, 23, and 21). Vin should be connected to the positive terminal of the DC power source. Remember, this positive voltage must be between 6 and 24 V_{DC}. Vss (pin 23) should be connected to the DC power source ground or the negative battery terminal. Under this connection scheme, Vdd is a regulated 5 V_{DC} output that can supply anywhere between 30 and 90 mA depending on the current demands placed on the Javelin Stamp's I/O pins.

The recommended reset circuit shown in Figure 2.3 is a normally open pushbutton switch that, when pressed, connects RES (pin 22) to ground. When RES is driven low by pressing the pushbutton, the Javelin Stamp goes into a reset state. When the button is released, the Javelin Stamp starts whatever program it was running from the beginning. When the pushbutton is not pressed, the RES input is floating. There is an internal pull-up resistor onboard the Javelin Stamp that keeps RES at 5 V when the input is floating.

Sout, Sin, ATN, and Vss (pin 5 this time) of the Javelin Stamp are used for programming and debugging and are connected to the computer's serial port as shown Figure 2.3. Note that there is a loopback connection between pins 6 and 7 on the computer's serial port. This loopback is used to help the Javelin Stamp IDE auto detect the COM port that the Javelin Stamp is connected to. If you do not use this loopback connection, you will have to tell the software which serial port the Javelin Stamp is connected to. For information on how to do this, see Chapter 5: Using the Javelin Stamp IDE.

IMPORTANT
Do not try to use a null modem adaptor or null modem cable for connecting the PC to the programming port. You will not be able to program your Javelin Stamp if you are not using a straight through serial cable. When the cable is labeled serial cable, or serial extension cable, it is straight through. If it is labeled null modem, it will not work for programming the Javelin Stamp.

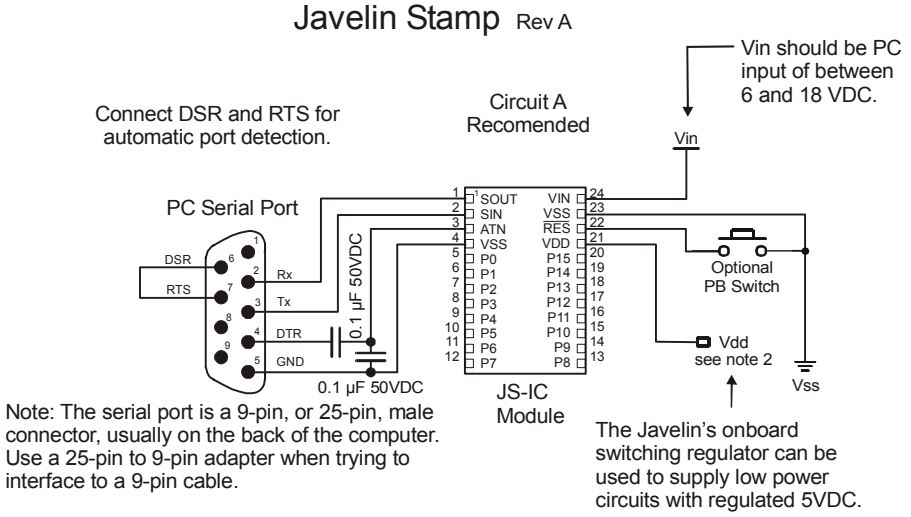


Figure 2.3 Javelin Stamp Com Port Connection and Recommended Power Connections

Figure 2.4 shows an alternate power supply scheme that can be used but is not recommended because of a 15 to 20 mA current draw penalty.

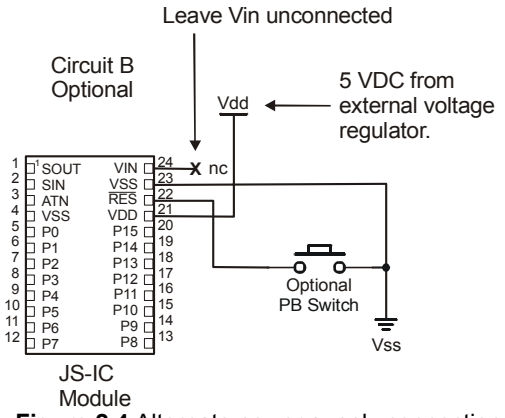


Figure 2.4 Alternate power supply connection diagram *(not recommended)*

Installing the Javelin Stamp IDE

The "IDE" in Javelin Stamp IDE, stands for Integrated Development Environment. The Javelin Stamp IDE is the software you will use to write, compile and download programs to the Javelin Stamp. The Javelin Stamp IDE also has a terminal window for sending messages to and receiving messages from the Javelin Stamp and a very powerful in-circuit debugging tool. These features are introduced here and examined more closely in Chapter 5: Using the Javelin Stamp IDE. For now we will focus on installing the software and taking it and the Javelin Stamp for a test drive.

Installation is simple, especially if you go with the default install. Selecting the default install options and installation path is especially useful if this is your first test drive of the Javelin Stamp.

INSTALLATION SHORTCUTS	**Web Download and Install:** Download the latest version of Javelin Stamp IDE Setup from the Javelin Stamp link at *www.parallax.com*. Save it to any folder and double click it to run. If your install was successful, skip to the section entitled: Test Program

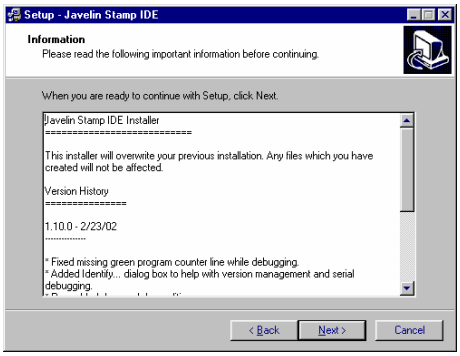
Figure 2.5 on the next page shows the windows you will see during the setup process, and each screen is summarized below.

(a) Setup wizard introduction for the Javelin Stamp IDE software. Click Next.
(b) Information screen contains version history, notes, and other helpful information. Review and then click Next.
(c) Destination directory. Especially if this is your first time using the Javelin Stamp, use the default directory. If you decide to install to a directory other than the default directory, make sure to consult the Class Path Considerations section in Chapter 5: Using the Javelin Stamp IDE. Click Next when ready.
(d) Review your install path, and click Next to install or Back to make changes.
(e) Confirm file association. Click Next if you are new to Java.

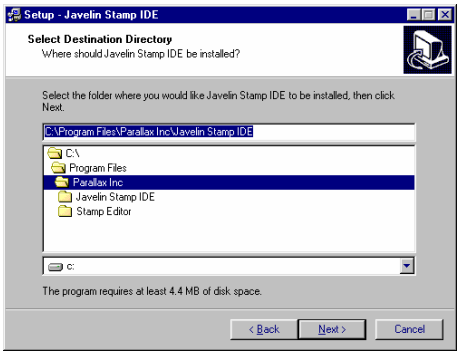
Java Programmers	Uncheck the checkbox next to "Associate Javelin Stamp IDE with .java extension" if you do not want the Javelin Stamp IDE to replace file associations that your existing Java development suite has established.

(f) As the Javelin Stamp IDE is installed, there is a blue bar that will show the progress and then automatically move to the next window after it reaches 100%.
(g) Setup is complete and successful message (not shown). Click the Finish button.

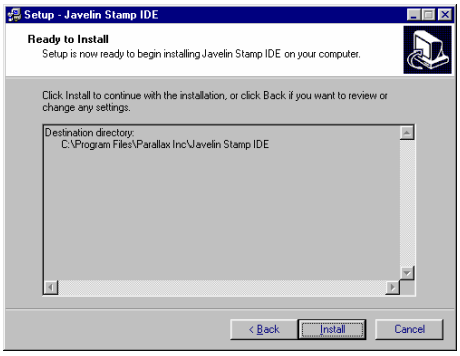
(a) Introducing the setup wizard



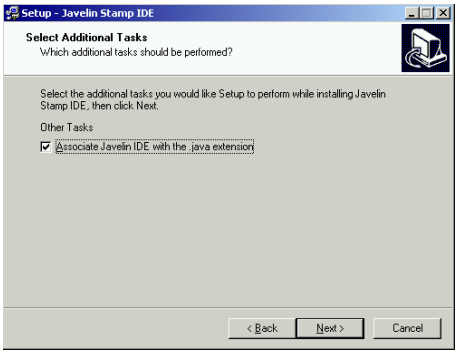
(b) IDE version information



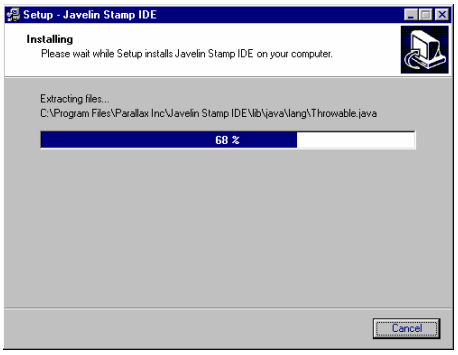
(c) Destination directory



(d) Confirm destination directory



(e) Confirm file association



(f) Watch the pretty blue bar get longer

Figure 2.5 Javelin Stamp Setup screens

Running the Javelin Stamp IDE and Loading a Test Program

The Javelin Stamp uses a language similar to Java but with special optimizations and features designed for embedded systems. The Javelin Stamp IDE will compile and link your code. This software downloads the compiled program to the Javelin Stamp. You can test your program, using the Javelin Stamp IDE to set breakpoints and examine variables. You can also make changes and go back to re-test your program until it does what you want it to do.

> *Once programmed, the Javelin Stamp remembers what it is supposed to do, so after you are done debugging your program, the Javelin Stamp will not need to remain connected to the PC – the Javelin Stamp will perform the last program you loaded every time it powers up. You can reprogram the Javelin Stamp up to1-million times.*

The first example we'll try is a simple "hello world" program (Program Listing 2.1 below). It will cause the Javelin Stamp to send a message back through the programming cable to the PC. The Javelin Stamp IDE's Messages window will display the message when it is received.

Program Listing 2.1 - Hello World!

```
public class HelloWorld {
  public static void main() {
    System.out.println("Hello World!");
  }
}
```

To run the Javelin Stamp IDE:

- ✓ Click the Windows Start Button
- ✓ Select Programs folder
- ✓ Select Parallax, Inc folder
- ✓ Select the Javelin Stamp IDE folder
- ✓ Select and click Javelin Stamp IDE icon

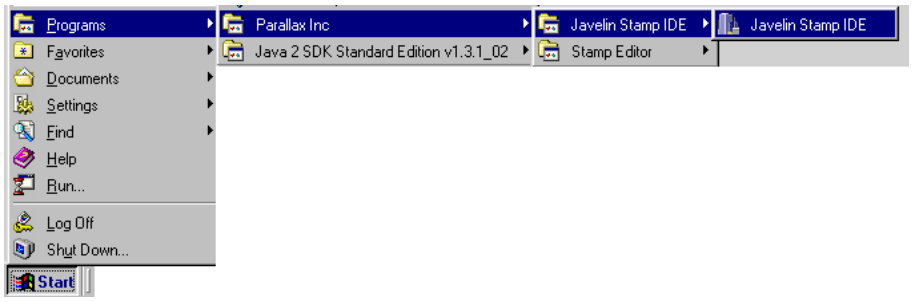


Figure 2.6 Running the Javelin Stamp IDE from the Windows Start menu.

2: Javelin Quick Start

The Javelin Stamp IDE will look similar to the window shown in Figure 2.7. To get to the point where you are ready to run the program, shown in the figure, follow these steps:

- ✓ Enter the program exactly as shown.
- ✓ Click the Save button.
- ✓ Save the file as **HelloWorld.java** in your projects directory. The path for your projects directory is:

```
C:\Program Files\Parallax Inc\Javelin Stamp IDE\Projects\
```

IMPORTANT	Your filename must always match the class name shown in the program, that's why this file must be saved as **HelloWorld.java**. (Java is case-sensitive therefore will distinguish the difference between lowercase and uppercase letters. Keep an eye out for this when typing in filenames or entering programs.) This name must match the class name, as well as the case of the letters, given in the line in the program that reads **public class HelloWorld{**

- ✓ Make sure your Javelin Stamp's power supply and serial cables are connected.
- ✓ Click the Program button.

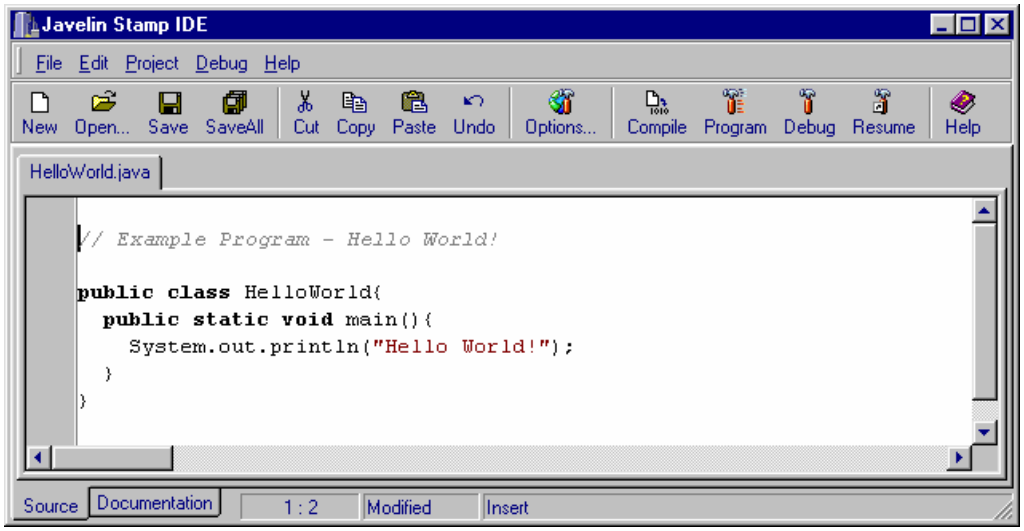


Figure 2.7 The Javelin Stamp IDE.

If the program was entered correctly, a small Progress window will appear in front of the Javelin Stamp IDE and display the following messages along with a graph of its progress:

- Linking Program
- Resetting Javelin Stamp
- Downloading Program
- Resetting Javelin Stamp

Next, the Messages from Javelin Stamp window shown in Figure 2.8 will appear. You can use this terminal window to view messages from the Javelin Stamp in the upper windowpane or send messages to the Javelin Stamp in the lower "transmit terminal".

Trouble Shooting	If you are having trouble getting the first program to run, turn to the section in this chapter entitled: Did That Work? – Trouble Shooting.

For the Javelin Stamp to receive messages, you have to program it to check for messages. The Javelin Stamp IDE installer placed an example file called TerminalTest.java in your projects directory. You can use this program to experiment with bi-directional Javelin Stamp communication using the Messages from Javelin window.

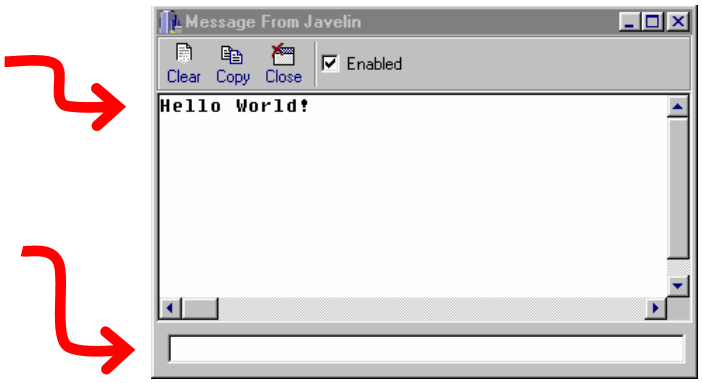


Figure 2.8 Messages from Javelin Window

Debugging Environment

Clicking the Debug Button in the Javelin Stamp IDE will open the IDE Debugger. This will be your best and most used tool for program and in-circuit debugging. By clicking the Memory Usage tab, you can see the display shown in the Figure 2.9. By clicking the Run button, you can make the Javelin Stamp send the PC another "Hello World" message via the serial cable. The Messages from Javelin Window will re-appear.

If you lose the Debug window, simply select Show Debug Window from the Debug menu. Similarly, if you lose the message window select Show Message Window from the Debug menu.

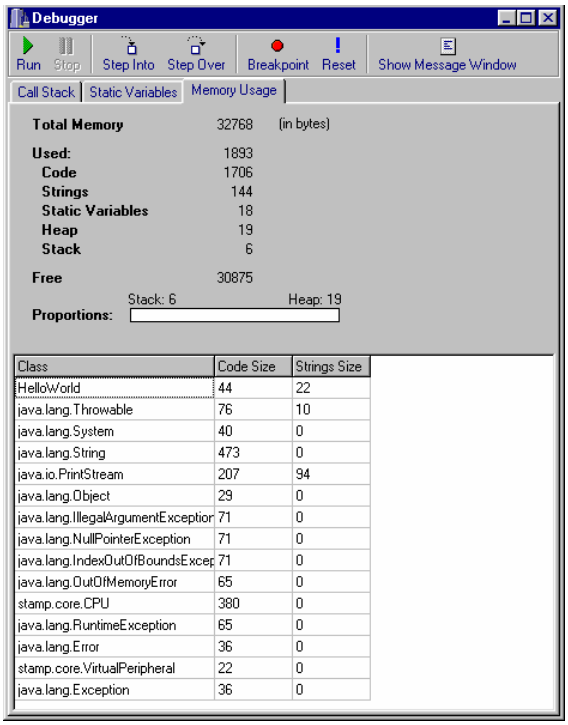


Figure 2.9 IDE Debugger

You can move around and resize your windows to a configuration that best suits you. Then you can save this configuration by selecting Save Desktop from the Project menu.

Here's another program to try with the Debugger:

Program Listing 2.2 - Count Down

```
import stamp.core.*;

public class CountDown {

  static int myVar;

  public static void main() {
    System.out.println("Commencing Countdown:");
    CPU.delay (10000);
    for(myVar = 10; myVar >= 1; myVar--) {
      CPU.delay(2000);
      System.out.println(myVar);
    }
    System.out.println("Liftoff!");
  }
}
```

- ✓ Enter Program Listing 2.2 into the Javelin Stamp IDE.
- ✓ Click the Debug button.

After the program loads, there should be two windows on your screen, the Javelin Stamp IDE and the Debugger.

- ✓ Click the Run button in the Debugger window to see what the program does. The Messages from the Javelin window will reappear and display a countdown from 10 to 1.
- ✓ Click the Reset button in the Debugger window to reset the program to its starting point.
- ✓ Click the gray left-hand margin in the Javelin Stamp IDE next to the **`CPU.delay(10000)`** command to set a breakpoint. The delay command will be highlighted in red with a red dot in the gray bar as shown in Figure 2.10.
- ✓ Set a second breakpoint, next to the **`System.out.println`** command.

2: Javelin Quick Start

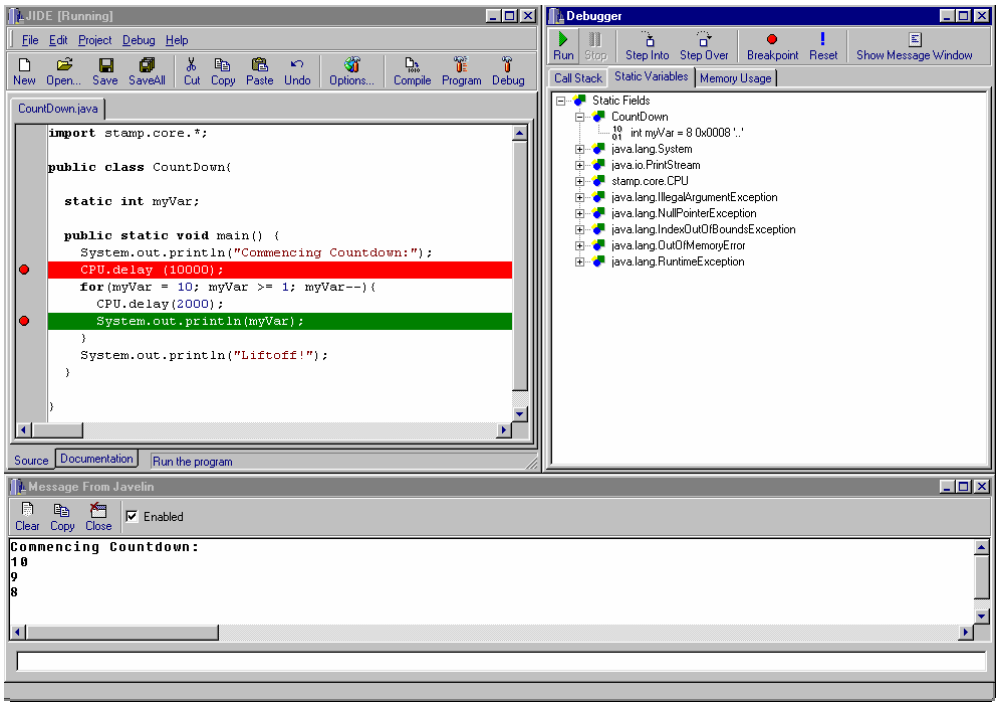


Figure 2.10 IDE, Debugger, and Messages from Javelin Windows all in use.

- ✓ Click the Run button several times and note how the green "current command" bar highlights the different breakpoints. You can also try the Step Over, Breakpoint toggle and Reset buttons. If you want to see the library classes and methods used by **`System.out.println()`**, you can click Step Over until you get to the command before the second breakpoint. Then, click Step Into. To get back to **`CountDown.java`**, just click Run, and it will take you back to the first breakpoint.

The Debugger doesn't just let you look at your program. You can also use it to look inside the Javelin Stamp as it executes code. For example:

- ✓ Click the Static Variables tab in the Debugger.
- ✓ Click the + next to Static Fields
- ✓ Click the + next to **`CountDown`**
- ✓ Note the value of **`MyVar`**

While developing applications, keep in mind that this powerful tool is at your disposal. The debugger is discussed in more detail in Chapter 5: Using the Javelin Stamp IDE.

Help File and Documentation

If you installed the Javelin Stamp IDE to the default directory, you can view the online help by entering:

```
C:\Program Files\Parallax Inc\Javelin Stamp IDE\doc\Javelin.chm
```

...into your web browser. You can also use the Javelin Stamp IDE to view Online Help:

- ✓ Click the Help button in the Javelin Stamp IDE

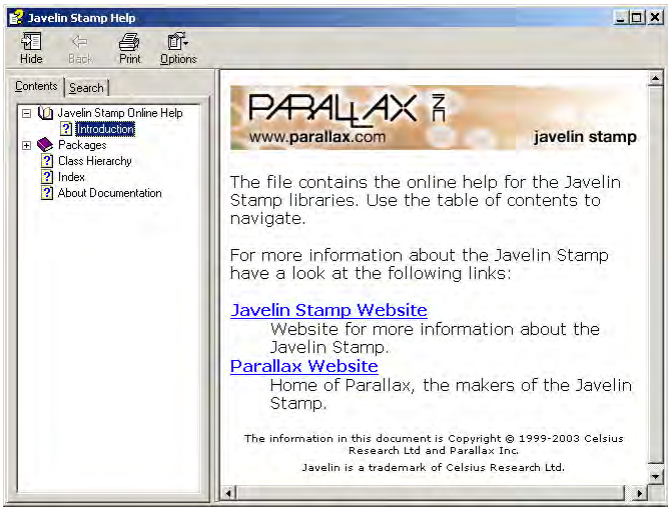


Figure 2.11
Help File and Documentation

The Javelin Stamp Help window will open. Use the sidebar's Table of Contents to browse the documentation on the library packages at your disposal as shown in Figure 2.11.

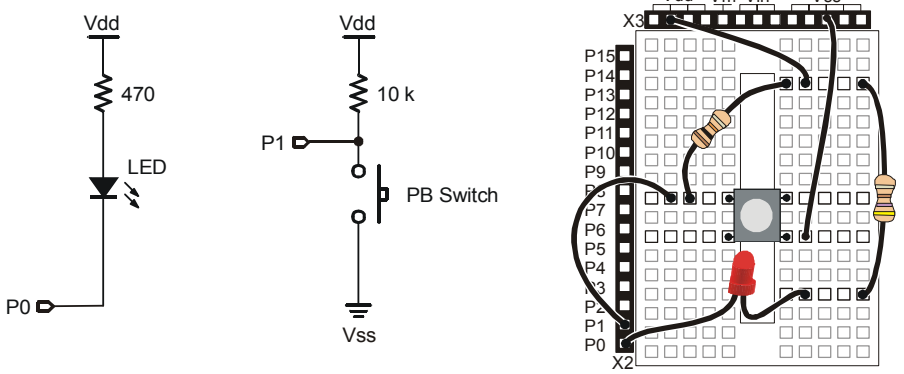
There are two links on the Introduction page, Javelin Stamp Website (www.javelinstamp.com) and the Parallax Website (www.parallaxinc.com). Please note that these two addresses are no longer active, and you may be redirected to www.parallax.com.

2: Javelin Quick Start

I/O Example

The real strength to the Javelin Stamp is its comprehensive I/O capabilities. With that in mind, why not try a simple I/O program before you continue with the rest of this manual? The schematic in Figure 2.12(a) shows a simple circuit with an LED and pushbutton. Since this is a "quick start" guide, an example of the circuit built on the Javelin Stamp Demo Board is also shown in Figure 2.12(b). For those of you unfamiliar with building circuits on a solderless breadboard, there is an introduction at the beginning of Chapter 4: Application Examples – Circuits and Programs.

Figure 2.12
Schematic and Breadboard Example for Program Listing 2.3



(a) Circuit (b) Breadboard

Program Listing 2.3 - Flash LED with Pushbutton

```
import stamp.core.*;

public class ButtonLED {

  static boolean P0 = true;

  public static void main() {

    while(true) {
      if (CPU.readPin(CPU.pins[1]) == false) {       // If button pressed
        P0 = !P0;                                    // Negate P0
        CPU.writePin(CPU.pins[0],P0);                // LED [On]
        CPU.delay(1000);
      }                                              // end if
      else {
        CPU.writePin(CPU.pins[0],true);              // LED [Off]
      }                                              // end else
    }                                                // end while
  }                                                  // end main
}                                                    // end class declaration
```

✓ Enter this program as shown and save it as **`ButtonLED.java`**.

- ✓ Click the Program Button.

You'll see the same downloading screen as before. When it completes the download, you can press the pushbutton to cause the LED to flash on/off at 5 Hz. When the pushbutton is released, the LED will not flash on/off.

Once the Javelin Stamp has been programmed, you can unplug it from your PC, turn the power off, move the Javelin Stamp somewhere else, reconnect the power and it will start running the program automatically. You only need the PC to program the Javelin Stamp. Once programmed, it will operate all by itself.

Did That Work? – Trouble Shooting

If the example worked as expected, great! You're ready to move on to the next section. If the example did not work, this section reviews some of the most common stumbling blocks and trouble shooting tips. Regardless of whether it's a compiler error or a download error, the error message will appear in a sub window in the IDE shown in Figure 2.13. Table 2.1 shows a list of the common problems and their error messages. Each problem and its solutions are discussed in this section.

Table 2.1: Problems and Error Messages

Problem	Error Message
Compiler Errors	[Error] HelloWorld.java…
Javelin Stamp Not Responding	Error [IDE – 0056] Possible Javelin on COM 1 did not respond. Error [IDE – 0054] Unable to find Javelin on any COM port.
Javelin Stamp Not Detected	Error [IDE – 0054] Unable to find Javelin on any COM port.

Compiler Errors

If you did not enter the program correctly (Java is case sensitive), the IDE might display an error message below your program. In Figure 2.13, the word **`Class`** should have been typed **`class`** in lowercase letters. You can double click the error message to get a hint from IDE as to what the error is. Notice how the word "Class" is highlighted. This is because the Java Error message that appeared below the program was double clicked.

Sometimes the majority of the code you typed will be highlighted when you click the compiler error. Check to make sure you didn't leave out one of the braces { }. Other times, there is more than one mistake. You might find that the next time you click the Program button, a different compiler error is displayed. Keep on fixing the errors. After each one is fixed, try clicking the Program button again. When all the errors are fixed, a "Compile successful" message will appear briefly at the bottom of the Javelin Stamp IDE window. Once the program syntax is correct, the Javelin Stamp will attempt to download the program.

Keep in mind that one bad line of Java code can create lots of errors, so always look at the first error in the list. After fixing that first error, try to run the program again (by clicking the Run button). It might run right away, or you may see more errors.

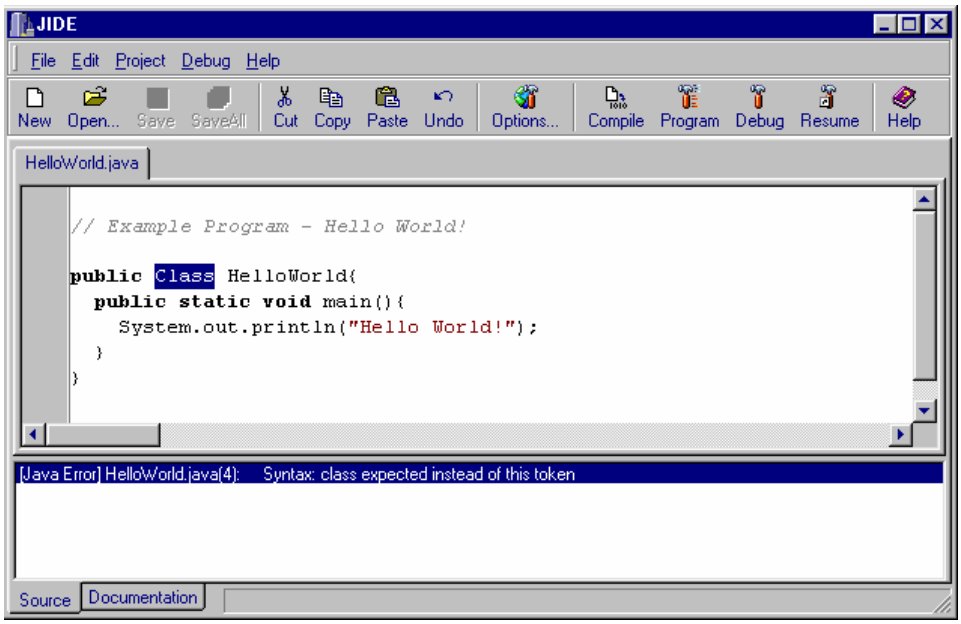


Figure 2.13
If you made a mistake...

Javelin Stamp Not Responding

If you get two messages, one of them stating that there is a possible Javelin Stamp found on COM 1, COM 2, etc, and the second stating that there is no Javelin Stamp found on any COM port, check your power supply. If everything compiles without errors, but you still have a communication problem, you'll see the progress indicator change to "Linking Program" and then "Resetting Javelin Stamp" – but then you'll see an error message (such as, "Javelin Stamp not found on serial port" or "Error reading from the serial port (timeout)".

In some cases a BASIC Stamp or other device may be connected to one of your other serial ports. The software may interpret these devices as Javelin Stamps that are not responding. You can instruct the software to look on a particular COM port by clicking the IDE's Options button. The Window shown in Figure 2.13 will appear. Next, click the Debugger tab. You can choose from the known COM ports by clicking the serial port field. There is a button with "..." on it next to the Serial Ports field. If you want to add a serial port to the list, click this button. Then enter the number into the Com# field and click add. You can also delete a COM port by clicking one of the known ports buttons in the list, then clicking Delete.

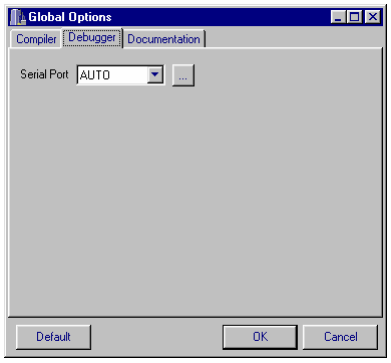


Figure 2.14
Debugger page of the Global Options window

Javelin Stamp Not Detected

If the Javelin Stamp IDE did not detect a Javelin Stamp on any of the known COM ports, try the following:

- Make sure your serial cable is properly connected to your Javelin Stamp/carrier board and to your computer's serial port.
- Verify that you are not using a null modem cable or adaptor.
- If you are using the Javelin Stamp Demo Board, make sure your serial cable is connected to the port labeled "JIDE port".
- Make sure other software such as a BASIC Stamp Editor/Debug Terminal is not using the COM port.
- If you have more than one COM ports on your computer, try connecting your Javelin Stamp to a different COM port. Make sure the Debugger shown in Figure 2.14 is set to either auto or to the correct port.
- If you have a Palm or other PDA, see below.

If You Use a Palm or PDA	Some software – notably hot sync programs for Palm computers and other handheld PDA's – will hold the serial port open even when you are not actively using it. • If you are using Microsoft's ActiveSync with your PDA you may also have a conflict. If your PDA is on a USB port, you will need to right click on the ActiveSync icon in your task bar. Then select Connection Settings and make sure you disable the COM port. • If your PDA uses a serial cable and you're using MS ActiveSync you can try specifically selecting a specific COM port by using the same method as above. If you still have problems, disable or exit the software.

Another thing to consider if your Javelin Stamp is not detected is that many older computers can't use COM1 and 3 (or COM2 and 4) at the same time. If you use a modem, for example, try disconnecting from the Internet (or other online service) and see if that helps. A serial mouse can also cause a problem since they are always in

use by Windows. If you have a mouse on COM1, COM3 may not be available for the Javelin Stamp's IDE or any other program.

If you are using any adapters, unusual devices, or odd cables on the serial port, you should double check to make sure the cable you're using connects straight through and passes at least the TX, RX, DTR, and GND signals as shown in Figure 2.3. If possible, try using a computer that will not require any special adapters and remove all unusual hardware connected to the serial port.

One final thing to recheck is power. Be certain that you have the Javelin Stamp adequately powered. If you don't, the Javelin Stamp IDE will not be able to communicate with it, and will report an error similar to a communications error.

If you tried all the suggestions in this section, and your Javelin Stamp still did not run the program, try one more thing: Install the software on a different PC, connect the Javelin Stamp and attempt to run the program. If this solves the problem, there may be some peculiarity in the BIOS settings of the first PC.

If all else fails, there are many ways to contact Parallax Technical Support for assistance; Resources and Technical Support on page 18.

Where to Next?

Now that you have a working system, you can take several paths to further your understanding of the Javelin Stamp. If you are fairly new to both Java and circuits, continue to the next chapter and follow through the chapters sequentially. BASIC Stamp programmers are also encouraged to take the same path because Chapter 3 is a first introduction to Java programming, and Chapter 4 circuit examples and Java examples to make them work. If you are an experienced Java programmer, skip to Chapter 4.

3: Beginner's Guide to Embedded Java Programming

This chapter contains explanations and examples to get you started, even if you have never programmed in Java before. Make sure to read the explanations and use your Javelin Stamp to run the example programs. Remember that all example programs in this manual are available for download from the www.parallax.com web site, and they also come with a standard Javelin Stamp IDE install in the `projects\examples\manual_v_1_0\` directory. Keep in mind that this is a starting point, and that many of the concepts and techniques introduced here are discussed in more detail in Chapters 6 through 8. Also, keep in mind that you can use this manual's table of contents and index to look up and learn more about the keywords, terms, and concepts introduced in this chapter.

The Class Wrapper and Main Method

There are several elements that must be present for a Java program to run:

- The program must be contained within a class definition
- The program must contain a main method
- Java commands are ended by semicolons

Think of the class definition as a wrapper for your program. After your class declaration **`public class ClassName`**, you must place an opening brace **`{`**. At the very end of the class, must also be a closing brace **`}`**. Your entire program, shown here as … is contained between these two braces.

```
public class HelloWorld {
   ...
}
```

The main method must appear within the opening and closing braces of the class definition. It is declared using the Java keywords **`public static void main()`**. As with the class definition, the main method has its own opening and closing braces, and within these braces you can place Java commands.

```
  public static void main() {
     ...
  }
```

Here is an example of an executable Java file with two commands within its main method.

Program Listing 3.1 - Hello World Again

```
public class HelloWorldAgain {
  public static void main() {
    System.out.println("Hello world!");
    System.out.println("Hello world again!");
  }
}
```

Always remember that the class name must match the program file name, and that both are case sensitive. Case sensitive means that capitalization matters. If you name your program **HelloWorld** but declare the class to be **helloWorld**, the compiler will give you error messages, and you cannot run the program until they are fixed.

Declaring Constants, Variables, and Arrays

Most programs work with two different types of quantities: variables and constants. Variables are numbers or characters that your program reads from an external source, computes, or changes in some way during execution. Constants are known at the time you write the program and never change.

Let's try declaring some variables of type **int**. In normal PC based Java, an **int** variable is 32-bits; in the Javelin Stamp, an **int** is 16-bits. A 16-bit **int** can be used to store signed integers between –32,768 and 32,767. To create an integer, you could write:

```
int abc;
```

However, the integer's contents are unknown until you assign a value to it:

```
abc = 10;
```

You can also declare an int variable and assign its value all in one step:

```
int xyz = 20;
```

To make a constant, simply use the **final** keyword with a variable declaration that includes an assignment. This prevents you from accidentally modifying the constant and also allows the compiler to generate code more efficiently since it knows the constant can't change. Here is an example constant:

```
final int invalidFlag = -1;
```

Program Listing 3.2 - Display Variables

```
public class DisplayVariables{

  public static void main(){
    int abc;
    abc = 10;
    System.out.println(abc);
    int xyz = 20;
    final int invalidFlag = -1;
    System.out.println(xyz);
    System.out.println(invalidFlag);
  }
}
```

We have already seen one method, the **main()** method. Additional methods that perform specific tasks can be added to a program, and they are introduced later in this chapter. If a variable is declared inside the main

method, another method can not use that variable. Likewise, if a variable is declared inside a special purpose method, other special purpose methods and the **main()** method cannot use that variable either. In Javanese, the "scope" of such a variable is called "local".

You can also declare a "global" or "class" variable, which is visible to all methods within the class. Instead of declaring the variable inside a method, you have to declare it outside of any method, but within the class. You also have to use the **static** keyword. Program Listing 3.3 shows an example of a class variable declaration. This will make the variable accessible to any method within the class.

Program Listing 3.3 - Global Variables

```
import stamp.core.*;

public class GlobalVariable {

  static int myVar = 20;

  public static void main() {
    System.out.println(myVar);
  }
}
```

The Javelin Stamp supports the following fundamental (primitive) data types: **boolean**, **byte**, **char**, **int**, and **short**. You will see some of them used in the examples in this chapter, and they are discussed in more detail in Chapter 6. Program Listing 3.4 declares and displays an example of each of these types.

Program Listing 3.4 - Display Primitive Types

```
public class DisplayPrimitiveTypes{

    static boolean logicValue = true;
    static char character = 'a';
    static short number = 900;
    static int anotherNumber = -2000;

  public static void main(){
    System.out.println(logicValue);
    System.out.println(character);
    System.out.println(number);
    System.out.println(anotherNumber);
  }
}
```

You can also declare arrays of primitive data types. Program Listing 3.5 declares and displays values from an **int** array.

Program Listing 3.5 - Example Array

```
public class ExampleArray{

  static int [] storeNumbers = {5000,4000,3000,2000,1000};

  public static void main(){
    for (int i = 0; i <= 4; i++){
      System.out.print(i);
      System.out.print("   ");
      System.out.println(storeNumbers[i]);
    }
  }
}
```

Performing Calculations

Once you have variables, it is natural to want to perform calculations with them. You can form expressions containing variables, constants, and literals. Consider this bit of code:

```
int result, temporary;
final int scale = 100;
temporary       = 14*2+3;
temporary       = temporary/10;
result          = temporary*scale;
```

The first two lines define variables and constants. The 3rd line performs a computation completely with literal numbers. In reality, the compiler will perform this computation at compile time. Since Java multiplies (and divides) before it adds (or subtracts), the result will be 31 (not 70). See Table 6.4 for a complete list of the order of operations.

The 4th line performs math with a variable "**temporary**" and the literal number, "10". Notice that it is common to use a variable to compute a new value for itself. This is so common that Java has a special way to write an expression like this:

```
temporary/=10;
```

Of course, you can use terms like: ***=**, **-=**, and **+=**, and other Java operators too. See Chapter 6 for a complete list.

The 5th line multiplies a variable and a constant and stores the result in a variable. You can write arbitrarily complex expressions and use parentheses to indicate grouping. So while it is a bit harder to read, you might have written:

```
result=(14*2+3)/10*scale;
```

This would compute the exact same result. Try Program Listing 3.6 to see these computations. Also, try experimenting with different values and note the results.

Program Listing 3.6 - Math Example

```
public class MathExample {

  static int result, temporary;
  final static  int scale=100;

  public static void main() {
    temporary = 14*2+3;
    System.out.println(temporary);
    temporary = temporary/10;
    System.out.println(temporary);
    result = temporary*scale;
    System.out.println(result);
    temporary /= 10;
    System.out.println(temporary);
    result = (14*2+3)/10*scale;
    System.out.println(result);
  }
}
```

Making Decisions

One common task in programming is taking action based on the value of a variable or an expression. For example, what if you wanted to print a message if a variable was greater than 100? You can do this with the **if** statement:

```
if (x>100)
  System.out.println("Limit exceeded!");
```

Notice that the test expression is in parentheses. You can also test for equality (two equal signs; **==**), less than (**<**), less than or equal to and greater than or equal to (**<=** or **>=**), and not equal (**!=**). These operators all return **boolean** values, either **true** or **false**. You can also put any expression that returns a boolean in the parentheses such as a boolean variable.

The statement after the parentheses will only execute if the expression in parentheses is true. If you want more than one statement to be executed if the condition is true, you'll need to surround the multiple statements with braces:

```
if (x>100) {
  System.out.println("Limit exceeded!");
  System.out.println("Please press reset");
}
```

It is allowable to use braces even if you have one statement. In fact, this is a good idea since you are less likely to mistakenly add extra lines later and forget the braces.

You can use the **else** keyword to specify a statement (or block of statements in braces) to execute if the condition is false. So:

```
if (x>100) {
  System.out.println("Limit exceeded!");
  System.out.println("Please press reset");
}
else {
  System.out.println("Process nominal.");
}
```

You may want to test several different conditions together. You can join boolean expressions with the **&&** (logical and) and || (logical or) operators. You can also reverse the sense of a boolean expression with the **!** (not) operator. This code fragment tests that **x** is greater than zero and also less than 100:

```
if (x>0 && x<100) System.out.println("In range");
```

For efficiency, the program will stop testing values as soon as it is certain what the end result is. For example, suppose **x** is 0 in the above example. The program will test **x>0**. Since this is not true (the test is **>** not **>=**) and the next expression is joined with an **&&** operator, the program will immediately stop testing and go to the next statement (not shown in the example). In this case, that isn't very important, but if the second part of the statement was a method call or had time consuming side effects this approach to evaluating boolean expressions can really come in handy.

The logical or (||) operator, of course, quits evaluating expressions as soon as one of the expressions returns **true**. You can write arbitrarily complex expressions and use parentheses to indicate grouping:

```
if (x>0 && (x<100 || runFlag==false)) . . .
```

Program Listing 3.7 demonstrates how the **if**/**else** code discussed earlier behaves when it encounters a **true** condition and when it encounters a **false** condition.

Program Listing 3.7 - Decision Example

```
public class DecisionExample{

  static int x = 50;

  public static void main(){

    if (x>100) {
      System.out.println("Limit exceeded!");
      System.out.println("Please press reset");
    }
    else {
      System.out.println("Process nominal.");
    }
```

```
    System.out.println(" ");

    x = 150;
    if (x>100) {
      System.out.println("Limit exceeded!");
      System.out.println("Please press reset.");
    }
    else {
      System.out.println("Process nominal.");
    }
  }
}
```

Repetitive Operations

One of the strengths of computers is that they can repeat steps over and over again. Java has many ways to control program loops. This sections introduces the **do…while** and **while** loops followed by discussion of the **for** loop and flow control using **break** and **continue**.

The **do…while** loop always executes once. At the end of each execution, the program decides if it should execute the loop again or continue with further processing. A **while** loop decides before executing any code. That means it is possible for a **while** loop to never execute if the condition required for it to execute is never met. Here is a **do** loop that counts to 10:

```
int i=0;
do {
  System.out.println(i);
  i=i+1;
} while (i<=10);
```

If you initialized the **i** variable at, say, 100, the loop would print 100, compute a new **i** (101) and then exit the loop since 101 is not less than or equal to 10. Adding one to a variable is so common that Java has a shortcut for it, the increment **++** operator. You can use the increment operator in place of **i=i+1**:

```
int i=0;
do {
  System.out.println(i);
  ++i;
} while (i<=10);
```

The **++i** expression adds one to the value of **i**. It also returns the new value for use in an expression (a fact the code above doesn't use). That means this could be written even more simply as:

```
int i=0;
do {
  System.out.println(i);
} while (++i<=10);
```

Technically, since this loop only has one statement, the braces are not necessary. However, it is a good idea to include them anyway to avoid future mistakes.

The same principles apply to a **while** loop:

```
int i=0;
while (i<=10) {
  System.out.println(i);
  ++i;
}
```

In this case, the test occurs before the loop. You don't want to use **++** in the loop since that would cause **i** to equal 1 during the first loop execution (unless that's what you wanted, but in this case you want it to match the **do** loop). If you change this example so that **i** starts out at 100, nothing will print since the loop will never execute. Program Listing 3.8 shows both loops doing the same thing, counting from 0 to 10.

Program Listing 3.8 - While Loop Examples

```
public class WhileLoopExamples {

  public static void main() {

    int i=0;
    do {
      System.out.println(i);
    } while (++i<=10);

    i = 0;                                          // Reset the value of i
    while (i<=10) {
      System.out.println(i);
      ++i;
    }
  }
}
```

Java also supports a more powerful loop construct known as a **for** loop. The **for** loop has three parts or clauses. The first clause executes code before the loop starts for the first time. The second clause tests for loop completion. The third clause executes after every loop. Semicolons separate the clauses. So if you wanted to count from 0 to 10 (as the above examples do) you might write:

```
int i;
for (i=0; i<=10; i++){
  System.out.println(i);
}
```

You can even declare the variable in the first clause (as long as you only need it within the loop):

```
for (int i=0;i<=10;i++)
  System.out.println(i);
```

The first clause defines the variable and sets it to zero. The second clause tests the variable and the third increments the variable at the end of each loop. If you want to control more than one statement, you should use braces as before (and you can use them even if you only have one statement in the loop).

There is nothing magic about the clauses – you can use any appropriate expression. For example, suppose you wanted to increase the count by 2 each time instead of one. You could write:

```
for (i=0;i<=10;i=i+2)
  System.out.println(i);
```

You can omit any of the clauses you don't need. For example, you might write:

```
int i=0;
for (;i<=10;i++) System.out.println(i);
```

You can even write endless loops using any of the three loop primitives:

```
for (;;) { . . . }
do { . . . } while (true);
while (true) { . . . }
```

Sometimes you want to exit a loop early. You can do this with the **break** keyword. For example:

```
for (i=0;i<=10;i++) {
  if (i == 3)
    break;
  System.out.print(i);
}
System.out.println("Skipping 3 and above");
```

The **break** statement works with any loop, not just **for** loops. Of course, you usually use **break** in conjunction with **if** since an unconditional **break** would just terminate the loop unconditionally.

You can also cause a loop to proceed to the next iteration (if any) by using **continue**. Suppose you wanted to count from 0 to 10, but you want to skip 5. There are many ways you might write this, here's one way:

```
for (i=0;i<=10;i++) {
  if (i==5) continue;                              // proceed to i=6
  System.out.println(i);
}
```

Program Listing 3.9 demonstrates **for** loops and the **break**, and **continue** keywords.

Program Listing 3.9 - For Loops

```
public class ForLoops{

  public static void main(){

    int i;
    for (i=0;i<=10;i++){
      System.out.println(i);
    }

    for (int j=0;i<=10;i++) System.out.println(i);

    for (i=0;i<=10;i=i+2) System.out.println(i);

    i=0;
    for (;i<=10;i++) System.out.println(i);

    for (i=0;i<=10;i++) {
      if (i == 3) break;
      System.out.println(i);
    }
    System.out.println("Skipping 3 and above");

    for (i=0;i<=10;i++) {
      if (i==5) continue;                          // proceed to i=6
      System.out.println(i);
    }
  }
}
```

Displaying Messages from the Javelin Stamp

Many of the earlier examples have demonstrated how you can use **`System.out.println`** to print a line to the Javelin Stamp debug window. You can also use **`System.out.print`** to print data without appending a new line. For example, consider this program:

```
public static void main() {
  System.out.print("Hello ");
  System.out.println("World");
}
```

The first statement prints "Hello" but does not start a new line. The second statement prints "World" directly following the first text and then starts a new line.

The **`print`** and **`println`** statements will accept most data types (for example, integers) and perform the necessary conversion to a **`String`**. So this small program is legitimate:

```
public static void main() {
  for (int i=0;i<10;i++) System.out.print(i);
}
```

Of course, there are times you might want to print the ASCII representation of a number (65, for example is a capital A). You can do this by casting the integer variable to a character, using **(char)** before the variable:

```
for (int i=65;i<70;i++) System.out.println((char)i);
```

Although it's not recommended for anything but a few initialization commands, in simple cases you can use the **+** (concatenation) operator to string items together, as in this example:

```
int t=100;
System.out.println("The threshold is " + t + " degrees.");
```

CAUTION	The Javelin Stamp does not support garbage collection and the compiler will create strings that the Javelin Stamp can never recover. If you use a command that uses the **+** (concatentaion) operator within a loop, you will run out of memory very quickly.

The above snippet has been re-written with separate print lines to achive the same result without having to worry about the lack of garbage collection.

```
int t=100;
System.out.print("The threshold is ");
System.out.print(t);
System.out.println(" degrees.");
```

Another approach is to use a StringBuffer object:

```
StringBuffer buf=new StringBuffer(32);                // 32 byte string
buf.append("The temperature is ");
buf.append('7');
buf.append('0');
buf.append(" degrees");
System.out.println(buf.toString());
```

In this way, you can use the **buf** variable again (unlike the compiler-generated temporary in the first example, which is not reused).

You can also use the **CPU.message** method to send a character array to the Messages from Javelin window. This requires fewer resources than the **System.out.print** method, but it is also less flexible since this method only accepts a character array. Here's an example:

```
String test="Parallax Javelin";
CPU.message(test.toCharArray(),test.length());
```

Notice that a **String** is not a character array, so the **toCharArray** call is required to perform the conversion. The second argument to **message** is the length of the array (which in this case is the same as the length of the **String**).

The **CPU.message** call does not automatically start a new line. You can include a new line (**"\n**") in the string to force a new line, as in this example:

```
String test="\nParallax Javelin\nWow!";
CPU.message(test.toCharArray(),test.length());
```

Notice that the **System** and **CPU** objects are **static**. You don't need to create these objects before using them. They are always present. However, if you don't use an **import stamp.core.*** or similar statement, you'll have to refer to the **CPU** object by its full name: **stamp.core.CPU**.

Keep in mind that the debug terminal only exists when the Javelin Stamp is physically connected to the PC. If the Javelin Stamp is running while disconnected from the PC, these messages are not displayed. Program Listing 3.10 shows these examples in action.

Program Listing 3.10 - Assorted Messages

```
import stamp.core.*;

public class AssortedMessages{

  public static void main() {

    for (int i=0;i<10;i++){
      System.out.print(i);
    }

    for (int i=65;i<70;i++) System.out.println((char)i);

    StringBuffer buf=new StringBuffer(32);           // 32 byte string
    buf.append("The temperature is ");
    buf.append('7');
    buf.append('0');
    buf.append(" degrees");
    System.out.println(buf.toString());

    String test="Parallax Javelin";

    CPU.message(test.toCharArray(),test.length());

    test="\nParallax Javelin\nWow!";
    CPU.message(test.toCharArray(),test.length());
  }
}
```

Sending Messages to the Javelin Stamp

Sending messages to the Javelin Stamp was first introduced in the Running the Javelin Stamp IDE and Loading a Test Program section of Chapter 2, see Figure 2.8. The **Terminal** allows you to either read a character, or determine if any characters are waiting to be read. Here is a simple example that just waits for you to press any key. The program doesn't care which character you press, so it doesn't record the value:

```
public static void main() {
  System.out.println("Press any key to continue");
  Terminal.getChar();
  for (int i=1;i<=10;i++) System.out.println(i);
  System.out.println("Press any key to exit");
  Terminal.getChar();
}
```

Program Listing 3.11 reads characters and converts them to uppercase. Remember that the messages from Javelin window can be used for bi-directional communication. Figure 2.8 in Chapter 2 shows the transmit terminal at the bottom of the Messages from Javelin Window. After running Program Listing 3.11, simply click the transmit terminal. Next, try typing a few characters. The characters will appear in the transmit terminal, and they will also be echoed in the messages window above. Immediately after each echoed character, you will also see the Javelin Stamp's converted character.

Program Listing 3.11 - Capitalize

```
import stamp.core.*;
public class Capitalize {

  public static void main() {
    char c;
    System.out.println("Begin");
    do {
      c=Terminal.getChar();         // Get character from keyboard
      if (c>='a' && c<='z') {       // Test if it's not a capital
        int tmp=(int)c;             // Create and assign 'tmp' the char c as an int
        tmp=tmp-32;                 // Convert lower case to upper by subtracting 32
        c=(char)tmp;                // Assign int tmp into char c
      }                             // end if
      System.out.print(c);          // Output character
    } while (c!=27);                // Do the above until escape key is pressed
  }                                 // end main
}                                   // end Capitalize
```

There is never a need to create a terminal object, it is always available. You must import **stamp.core** or use the full name **stamp.core.Terminal**. Keep in mind that the debug terminal only exists when the Javelin Stamp is connected to the PC. If the Javelin Stamp is running while disconnected from the PC, a debug window will not be available to you to accept input.

The **`getChar`** *method stops your program's execution until a key is ready for reading. The* **`Terminal.byteAvailable()`** *method returns* **`true`** *if there is at least one character waiting to be read. This method allows you to decide whether or not to perform other processing while waiting for keyboard input.*

Creating a Method

Once you start writing programs, you'll find there are things you want to use over and over in your program, but you don't want to keep rewriting the same program steps. Not only is rewriting the same steps tedious, it is not a very efficient use of the Javelin Stamp's resources. The solution for this problem is to use **`methods`**. Any method must be within a class, and not within another method. So, your program might look like this:

```
import stamp.core.*;
public class MyExample {

  // Custom methods could go here

  public static void main() {
    // Your main program
    // contains code that makes use of your custom methods.
  }

  // Custom methods could go here
}
```

The simplest method is one that performs a task, but does not expect information does not return any information. The **void** before the name of the method means that the method is not returning information, and the empty parentheses **( )** indicates that the method does not expect to receive any information either.

```
  static void startMessage(){
    System.out.println("This program performs some calculations.");
  }
```

A command inside the main method can then "call" this method, for example:

```
public static void main (){
  …
  startMessage();
  …
}
```

A method can also receive information and act on it, but not return anything. Here is a method that receives a number, multiplies it by 5, and then displays it. Note that a variable is declared to receive the value.

```
static void display5X( int i){
  i = i * 5;
  system.out.println(i);
}
```

A command in the main method that wants to display 5 X 5 could then send a 5 to the display5X method by placing a 5 inside the parentheses of the method call:

```
public static void main (){
  …
  display5X(5);
  …
}
```

A method can send back a value without receiving one. In this case, the method itself is declared to be an **int** value, but the parentheses are empty. If a method is sending back a value, it must do so using the **return** keyword.

```
static int sendBackValue(){
  i = 20;
  return i;
}
```

A command in the main method that wants to receive this value can do so by setting a variable equal to the method call.

```
public static void main (){
  …
  int x;
  x = sendBackValue();
  …
}
```

A method can both receive and return a value as shown here. Note that only one local variable was declared for the incoming variable. In this case, the method multiplies the value it receives by 9 and returns the answer.

```
static int performOperation(int j){
  j = 9*j;
  return j;
}
```

A command in the main method that wants to send this method a value (such as 7) and receive the answer would look like this:

```
public static void main (){
  …
  int y;
  y = performOperation(7);
```

```
    …
  }
```

A method can receive more than one value. Here is a method that averages five numbers:

```
public static int avg(int n1, int n2, int n3, int n4, int n5) {
  return (n1+n2+n3+n4+n5)/5;
}
```

Notice that the method is named **avg** and it expects five arguments and returns the avereage, an **int** value. The parentheses in the **return** statement's expression are required because Javelin Stamp would otherwise divide before adding.

You don't necessarily have to set a variable equal to the method to capture the value. For example, a command in your main method, or another method in the same class, can simply use **println** to display the value returned by the **avg** method:

```
System.out.println(avg(10,13,99,7,12));
```

You can also provide variables or expressions:

```
System.out.println(avg(10+x,13/y,z1,z2,12));
```

You can even combine this method with other items in expressions:

```
sigmaT=avg(a,b,c,d,e) + 100/x;
```

Program Listing 3.12 demonstrates the use of some of the methods and method calls just discussed.

Program Listing 3.12 - Method Example

```
public class MethodExample{

  static int sigmaT, a = 1, b = 1, c = 3, d = 4, e = 5;

  static void startMessage(){
    System.out.println("This program performs some calculations.");
  }

  static void display5X( int i){
    i = i * 5;
    System.out.println(i);
  }

  static int sendBackValue(){
    int i = 20;
    return i;
  }
```

```
  static int performOperation(int j){
    j = 9*j;
    return j;
  }

  public static int avg(int n1, int n2, int n3, int n4, int n5) {
    return (n1+n2+n3+n4+n5)/5;
  }

  public static void main(){
    startMessage();
    display5X(10);
    int x;
    x = sendBackValue();
    System.out.println(x);
    int y;
    y = performOperation(7);
    System.out.println(y);
    System.out.println(avg(10,13,99,7,12));
    sigmaT = avg(a,b,c,d,e) + 100/x;
    System.out.println(sigmaT);
  }
}
```

Creating and Using a Library Class

A method does not have to be in the same file as the program you are writing. You can call a method from your main method that exists in a separate file. That method can call a method in yet another file, and so on… You can also write classes that contain methods to perform various operations. Here is a simple library file that was saved as **`LibraryFile.java`** in the `projects\examples\manual_v1_0` folder. It has no main methods, just two methods that some other program can call.

Program Listing 3.13 - Library Class: Library File

```
package examples.manual v1 0;

public class LibraryFile{

  public static void countToTen(){
    for (int i = 0; i<=10; i++){
      System.out.println(i);
    }
  }

  public static void countToTwenty(){
    for (int i = 0; i <= 20; i++){
      System.out.println(i);
    }
  }
}
```

Program Listing 3.13 is an example program that you can run that uses the **`countToTen()`** method in LibraryFile.java. The programmer has to do a few different things to make the methods in LibraryFile

available. Before the class declaration, there is a compiler directive that tells the Java compiler to import all the files in the examples.manual_v1_0.* folder.

```
import examples.manual_v1_0.*;
```

This means that any class in the examples.manual_v1_0 folder can be accessed without having to refer to the class by path and name. This makes declaring a LibraryFile object easier because instead of writing:

```
  static Projects.examples.manual_v1_0 myLib;
```

You can simply write:

```
  static LibraryFile myLib;
```

This declaration creates an instance of a **LibraryFile** object named **myLib**. Now, you can use the methods in the instance of LibraryFile.java named **myLib**. How? Just use the term myLib, followed by a dot, followed by the method you want to call within LibraryFile.java. For example:

```
    myLib.countToTen();
```

If LibraryFile had **public** constants and variables, they would also be at the programmer's disposal using the same technique. This next example shows how to make a new LibraryFile object and call one of its methods.

Program Listing 3.14 - Library Class: Executable Uses Library File

```
import examples.manual_v1_0.*;

public class ExecutableUsesLibraryFile{

  static LibraryFile myLib;

  public static void main(){
    System.out.println("Library file displays count to 10:");
    myLib.countToTen();
  }
}
```

The ability to access reuseable code in library files is one of Java's most powerful features. Chapter 4 makes extensive use of library files. The library files in the `lib\stamp\core` folder contain library files with methods designed to make it easy to use the Javelin Stamp to read sensors, control circuit outputs, communicate with peripherals, and more. This folder and its collection of library files is referred to as a package, the core (or stamp.core) package in this case. The core package is introduced in Chapter 4, and documented in Chapter 9. There are many other packages available, such as java.lang, java.io, stamp.util, and so on. The library classes in these packages are discussed in Chapters 7 and 8. The library files that come with the Javelin Stamp install are also documented in HTML format and can be accessed following the Online Help link after clicking the Help button in the Javelin Stamp IDE. You can also view Online Help by navigating to:
`C:\Program Files\Parallax Inc\Javelin Stamp IDE\doc\Javelin.chm`.

Circuits and Example Code

This chapter has a few circuits and example program listings for you to try. There are two other chapters in this manual where you can find circuits and accompanying program listings:

- Chapter 2: Quick Start Guide
- Chapter 9: Javelin Stamp Hardware Reference

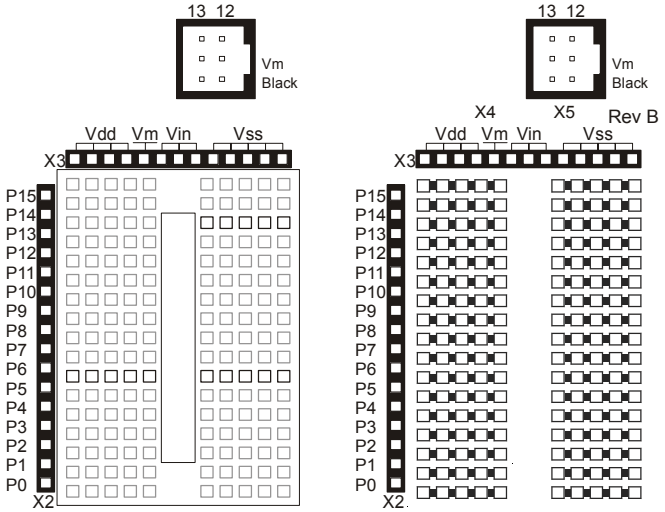
The beginning of Chapter 3 contains some recommendations for those new to Java on how to use the explanations and examples. Similar recommendations apply for material in this chapter, and they are listed below:

- All example programs in this manual are available for download from the www.parallax.com web site and also come with a standard install in the `Projects\examples\manual_v_1_0\` directory.
- Many of the concepts and techniques introduced here are discussed in more detail in Chapters 6 through 9.
- You can use this manual's table of contents and index to look up and learn more about the concepts introduced in this chapter.

About Solderless Breadboards

If you haven't built circuits on a solderless breadboard before, it's easy once you know what's underneath the surface of the breadboard. On the next page, Figure 4.1 (a) shows the top view of the breadboard and prototyping area on the Javelin Stamp Demo Board while (b) shows the connections underneath the breadboard. Each row of five holes on either side of the slot running through the center of the breadboard is electrically connected underneath. If you want to connect two components together, just plug into the same row of five sockets.

Figure 4.1 Javelin Stamp Demo Board Solderless Breadboards



Also note the sockets to the left of the breadboard labeled P0, P1…P15. These give you access to the Javelin Stamp's 16 general purpose I/O pins. The sockets above the breadboard are labeled Vdd, Vm, Vin, and Vss. Here is what each of these labels stands for:

- Vdd = + 5 V, used as the positive supply terminal for most of the circuit examples shown in this manual.
- Vm = motor voltage. You can connect this to either Vdd or Vin to supply the positive terminal for your servo port (header labeled X5 If you are using a wall mounted power supply (1000 mA recommended), connect Vm to Vdd. If you are using a 6 V battery pack like the one that comes with the Parallax Boe-Bot, connect Vm to Vin.
- Vin = the positive terminal of the unregulated input voltage coming from the DC Power Supply or battery pack. Be careful, DC Power Supplys labeled 9 V DC often deliver a much higher voltage, like 12 or even 15 V when the current draw is low.
- Vss = Ground, 0 V, the negative supply terminal for the examples shown in this manual.

Pushbutton and LED Revisited

Program Listing 4.1 below revisits Figure 2.12 in Chapter 2. When the switch is pressed, the program lights the LED for a predetermined time interval. The switch, on pin 1, connects to ground. There is a 10 k pull up resistor between the pin and +5 V. Therefore, when the switch is open, the input reads as a one. Pushing the switch causes the Javelin Stamp to read a zero.

Program Listing 4.1 - LED Push Button

```
import stamp.core.*;

public class LEDPushButton {

  // Define Variables & Constants
  final static int LED = CPU.pin0;              // To control the L.E.D.
  final static int SWITCH = CPU.pin1;           // To control the Button
  final static boolean ONSTATE = false;         // Button Pressed Down
  final static boolean OFFSTATE = true;         // Button Normal State

  public static void main() {
    CPU.writePin(LED,OFFSTATE);                 // Turn LED off
    while (true) {                              // Do loop forever
      if (CPU.readPin(SWITCH)==ONSTATE) {       // Was button pressed?
        CPU.writePin(LED,ONSTATE);              // Turn LED On
        CPU.delay(25000);                       // Wait (while LED on)
        CPU.writePin(LED,OFFSTATE);             // Turn LED Off
      }                                         // end if
    }                                           // end while
  }                                             // end main
}                                               // end LEDPushButton
```

Notice that instead of placing the pin constants in the program, **`LEDPushButton`** defines several constants (marked with the **`final`** keyword). This allows you to easily change the 1I/O definitions. The **`ONSTATE`** and **`OFFSTATE`** constants allow you to easily adapt the program to use a switch and LED that are active with a logical 1 state.

The main program uses **`writePin`** to make sure the LED is off. There is no need to explicitly set the pin to an output (or input). All pins are inputs when the Javelin Stamp resets. Any call to **`writePin`** (or other output methods) will automatically turn the affected pin (or pins) into an output.

Next, the main program loops forever using **`while(true)`**. This is not uncommon in embedded programs. The program tests for the switch closure, and when it detects it, it lights the LED, pauses, and turns the LED off again. The loop resumes waiting for another switch depression. Of course, if you hold the switch down, the loop will immediately turn the LED on again. This happens so fast, that it will appear the LED stays on as long as you hold the button down.

Digital to Analog Conversion

The Javelin Stamp can generate voltages on any of its output pins with the addition of some simple circuitry. The Javelin Stamp doesn't really generate a voltage. Instead, it generates a train of pulses that you can average with a resistor and capacitor as shown in Figure 4.2.

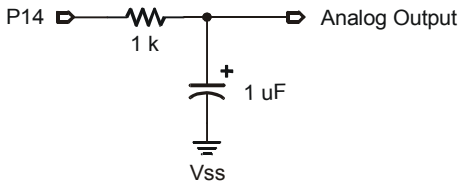


Figure 4.2
Circuit for use with DAC object

This program creates an analog output on pin 14. Then it ramps the voltage up by calling **`update`** inside a loop. A value of 0 generates a 0 voltage, and a value of 255 generates 5 V. Values in between generate a proportionally different voltage.

Because of the **`delay`**, you can watch the voltage change on an ordinary voltmeter. If you have access to a fast scope, reduce the delay value and watch the ramp on a scope.

Program Listing 4.2 - Make Voltage

```
import stamp.core.*;                    // Import Javelin's classes

public class MakeVoltage {              // class declaration

  public static void main() {           // main declaration
    DAC dac = new DAC(CPU.pin14);       // create new DAC object
    while (true) {                      // do while loop forever
      for (int i=0; i<255; i++) {       // loop 0v to +5v
        dac.update(i);                  // Update DAC with new voltage
        CPU.delay(1000);                // Delay
      }                                 // end for
      for (int j=255; j>=0; j--) {      // loop +5v to 0v
        dac.update(j);                  // Update DAC with new voltage
        CPU.delay(1000);                // Delay
      }                                 // end for
    }                                   // end while
  }                                     // end main
}                                       // end class declaration
```

Analog to Digital Conversion

Delta Sigma Analog to Digital Conversion is one of the more exciting new virtual peripherals on the Javelin Stamp. It lets you read an analog voltage from any I/O pin using just a few passive components. Figure 4.3 shows the circuit for use with Program Listing 4.3. You can connect any value between 5 V to 0 V, and the ADC object will return a number between 0 and 255 corresponding with the input voltage. This number corresponds to the duty cycle required to keep the voltage at P9 at the 2.5 V CMOS logic threshold.

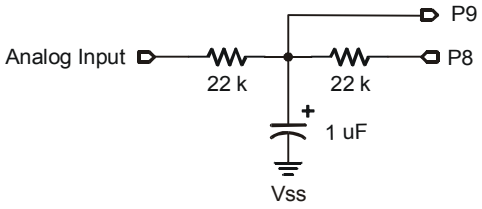


Figure 4.3 Circuit for use with ADC object

Program Listing 4.3 - ADC Test

```
import stamp.core.*;

  public class ADCTest {

  final static char CLS = '\u0010';
  static int ADCValue;
  static ADC voltMeasurer = new ADC(CPU.pin9, CPU.pin8);

  public static void main() {

    while(true){
      CPU.delay(5000);
      ADCValue = voltMeasurer.value();
      System.out.print(CLS);
      System.out.println("ADC value is: ");
      System.out.println(ADCValue);
    }                                                    // end while
  }                                                      // end main
}                                                        // end class declaration
```

Measuring Resistive and Capacitive Elements

rcTime has been used by BASIC Stamps to measure resistive and capacitive values for over 10 Years now. Resistive and capacitive sensors are very common, and **rcTime** offers an easy inexpensive way to get measurements from these sensors. Reading an **rcTime** value depends on either R or C remaining constant while the other component's value (a sensor) varies. In Figure 4.4, the 1 µF capacitor is constant, and the photoresistor varies with light exposure. As the value of R varies with light, the value of R×C varies as well. The fact that R×C varies is crucial, because it changes the speed at which the voltage at the capacitor's lower

plate responds to changes. In this example, the Javelin Stamp is used to measure this response time. The technique shown in the Program Listing 4.4 below is simple. It takes two commands to set up the **rcTime** measurement:

```
CPU.writePin(CPU.pins[4],true);
CPU.delay(10);
```

These commands apply voltage to the circuit so that the voltage at the RC connection approaches 5 V.

Then, the command:

```
dischargeTime = CPU.rcTime(10000,CPU.pins[4],false);
```

performs the measurement and saves it in the **dischargeTime** variable. **CPU.rcTime(10000,CPU.pins[4],false)** changes P4 from an output to an input and starts tracking time, waiting for the voltage at the capacitor's lower plate to drop below the 2.5 V logic threshold. This amount of time is proportional to R×C, and the math is discussed in the documentation for the **rcTime()** method in Chapter 9.

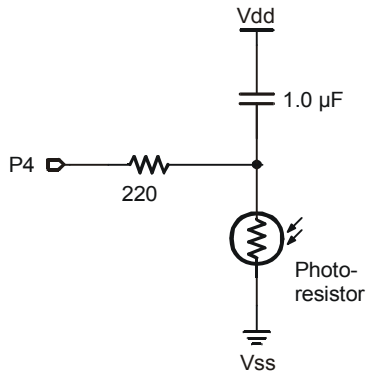


Figure 4.4
Circuit for use with rcTime

Program Listing 4.4 - Photo Resistor

```
import stamp.core.*;

public class PhotoResistor {

  final static char CLS = '\u0010';
  static int dischargeTime;
  static int chargeTime;

  public static void main() {

    while(true){
      System.out.print(CLS);
      CPU.writePin(CPU.pins[4],true);
```

```
      CPU.delay(10);
      dischargeTime = CPU.rcTime(10000,CPU.pins[4],false);
      System.out.print("RC rise time is: ");
      System.out.println(String.valueOf(dischargeTime));
      CPU.delay(10000);
    }
  }
}
```

Controlling a Servo with a Background PWM Object

The PWM object can be used to vary the brightness of a lamp or LED (assuming the device doesn't exceed the Javelin Stamp's drive capability). With appropriate drive electronics you can also use this command to control the speed of a DC motor or the brightness of lights that are too large to drive directly. However, the Javelin Stamp requires an external driver (like a transistor or FET) to handle the current required by a motor or a large light.

Sending Control Pulses to a Servo Motor

Many robotic and motion projects use servo motors to provide motive force. These motors are convenient since they accept a digital logic input and all the power control electronics are onboard. Typically, these motors don't rotate. Instead, they move between two extremes. However, there are many ways to modify the servos to achieve continuous rotation.

The servo's digital input requires a pulse. The details may vary depending on the type of servo you have. However, a typical servo requires a 1.5 ms high pulse to go to the center position (this corresponds to standing still for a modified servo). You must supply a pulse roughly every 20 ms to tell the servo what position you want or else the servo will not supply much force to hold its position (for most servos, at least). The range of pulse widths for a typical hobby servo range from 1.0 to 2.0 ms. Pulses shorter than 1.5 ms will cause the shaft to move in one direction away from the center and longer pulses will cause the shaft to rotate in the other direction. If the servo runs continuously, short pulses will cause the shaft to rotate in one direction and long pulses in the other. The more difference between the input pulse and the reference 1.5 ms pulse will affect the speed of the motor in this case.

If you are building this circuit on the Javelin Stamp Demo Board, use a wire to tie Vm to Vdd on the power header (X3). Then, plug the servo into the servo port labeled 12 on the X5 header. Make sure that the black (ground) wire for the servo lines up with the Black label next to the servo port.

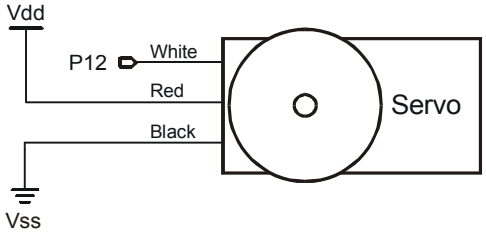


Figure 4.5 Circuit for use with DAC object

It is easy to use the **PWM** class to control a servo. This class allows you to specify the on time and the off time of a pulse. Program Listing 4.5 allows you to enter characters into the Messages from Javelin terminal to control the position (or rotation) of a servo. When you run the program, the Messages from Javelin window will appear and prompt you to enter one of three characters to adjust the servo's position to either right, left, or center. Click the field below the messages window and enter your characters there (See Figure 4.6).

Figure 4.6 Entering Messages into the Terminal Window

Program Listing 4.5 - Basic Servo Control

```
import stamp.core.*;

public class ServoControl {

  static PWM servo = new PWM(CPU.pin12,173,2304);

  public static void main() {

    System.out.println("Welcome to Servo Control: ");
    System.out.println(" r - right");
    System.out.println(" l - left");
    System.out.println(" c - center");

    while (true) {
      switch ( Terminal.getChar() ) {
      case 'r':
        servo.update(130,2304);
        break;

      case 'l':
        servo.update(220,2304);
        break;

      case 'c':
        servo.update(173,2304);
        break;
```

```
        }
      }
    }
}
```

You can also use the **PWM** class in the same way you use the **DAC** class. The output voltage is proportional to the ratio between the on time and the off time. Usually the **DAC** class is more convenient for this purpose.

Communicating with Peripheral ICs

The DS1620 (Figure 4.7) is a one of many ICs on the market that the Javelin Stamp can communicate with using the CPU class **shiftIn** and **shiftOut** methods. This particular IC reports the temperature it measures in ½-degrees Celsius increments.
Program Listing 4.6 makes use of a DS1620 class that comes in the stamp.peripheral.sensor.temperature package. Every library class listing has an HTML page that describes the methods you can call from the code you are writing. To view this HTML page:

- ✓ Click the Help button in the Javelin Stamp IDE.
- ✓ Click the Online Help Link.

– or –

use your we browser to view:
`C:\Program Files\Parallax Inc\Javelin Stamp IDE\doc\Javelin.chm`

- ✓ Click the stamp.peripheral.sensor.temperature link.
- ✓ Click the DS1620 link.

The first datum on this HTML page is the path to the DS1620.java file.
Program Listing 4.6 uses this information to import this file using the compiler directive:

```
import stamp.peripheral.sensor.temperature.DS1620;
```

The information in the constructor summary and constructor detail is used to declare a new DS1620 object:

```
DS1620 indoor = new DS1620(CPU.pin4,CPU.pin5,CPU.pin6);
```

Now, the methods of the DS1620 class, which are also described on the HTML page, are available to the programmer. For example, the **setTempLo()** method is called using the command:

```
indoor.setTempLo(68,'F');
```

and the **getTempF()** method call is nested inside another method call that appends the temperature returned to the end of a **StringBuffer** object named **msg**:

```
msg.append(indoor.getTempF());
```

So that you can see the **shiftIn()** and **shiftOut()** methods at work, Program Listing 4.7 performs the temperature measurement without making use of a library class. The circuit is the same for both programs. Use Figure 4.7 to build your circuit, and make the following I/O pin connections between the Javelin Stamp and the DS1620:

- dataPin to P4
- clockPin to P5
- enablePin to P6

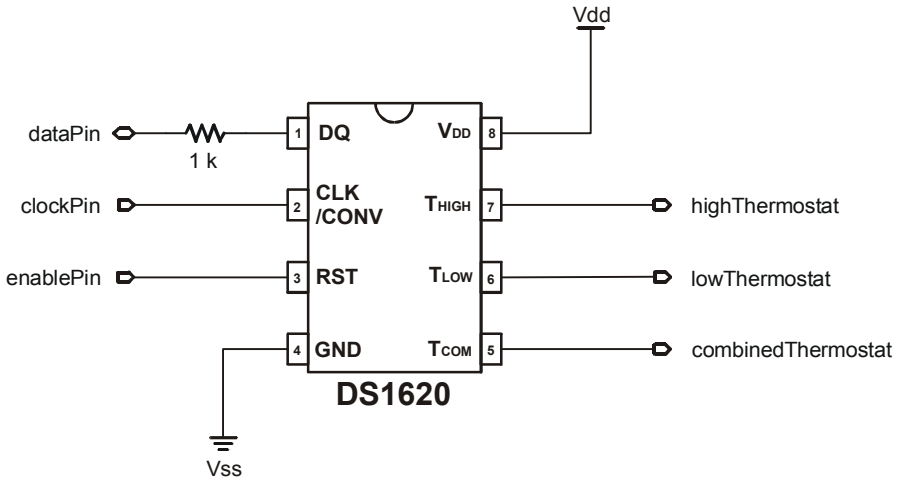


Figure 4.7
DS1620 Circuit

Program Listing 4.6 - Simple DS1620

```
import stamp.core.*;
import stamp.peripheral.sensor.temperature.DS1620;

public class testDS1620 2 {

  final static char HOME = 0x01;

  public static void main() {

    DS1620 indoor = new DS1620(CPU.pin4,CPU.pin5,CPU.pin6);
    StringBuffer msg = new StringBuffer(128);

    // set A/C thresholds
    indoor.setTempLo(68,'F');
    indoor.setTempHi(78,'F');
```

```
    while(true) {
      // get temps (build msg)
      msg.clear();
      msg.append(HOME);
      msg.append(" F \nInside.... ");
      msg.append(indoor.getTempF());
      msg.append(" F \n\nA/C..... ");

      // check A/C settings
      if (indoor.tempLo())
        msg.append("Heat");
      else if (indoor.tempOk())
        msg.append("Off ");
      else if (indoor.tempHi())
        msg.append("Cool");
      else
        msg.append("?   ");

      System.out.println(msg.toString());
      CPU.delay(10000);
    }
  }
}
```

Program Listing 4.6 communicates with a DS1620 without the use of a library class, and it demonstrates how numeric values are sent back and forth between the Javelin Stamp and the DS1620 using the **shiftIn()** and **shiftOut()** methods. Compared to
Program Listing 4.6, Program Listing 4.7 really highlights how much simpler and more powerful your code can be when you use a library class to do the job.

The segment of code below is from the **dsTemp()** method in Program Listing 4.7, and it is important because it uses **shiftIn()** and **shiftOut()** to communicate bi-directionally with the DS1620. Before communicating with the DS1620, its **enablePin** must be set to **true**. Then a **command**, hexadecimal AA for report temperature, is sent to the DS1620 using the **shiftOut()** method. Next, the variable **data** is set equal to the **shiftIn()** method. Since 9 bits of data are shifted in LSb-First into the 16-bit **data** variable, the shift right operator **>>7** is used to shift the data another 7-bits to the right. The extra shift is always necessary when shifting in LSb-first or shifting out MSb-first. For more information on the **shiftIn()** and **shiftOut()** methods, see Chapter 9.

```
CPU.writePin(enablePin,true);
CPU.shiftOut(dataPin,clockPin,8,CPU.POST_CLOCK_LSB,command);
data = ((CPU.shiftIn(dataPin,clockPin,9,CPU.SHIFT_LSB)>>7));
CPU.writePin(enablePin,false);
```

Program Listing 4.7 - Shift DS1620

```
import stamp.core.*;

public class ShiftDS1620 {

  // declare I/O pins connected to DS1620

  final static int dataPin   = CPU.pin4;
  final static int clockPin  = CPU.pin5;
  final static int enablePin = CPU.pin6;

  // Home character used for placing the cursor in the Messages from Javelin Window

  final static char HOME = 0x01;

  // DS1620 codes for initialization and for requesting temperature measurement

  final static int WRITE CONFIG  = 0x0C;
  final static int WRITE TL      = 0x02;
  final static int START CONVERT = 0xEE;
  final static int READ TEMP     = 0xAA;

  static int DSValue, sign, i, data;
  static int[] setup = {WRITE CONFIG,WRITE TL,START CONVERT};

   // Using a loop, the dsInit method (below) accesses values in the setup array
   // (above).  The shiftOut command is what clocks each value into the DS1620.

  static void dsInit(int config[]) {
    CPU.writePin(enablePin,false);
    CPU.delay(10);
    for (int i = 0; i < config.length; i++) {
      CPU.writePin(enablePin,true);
      CPU.shiftOut(dataPin,clockPin,8,CPU.SHIFT LSB,config[i]);
      CPU.writePin(enablePin,false);
    }                                                  // end for
  }                                                    // end dsInit

  // The dsTemp method accepts commands from the main routine and uses
  // the shiftOut() method to send this value to the DS1620.  Then the shiftIn()
  // method is used to shift in the temperature data from the DS1620.  The
  // positive or negative value is returned to the main routine.

  static int dsTemp(int command){
    CPU.writePin(enablePin,true);
    CPU.shiftOut(dataPin,clockPin,8,CPU.SHIFT LSB,command);
    data = ((CPU.shiftIn(dataPin,clockPin,9,CPU.POST CLOCK LSB)>>7));
    CPU.writePin(enablePin,false);

    sign = data >> 8;
    if ( sign == 1 ) {
      return -data;
    }
    else
      return data;
```

```
  }                                                    // end dsTemp

  // The main routine calls the dsInit method to initialize the DS1620,
  // then it gets the temperature value from the dsTemp method and displays it.

  public static void main(){

    dsInit(setup);
    while (true){
      System.out.print(HOME);
      System.out.println ("Celsius temperature: ");
      System.out.println(dsTemp(READ_TEMP)/2);       // Divide by 2 for deg-C
      CPU.delay(5000);
    }                                                  // end while
  }                                                    // end main
}                                                      // end class declaration
```

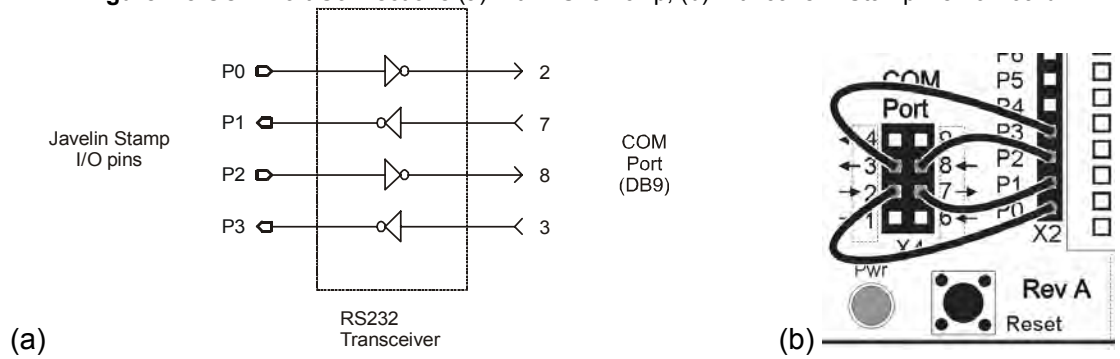
Communicating with Other Computers

Using the built-in **`Uart`** virtual peripheral, it is easy to communicate with a PC or other microcontroller. Since virtual peripherals always run in the background, you don't have to constantly poll for serial input. If input arrives while your program is doing something else, the virtual peripheral will buffer the data for you until you decide to process it. Each **`Uart`** object handles communication in one direction, so for two-way communications, you'll need two **`Uart`** objects.

*You can find more information about the **`Uart`** class in Chapter 9.*

Figure 4.8(a) shows the connection diagram for a full duplex hardware flow controlled UART. You can connect this to a serial cable via an RS232 transceiver like the MAX 233 or you can use the COM port connections on the Javelin Stamp Carrier board as shown in Figure 4.8(b). Program Listing 4.8 will use this connection to communicate with you through your PC's HyperTerminal program. If you are using the Javelin Stamp Carrier Board, make sure to connect the serial cable used by HyperTerminal to the port labeled COM Port.

Figure 4.8 COM Port Connections (a) with RS232 chip; (b) with Javelin Stamp Demo Board



If you want to build your own driver circuit, use
Figure **4.9** as a reference for the connections made in Figure 4.8. Keep in mind that this COM port is designed to connect to a computer's COM port. If you want to communicate with a peripheral instead of a PC, you will need to add a null modem adaptor. See the next section entitled Communicating with Peripheral Devices for more information.

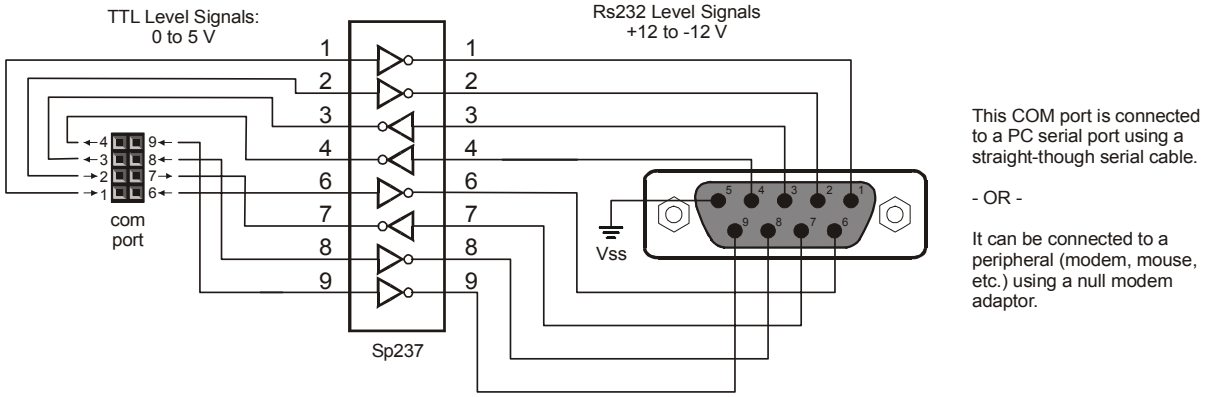


Figure 4.9: Javelin Stamp Demo Board COM port Connection Diagram.

If you are using one cable for both HyperTerminal and Javelin Stamp IDE, you will need to close the Javelin Stamp IDE after you load the program into the Javelin Stamp. Then, open HyperTerminal and start a new session. For connection, choose Direct to the COM port you will be testing. Select Properties from the File menu, then select the Configure icon from the Connect To tab. Choose the following settings:

- Bits per second 9600
- Data bits 8
- Parity No
- Stop bits 1
- Flow control Hardware

When your Javelin Stamp is running Program Listing 4.8 and HyperTerminal is connected (select Call from the Call menu), you can press and release the Javelin Stamp's Reset button to restart the program. Then, follow the prompts that appear in HyperTerminal for entering messages. If your JIDE port and COM port can be connected to two separate serial ports on your PC, use Debug and take a look at the contents of the buffer fields inside the UART objects.

Program Listing 4.8 - Bi-directional Communication with HyperTerminal

```
import stamp.core.*;

public class HyperTermCOM {                          // COM Port (9-pin serial)
```

```
  final static int SERIAL_TX_PIN = CPU.pin0;        //     2
  final static int SERIAL_RTS_PIN = CPU.pin1;       //     7
  final static int SERIAL_CTS_PIN = CPU.pin2;       //     8
  final static int SERIAL_RX_PIN = CPU.pin3;        //     3


  static Uart txUart = new Uart( Uart.dirTransmit, SERIAL_TX_PIN, Uart.dontInvert,
                          SERIAL_RTS_PIN, Uart.dontInvert, Uart.speed9600,
                          Uart.stop1 );

  static Uart rxUart = new Uart( Uart.dirReceive, SERIAL_RX_PIN, Uart.dontInvert,
                          SERIAL_CTS_PIN, Uart.dontInvert, Uart.speed9600,
                          Uart.stop1 );

  static StringBuffer buffer = new StringBuffer(128);
  static char c;

  static void bufferMessage(){
    c = 0xff;
    while (c != '\r'){
      if(rxUart.byteAvailable()){
        c = (char)rxUart.receiveByte();
        buffer.append(c);
      }
    }
  }                                                 // end bufferMessage

  public static void main(){
    do{
      buffer.clear();
      txUart.sendString("Type a message, the press enter: \n\r");
      bufferMessage();
      txUart.sendString("The message you sent was: \n\r");
      txUart.sendString(buffer.toString());
      txUart.sendString("\n\rDo you want to enter another message? (y/n) \n\r");
      c = (char) rxUart.receiveByte();
    } while(c == 'y' || c != 'n');
    txUart.sendString("Goodbye!\n\r");
  }                                                 // end main
}                                                   // end class declaration
```

Communicating with Peripheral Devices

You can use the Javelin Stamp to communicate with one or more asynchronous serial peripheral devices. Some of the more interesting and useful serial devices that can be incorporated into embedded applications include:

- LCDs
- Mice
- Camera modules
- GPS units
- Phone modems

Without the null modem adaptor, the COM port on the Javelin Stamp Demo Board is designed to be connected directly to a computer's serial port. In this configuration the port will behave just like any other serial peripheral device. If you want to connect the Javelin Stamp to a serial peripheral device, simply attach the null modem adaptor to the Javelin Stamp Demo Board's COM port, then attach the peripheral to the null modem adaptor. This makes, the Javelin Stamp Demo Board's COM port behave like a computer, and it can communicate with a serial peripheral device.

- ✓ Connect a serial modem to the null modem adaptor included in the Javelin Stamp Starter Kit.
- ✓ Connect the null modem adaptor to the Javelin Stamp Demo Board's COM port (not to the JIDE port)
- ✓ Use the previous example's circuit (Figure 4.8).
- ✓ You can use Program Listing 4.9 to call the Javelin Stamp at Parallax.

```
txUart.sendString("ATDT19166240160\r");
```

Note: If you do not live within the 916 area code, a long distance toll charge will apply. A simple test that you can do and avoid the long distance charge is to substitute your own phone number for the Parallax Javelin Stamp's phone number. Most modems will send a BUSY message back to the Javelin Stamp.

- ✓ Run Program Program Listing 4.8 to communicate with the serial modem.

Program Listing 4.9 - Modem Test

```
import stamp.core.*;

public class ModemTest {

// On Demo board's X4
// connect pin 0 to DB9-2
// connect pin 1 to DB9-7
// connect pin 2 to DB9-8
// connect pin 3 to DB9-3
// The Demo board has a level converter

  final static int SERIAL TX PIN  = CPU.pin0;
  final static int SERIAL RTS PIN = CPU.pin1;
  final static int SERIAL CTS PIN = CPU.pin2;
  final static int SERIAL RX PIN  = CPU.pin3;

  static Uart rxUart = new Uart( Uart.dirReceive, SERIAL RX PIN, Uart.dontInvert,
                           SERIAL CTS PIN,Uart.speed9600,
                           Uart.stop1 );
  static Uart txUart = new Uart( Uart.dirTransmit, SERIAL TX PIN,Uart.dontInvert,
                           SERIAL RTS PIN,  Uart.speed9600,
                           Uart.stop1 );

  public static void main() {

    /* You can use this phone number to call a Javelin Stamp
     * connected to a modem at Parallax.  A long distance charge will apply.
     *
```

```
     * A simple test that costs little or nothing is to use your own phone
     * number.  The modem typically sends the Javelin Stamp a BUSY message
     * since the Javelin Stamp is dialing the same number it is calling.
     */

     txUart.sendString("ATDT19166240160\r");

     // display modem's response (if any)
     while (true) {
       System.out.print((char)rxUart.receiveByte());
     }
  }
}
```

Figure 4.10 shows what's inside the null modem adaptor. Note that it re-routes transmit lines to receive pins and visa-versa.

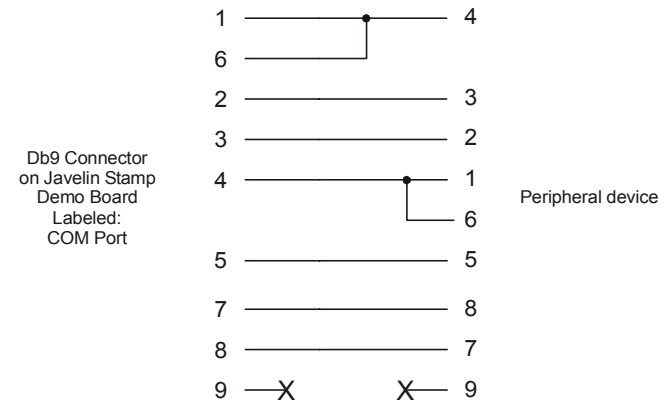


Figure 4.10 Null modem adaptor connection diagram

4: Application Examples – Circuits and Programs

The Javelin Stamp IDE (Integrated Development Environment) provides a work environment where you can write, run, and debug your Javelin Stamp programs. In addition, you can view the javadoc documentation from within the IDE.

Starting the IDE

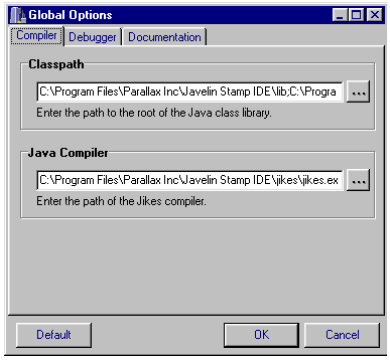
You can run the IDE by selecting the icon from your Start menu. From Windows, press on the Start button on your menu bar. Mouse through All Programs → Parallax Inc → Javelin Stamp IDE → Javelin Stamp IDE and the program will begin. You may wish to maximize the window (double click on the title bar, use the system menu on the left-hand side of the title bar, or use the maximize button to the right-hand side of the title bar).

By default, you'll see two command areas just below the title bar. The first area holds the main menu (which has items for File, Edit, etc.). The second area is a toolbar that has small icons to execute common methods. Below the tool bar, you'll see a tab that reads Untitled.java. This is the name of the file you are editing. If you open multiple files, each will have its own tab and you can switch between them by clicking on the tabs. The area below the tab is where the text will appear. The gray area to the left will contain indicators while debugging, as you'll see shortly.

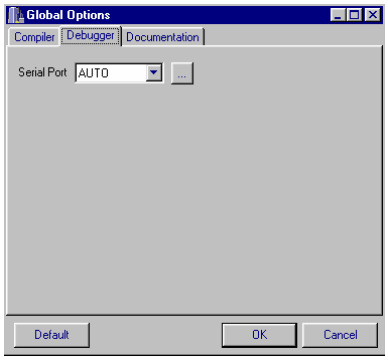
Setting Global Options

Before you get started, it is a good idea to review the option settings found within the Global Options… under the Project menu. The dialog (see Figure 5.1) that appears has two tabs. The first tab, Compiler, should contain the Class Path and the path to the compiler. Having the correct Class Path is vital so that the IDE can find the library files required for your programs. Be careful not to change the settings unless you are certain you know what you are doing (you'll learn more about changing the Class Path at the end of this chapter).

Figure 5.1 Global Options for IDE



(a) Compiler



(b) Debugger

The Debugger tab has a single option that allows you to set the COM port you've used when connected your Javelin Stamp. The IDE uses this port to communicate with the Javelin Stamp. You can press the Auto button and the IDE will attempt to detect the Javelin Stamp automatically.

If you change things inadvertently, you can push the Default button to restore everything to its original state. For now, the only thing you should change is the COM port setting on the Debugger tab.

Starting a Project

To start a project, you can just begin defining a class in the Untitled.java window. However, it is easier if you use the Insert Template under the File menu to insert a prototypical class into the editor workspace.

Here is the code inserted by the Insert Template command:

```
import stamp.core.*;

/**
 * Put a one line description of your class here.
 * <p>
 * This comment should contain a description of the class. What it
 * is for, what it does, how it use it.
 *
 * You should rename the class and then save it in a file with
 * exactly the same name as the class.
 *
 * @version 1.0 Date
 * @author Your Name Here
 */
public class MyClass {

  // Put variables here.
  static int myVar;

  public static void main() {
    // Your code goes here.
  }

}
```

You'll need to change **MyClass** to an appropriate name. You'll also want to alter the comments and **myVar** variable to suit your program. Java requires that each file have only one public class and that the class have exactly the same name as the Java file (including the case of the name). So if your class is **MyFirstClass**, you should save the file as **MyFirstClass.java** Save or Save As under the File menu.

You can also ask the IDE to help you write your code by invoking specific templates. If you press CONTROL+J while editing a file, you'll see a list of templates you can insert. For example, if you select the **for** *(count)* template, this will appear in your file:

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

}
```

If you've already typed a partial statement, pressing CONTROL+J will automatically insert the correct template without displaying a list. For example, if you enter **`if`** and then press CONTROL+J, the IDE will automatically insert the code template for **`if`**.

Table 5.1 shows the available templates and the keywords that will automatically invoke them.

Table 5.1: Javelin Stamp Templates

Menu Item	Menu Item	Menu Item
Template	Keyword	Example
Array declaration	Arrayii	int [ ] = {1, 2, 3};

Class declaration	Class	public class {
		
		}

Class declaration (with extend)	Classes	public class extends Object {
		
		};

Complete program	Program	See above example

For statement	For	for (; ;) {
		
		}

For statement (count)	Forc	for (int i = 0; i <; i++) {
		
		}

If	If	if ( ) {
		
		}

If else	Ife	if ( ) {
		
		}
		else {
		
		}

Table 5.1: Javelin Stamp Templates

Menu Item	Menu Item	Menu Item

Try/catch	Tryc	try {
		
		}
		catch ( ) {
		
		}

Try/catch/finally	Tryf	try {
		
		}
		catch ( ) {
		
		}
		finally {
		
		}

While	While	while ( ) {
		
		}

Do while	Whiled	do { } while ( );

Switch statement	switch	switch ( ) {
		case a:
		break;
		
		case b:
		break;
		}

Switch statement (with default)	switchd	switch ( ) {
		case a:
		break;
		
		case b:
		break;
		
		default:
		
		}

Table 5.1: Javelin Stamp Templates

Menu Item	Menu Item	Menu Item

Method declaration	method	/**
		*
		*
		* @param
		* @return
		*/
		void () {
		
		}

Method declaration (public)	methodp	/**
		*
		*
		* @param
		* @return
		*/
		public void () {
		
		}

Method declaration (with throws)	methodt	/**
		*
		*
		* @param
		* @return
		*/
		public void () throws Exception {
		
		}

Field declaration	field	/** * */ int ;
Debugging output	debug	System.out.println("");

5: Using the Javelin Stamp IDE

Building your Program

There are several ways to build your program depending on what you want to do with it. On the Project menu you'll find five important menu items:

- **`Compile`** – This option simply converts your source code into a class file. This will catch any compile-time errors, but it won't send any code to the Javelin Stamp.
- **`Link`** – Linking takes all the class files referred to by your program and binds it together for transmission to the Javelin Stamp. However, it doesn't actually send any code to the Javelin Stamp either.
- **`Program`** – This is the most common option. It compiles, links, and downloads your program to the Javelin Stamp.
- **`Debug`** – This command is similar to the Program command, but it also adds the necessary code that allows the IDE to debug your program.
- **`Resume Debug`** – If you are debugging a program and you get interrupted (perhaps you shut your computer down and restart it later), you can start a new debugging session without having to recompile, relink, and download. This does not resume your previous debugging session. It simply allows you to start a new one without reprogramming the Javelin Stamp.

When you use the Program option, the Javelin Stamp will run the program by itself. If you use one of the Debug options, the Javelin Stamp will require commands from the IDE to execute, so you'll want to use Program before you detach the Javelin Stamp. The Compile and Link options are handy for testing your program's syntax before downloading it to the Javelin Stamp.

Dealing with Errors

Of course, sometimes you'll have compile-time errors (such as syntax errors) that will prevent any of the above commands from working. For example, suppose you left the **`static`** keyword off of the two variable declarations in the example program:

Program Listing 5.1 - My Test Class (Dealing With Errors)

```
// (This program contains intentional errors)

import stamp.core.*;

class MyTestClass {
// Put variables here.
int pause=5000;
boolean state=false;

  public static void main() {
    while (true) {
      CPU.writePin(CPU.pin0,state);
      state=!state;
      CPU.delay(pause);
    }
  }
}
```

The two variable declarations should really look like this:

```
// Put variables here.
static int pause=5000;
static boolean state=false;
```

When you try to compile, run, or debug the program a window appears at the bottom of the IDE. This window will contain four error messages (see Figure 5.2). The error message shows the type of error, the file name, the line number, and the actual error message. In complex programs, compiling one file may cause other files to compile, so pay close attention to the file name, as it may not be the same as the current file name.

Regardless of the file name, double clicking one of the error messages will take you to the part of your program where the compiler detected the error. Notice that this is not always the same place as where you created the error.

In this case, for example, the first error message is:

> The name "state" does not denote a class (static) variable

This error appears on the **`CPU.writePin line`**. However, the real error is not on this line. The mistake here is that the **`state`** variable is an instance variable while the **`main`** method is (by necessity) **`static`**. A **`static`** method can't directly access instance variables, so an error occurs. All by itself, there is nothing wrong with creating an instance variable named **`state`**, so the compiler can't guess that this line is in error. That's because syntactically it isn't in error. The only reason the line is incorrect is because the program uses the variable contrary to its declaration and the compiler detects the error when the program tries to use the variable. The real mistake, of course, is in the declaration and that is where you'll have to fix the program.

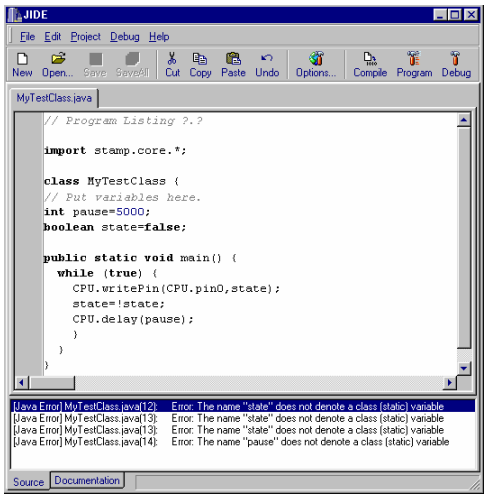


Figure 5.2 Error Messages

The other three errors follow the same logic. Even though there are four errors on three different lines, only two things require repair and they aren't on any of those lines at all. Of course, you need to make the two variable declarations **`static`**. You could also elect to have **`main`** create a new **`MyTestClass`** object and call an instance method (which could then directly refer to non-static fields). However, that's a major change to the program's design, not a repair.

Using the Debugger to Look Inside the Javelin Stamp

In a perfect world, you would write your program, download it to the Javelin Stamp, and be finished. In real life unfortunately, it isn't unusual for a program not to behave as you expected. Luckily, the Javelin Stamp's built-in debugger makes it very easy to troubleshoot misbehaving programs.

Of course, debugging won't help you locate syntax errors and other problems that prevent your program from compiling. You can find these by reading the messages the compiler and linker generate. However, just because the compiler thinks your program is correct doesn't mean the program does what you think it should. The compiler can accept a program that doesn't do what you want it to do (that is, your program contains an error in its logic). That's where the debugger comes into play.

To start the debugger (Figure 2.10), press the Debug button on the toolbar (or press CONTROL+D or select Debug from the Project menu). The debugger window that appears has several useful buttons and tabs:

- Run – This button starts your program executing under the debugger. The program will stop at a breakpoint, if any are set. You can set a breakpoint clicking in the gray area to the left of a program line, using CONTROL+B, or using the Toggle Breakpoint from the Debug menu (in the main Javelin window). A line with a breakpoint appears highlighted in red and has a red dot in the left margin.
- Stop – If the program is running, the Stop button will cause execution to halt as though a breakpoint had occurred.
- Step Into – When the program is stopped, this will cause one line of program code to execute. If the line makes a method call, the new stop location will be inside the called method.
- Step Over – When the program is stopped, this will cause one line of program code to execute. If the line makes a method call, the Javelin Stamp will attempt to execute the entire method before stopping again. Notice that some program lines make multiple method calls, so the stop position will appear not to move until you press the Step Over button multiple times.
- Breakpoint – Push this button to place (or remove) a breakpoint on the current line. When the program executes this line, the debugger will stop and wait for further user commands.
- Reset – Press this button to restart the program from the beginning.
- Show Message Window – This button displays the window the Javelin Stamp uses to display messages.
- Call Stack – This tab shows you the method calls that are currently active. So if the **`main`** method calls method A, and method A calls method B, you'll see **`main`**, A, and B in the display window when this tab is active. The window also shows local variables and fields.

- Static Variables – This tab allows you to examine the static variables of each class in your program. Click on a '+' sign to expand the display to see details, then click the '-' sign to hide those details again.
- Memory Usage – Use this tab to display statistics about how much memory your program is using.

The easiest way to learn to use the debugger is to load a simple example program and start the debugger. Use the *Step Over* and *Step Into* commands while examining the different tabs in the debugger window. Set a breakpoint on a line and use the Run command.

The compiler can detect your syntax errors, but it can't find mistakes in the logic of your program – only you can do that. That's why the IDE was programmed to have sophisticated debugging support to help you examine what your program is actually doing and make it easier to spot mistakes.

Once you clean up any compile errors you can begin to debug the program. Select Debug under the Project menu (or CONTROL+D) to begin the debugging process. Once the IDE downloads the program to the Javelin Stamp, you'll see a green bar indicating the first line of your program that will execute. You'll also see a debugging window (see Figure 5.3).

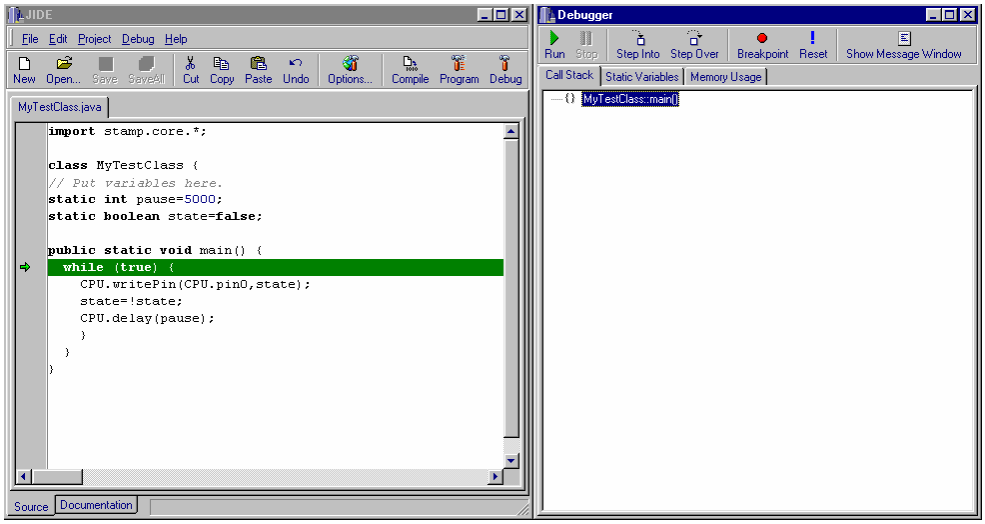


Figure 5.3 Javelin Stamp IDE and Debugger

The debugging window has a toolbar that mimics the method on the main Debug menu (covered shortly). It also has three tabs: Call Stack, Static Variables, and Memory Usage. The Call Stack tab shows the current method executing, along with the return path from methods that are in the middle of calling this method. The Static Variables tab shows you the name and value of all static variables. Finally, the Memory Usage tab allows you to examine how much memory your program is using and how much of that memory is code or data. You can also use the table at the bottom of the debug window to examine memory usage for each class in your

program. If the debug window is small, you may have to increase its size vertically (by dragging the window border) to make the table visible.

If you lose the debug window accidentally, you can always get it back by selecting Show Debug Window from the Debug menu. In addition, you can make the message window visible by selecting Show Message Window from the Debug menu. The message window shows any output your program sends using **`System.out`** or **`CPU.message`**. You can use **`System.out.println`** to write debugging messages to the message window to help you debug your program.

There are several ways to execute your program. If you select Run from the Debug menu (or the green arrow in the toolbar, or F9) then the program will execute normally. To stop the program, you can select Stop from the Debug menu (or the double red bars in the toolbar, or F8).

If you want the program to stop at a particular spot, you can do this by setting a breakpoint. Place the cursor on the line in question and select Toggle Breakpoint from the Debug menu (CONTROL+B), or use the stop sign on the toolbar. You can also click on the gray area to the left of the line. In any event, you'll see a red stop sign icon appear in that left-hand area to indicate the breakpoint. Repeating the step will turn the breakpoint off and make the stop sign icon disappear.

Sometimes you don't know where you want the program to stop. In that case, you can single step through the program. The Step Into menu item (on the Debug menu) causes your program to execute one line of source, and steps into method calls. Step Over is the same, except that any method calls will run to completion. The green execution bar will show you which statement will execute next. On the toolbar, these operations show a small box with an arrow pointing into the box (Step Into) or jumping over the box (Step Over).

While stepping through the program or if you are stopped at a breakpoint, you can always resume execution with the Run command. This will cause the program to continue until it ends or it encounters another breakpoint.

The only other command on the debug menu is Reset (CONTROL+F2). This causes the Javelin Stamp to prepare to run the program again. In other words, a Step Into, Step Over, or Run command will start the program at the beginning after a Reset.

An Example Debugging Session

Using the DebugExample (see Program Listing 5.1) type it in exactly as you see it (if you think you see a mistake, leave it as it is). Save the file in **`DebugExample.java`**. The intent of this program is to blink an LED with a 2 second off time and a half-second on time. The idea is to use a half-second (that's 5000 100us periods) time base and only turn the LED on every fifth count. Of course, you can wire the LED so that the LED will be off every fifth count and on the remainder of the time – the important point is that the LED will be in one state for a single 500 ms period and in the opposite state for 2 seconds.

You can run the program by selecting Program from the Project menu. However, it doesn't work. Why not? You might be able to find the answer by inspecting the program, but often debugging is easier.

To prepare for debugging, select Debug from the Project menu (or press CONTROL+D). Your screen should look like the one in Figure 5.4. The green bar and arrow in the source code tells you that you the next line that will execute. The Call Stack tab in the debug window shows you that you are in the **`main`** method.

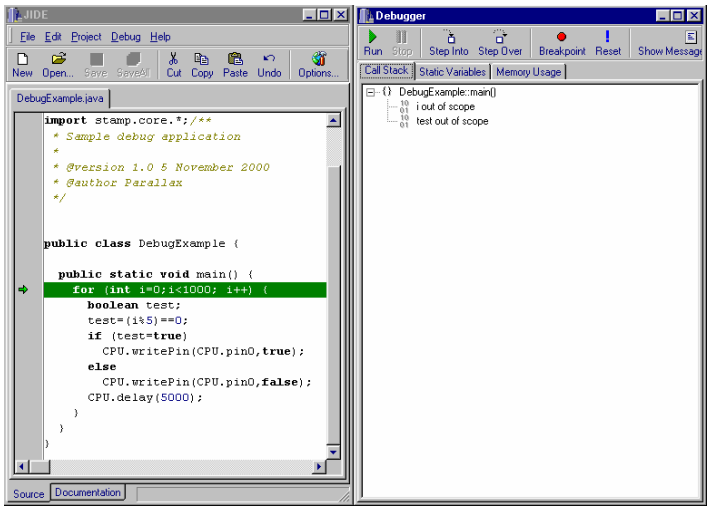


Figure 5.4 Stepping through Code

Use F7 to step through the program a line at a time. Notice that the Call Stack tab also shows the local variables (like **`i`** and **`test`**). Press F7 until you make one pass through the loop and notice the state of the local variables at each step.

On the first loop (where **`i`** is 0) everything seems to work, as you'd expect. On the second pass however, pay particular attention to the **`if`** statement. Press F7 until the green bar rests on the **`if`** statement (and **`i`** is equal to 1). Before executing the **`if`** statement, the **`test`** variable is **`false`**. That's right because 1 is not evenly divisible by 5 so **`i%5`** is not equal to 0. Now press F7 to step through the **`if`** statement. Suddenly, **`test`** is now **`true`** and the incorrect branch of the **`if`** executes. Do you see why? Careful observation of the **`if`** statement shows that there is only one equal sign! Instead of testing to see if **`test`** is true, this statement sets **`test`** to **`true`** and therefore assures that the **`else`** clause will never execute. The answer – or at least, one answer – is to change the single equal sign to two equal signs. On the other hand, you could rewrite **`main`** like this:

```
public static void main() {
  for (int i=0;i<1000; i++) {
    CPU.writePin(CPU.pin0,(i%5)==0);
    CPU.delay(5000);
  }
}
```

Editing Text

The IDE text editor window works the same as any other Windows editor. You can use the File and Edit menus as shown in Table 5.2 and Table 5.3.

Table 5.2: File Menu Commands

Menu Item	Command	Shortcut
New	Start a new document	CONTROL+N
Insert Template	Insert a sample class definition	CONTROL+J
Open…	Open an existing file	CONTROL+O
Reopen	Opens a recently used file	ALT, F, R
Save	Save the current document	CONTROL+S
Save As…	Save the current document with a new name	ALT, F, A
Close	Close the current document	CONTROL+F4
Print	Print the current document	CONTROL+P
Exit	Ends IDE	ALT, F, E

You can also use common Windows shortcuts to perform common editing operations shown in Table 5.3.

Table 5.3: Edit Menu Commands

Menu Item	Command	Shortcut
Undo	Undo the last editing action	CONTROL+Z
Cut	Remove the selection to the clipboard	CONTROL+X
Copy	Copy the selection to the clipboard	CONTROL+C
Paste	Paste the clipboard contents to the document	CONTROL+V
Select All	Select all text	CONTROL+A
Find and Replace…	Find or find and replace text	CONTROL+F
Find Again	Repeat last find operation	F3

Class Path Considerations

One of the most critical aspects of working with any Java or Java-like development tools is the CLASSPATH. Each time you name a class in your program, the compiler searches for the appropriate class file by searching the directories named in the CLASSPATH (you'll find more about this topic in Chapter 3).

It is crucial that the directories in the CLASSPATH refer to the correct class files, and not class files aimed at another target system (like the PC, for example). In addition, if you create your own libraries of code, you'll want to place the correct directories for that code in the CLASSPATH.

Selecting Global Options under the Project menu will give you a Global Options window. You can select the Compiler tab to view the CLASSPATH variable. You can directly change the string you find there if you like. It is simply a list of paths separated by semicolons.

The paths should be absolute (e.g., `c:\myclasses\lib1` instead of `..\lib1`).

However, it is easier to change the CLASSPATH by pressing the … button next to the path (see Figure 5.5). Here, you can change each part of the path separately. You can use the … button to browse your files and the up and down arrow buttons to alter the order of each directory in the CLASSPATH. The order is important, because the compiler begins searching with the first directory, and proceeds in order. Once it finds a suitable class file, it stops searching, so if two directories in the CLASSPATH contain class files named the same, the first one mentioned in the CLASSPATH will override any subsequent directories.

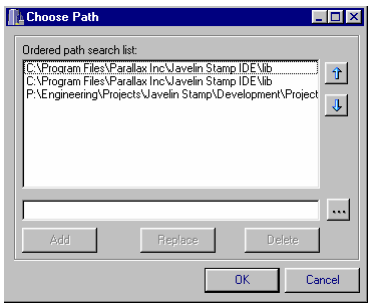


Figure 5.5
Class Path Settings

Working with Packages

If you make a class or a group of classes that you want to reuse, you might consider putting them in a package. First, at the start of the java files that contain your classes, you'll put a **`package`** statement. The convention is to use your inverted Internet domain name (for example, com.parallaxinc) to begin the package name. After that, you can use as many words as you like separated by periods.

For example, consider this class:

```
package com.parallaxinc.testlib;

public class doubler {
  private int val;
  public doubler(int v) { val=2*v; }
  public int value() { return val; }
}
```

This class (*doubler*) is part of the *com.parallaxinc.testlib* package. You need to save the file (or at least the class file) in a file that is in several subdirectories. In particular, the file name must be `com\parallaxinc\testlib\doubler.java`. This is a relative path name. The compiler will look in the current directory and in all the CLASSPATH directories for this directory structure. So imagine that your CLASSPATH had a single directory in it (`c:\classes`) and that the current directory is `c:\projects`. The compiler will look for `com.parallaxinc.testlib.doubler` in the `doubler.java` file. It will search for that file in:
`c:\projects\com\parallaxinc\testlib` and in `c:\classes\com\parallaxinc\testlib`

To use the **doubler** class, you'd need to refer to its entire name, or use an **import** statement. For example, you might write:

com.parallaxinc.testlib.doubler dbl = new com.parallaxinc.testlib.doubler(20);

Notice that you have to use the entire name every time you refer to the object. This is not very convenient so you'll usually use an import:

```
import com.parallaxinc.testlib.doubler;

doubler dbl = new doubler(20);
```

Remember, the packaged class must be in the correct directory tree and that directory tree's root directory must be in the CLASSPATH.

Working with Projects

You can organize your work into projects. From the main file of the project, you can select Make Project from the Project menu. You can also right click the file's tab and select Make Project from the resulting menu (if this option is gray, you have not saved the file yet).

Once you've made a project, the tab for that file will have a file folder icon to the left of the file name. One project can be active at a time. The active project will have a green checkmark in the file folder.

Projects are useful when you want your Java file to have its own options. The active project has its own private options that you can access by selecting Project Options from the Project menu. From here you can set the class path for compilation, the debugger settings, and packages you want to include in the javadoc documentation. You can also specify the directory where the IDE will create documentation.

6: Javelin Stamp Programmers Reference

This chapter details the Java language as it is used with the Javelin Stamp. Java, is a language developed by Sun Microsystems, and many find its syntax and structure similar to C++ (which is an object-oriented extension to C). However, there are two major differences:

1. Java is strictly an object-oriented system. You can use C++ without using objects, but Java requires you to use objects at all times.
2. Java handles some of the more error-prone parts of programming to reduce the burden on the programmer.

If you don't know object-oriented programming, don't worry. It does require you to change how you approach programming a little, but the payoff is well worth the effort. If you've programmed in virtually any other language, you'll find Java is simple to learn. If you've looked at books about Java before, you may have been put off by the complexity of the example programs. That's because most books concentrate of graphical user interfaces, which are complex by their very nature. In an embedded system, programs are usually much more straightforward.

Java Differences

If you are an experienced PC Java programmer – or you plan to read about Java – you should be aware that the Javelin Stamp uses a subset of Sun Microsystems' Java 1.2 class libraries. The Javelin Stamp also does not encompass certain variable types and object behaviors that PC Java programmers may expect to see. These differences are necessary to allow the Javelin Stamp to execute your programs on such a small computer and to ensure that embedded programs behave properly.

This manual shows you how to develop embedded applications using the Javelin Stamp. Experienced Java Programmers should consult Chapter 10, Summary of Java Differences before continuing. Java programmers are also encouraged to review the example programs in this manual for a clearer understanding of the scope of Javelin Stamp embedded projects and the way the Javelin Stamp utilizes a subset of Java for project development.

Getting Started

Every Java program consists of at least one public class. Of course, larger programs may consist of many classes of different types. To make your class executable, it must contain a static main method. You can generate a template from the IDE program by selecting Insert Template from the File menu. Be sure to replace **`MyClass`** in the generated code with a unique name. Save your new class file with the same name as the name you used in the class definition. For example, the class MyClass is saved as a file named MyClass.java..

Java statements can extend to multiple lines and must end with a semicolon. This is similar to C or C++ and is referenced as a code block. You can have nested blocks of code, in fact there is no limitation to how many blocks of code you can have nested within blocks of code.

What About the Braces?

In Java, curly braces surround groups of statements. This group is called a code block. Consider the **if** statement. This statement evaluates a boolean expression and executes the following statement if the expression is **true**. If you want to execute multiple statements, you must enclose them in braces so the compiler will see them as a single code block.

Of course, you can enclose a single statement in braces, if you like. In other words, these two **if** statements are the same:

```
if (x==0)
  System.out.println("zero value");

if (x==0) {
  System.out.println("zero value");
}
```

Using the second form helps prevent a common mistake. Often, you'll go back to add code to the **if** (or similar statement) and forget to add the braces, which are now necessary. For example:

```
if (x==0)
  System.out.println("zero value");
  System.out.println("Please restart");
```

The indenting of the code makes it appear that the **if** controls both **println** statements. However, this is not correct. The compiler doesn't actually pay attention to indentation – that's just to make your code more readable. In this case, the "Please restart" message will always appear no matter what the value of **x** is. The correct code is, of course:

```
if (x==0) {
  System.out.println("zero value");
  System.out.println("Please restart");
}
```

Some code must be grouped. For example, the code in a class declaration must be within braces. However, for **if**, **for**, **while**, and similar statements you can omit the braces if (and only if) the statement controls only one other statement. If there are multiple statements, you must surround them in braces. Notice that you don't place a semicolon after the closing brace.

The compiler doesn't really care about the indentation level. It also doesn't pay attention to where you place your braces. Many Java programmers follow the standard borrowed from the C language. This standard places the opening brace at the end of the line and then indents the following lines. The closing brace then appears on a line by itself, indented to the same level.

Some programmers have adopted one of two newer styles of writing braces. In both of these styles, both braces appear on their own lines. The only difference is how the braces indent. Consider these two examples:

```
if (x==0)
{
  System.out.println("Ready");
}

if (x==0)
  {
  System.out.println("Ready");
  }
```

Regardless of what style you use, you should pick one and stick to it. Using consistent braces and indentation will help you visually inspect your code for mismatched braces.

Variables, Types, and Constants

Variable store values, such as numbers or letters, or references to objects. Objects will be discussed later in the chapter. Each variable has a characteristic, called a data type, which describes what kind of data will be stored in the variable. The Javelin Stamp supports five fundamental data types, listed in Table 6.1 below.

Table 6.1: Fundamental Data Types

Type	Description
boolean	True/False value
char	8-bit ASCII *(not Unicode)* character ('\u00' : '\uFF')
byte	8-bit signed integer (127 : -128)
short	16-bit signed integer (32767 : -32768)
int	16-bit signed integer (32767 : -32768)

In Program Listing 6.1 you can see the variable declaration (`int i;`) and an assignment statement that computes a value and stores the result in `i`. Names are case-sensitive in Java, so it is possible (although not a good idea) to have another variable named `I`. Having two variables `I` and `i` makes reading the code much more confusing.

You can assign a value when you declare a variable as in this example:

```
int i=10;
```

You can also define multiple variables of the same type in a single line of code:

```
int i,j,k=33,loopctr=0;
```

You can create literal characters by using single quotes around any ASCII character. For example:

```
char stop='X';
```

Let's look at the Calculate class in Program Listing 6.1. Notice that there are two places where variables are declared. The **usecount** variable is declared outside of the **main()** method, but inside of the **Calculate** class declaration. The variable **i** is declared within the **main()** method. The difference between these declarations has to do with something called **scope**. Scope defines the area of your code where a declaration is visible. The **i** variable is visible only to the code in the **main()** method. Other methods in the **Calculate** class cannot access it. The **i** variable is created when the **main()** method is called and destroyed when **main()** returns.

The **usecount** variable is declared outside of any method, so it can be accessed by the methods within the class. This variable is declared at the class level. Variables declared at this level are called **Fields**. Fields are discussed in more detail later in this chapter.

Program Listing 6.1 - Calculate

```
public class Calculate {                // class Declaration

  int usecount;                         // Variable Declaration

  public static void main() {           // main Declaration
    int i;                              // Create i variable to store calculation
    i=33*9;                             // Perform calculation
    System.out.println(i);              // Print result
  }                                     // end main
}                                       // end class declaration
```

Constants

Sometimes you'd like to make a variable that has a constant value. For example, you might want to write:

```
int scale = 100;
```

However, let's say that your program should never change the value. It is a constant. In the line above, your program could, perhaps by accident, change the value of scale. The Java compiler has no way to know that the value should never change, and it might be able to generate better code if it knew that was the case.

To solve this problem, you can modify the type of the variable by declaring the variable to be **final** . This tells the compiler that the value of the variable is permanent and can't be changed. A final variable is always initialized with a value when it is declared, because you can't change the value after it has been declared. For example, the declaration:

```
final int scale= 100;
```

defines an integer constant equal to 100.

Table 6.2 shows some escape sequences used to generate special characters (like a single quote, or a new line). You can also use a C-style escape, **\ddd** (where **ddd** is the octal value of a character. String literals follow the same rules, but you enclose them in double quotes, not single quotes.

Table 6.2: Escape Sequences

Sequence	Meaning
\b	Backspace
\f	Form Feed
\n	New line
\r	Carriage return
\t	Tab
\u0010	Clear Screen
\\	Backslash
\'	Single quote
\"	Double quote
\xxx	Any character (xxx is octal number)

Number Bases

You can also specify literal integers in octal (base 8) or hexadecimal (base 16) form. Octal numbers have a 0 (zero) prefix, while hexadecimal (or hex) numbers have a 0x (zero x) prefix. This can be tricky. Consider this code fragment:

```
int x=010;
System.out.println(x);
```

The result printed is 8, because the leading zero marks the literal 010 as an octal number.

Expressions

When you write **x=10+3**, **x=x+1**, or even **x=0** you are assigning an expression to the **x** variable. Expressions combine variables and constants using operators (see Table 6.3).

Table 6.3: Basic Java Operators

Operator	Definition	Operator	Definition
++	Pre or post increment	<	Less than
--	Pre or post decrement	<=	Less than equal to
~	Bitwise invert	>	Greater than
!	Boolean invert	>=	Greater than equal to
*	Multiply	==	Equal to
/	Divide	!=	Not equal to
%	Remainder from integer division	&	Bitwise AND
+	Addition, String concatenation)	^	Bitwise exclusive OR
-	Subtraction	|	Bitwise OR
<<	Left shift	&&	Logical AND
>>	Right shift with sign extension	||	Logical OR
>>>	Unsigned right shift	?:	Conditional (ternary)

Consider this line of code:

```
x=5+3*2
```

The value of x depends on the order in which the expression is evaluated. If the addition is performed before the multiplication, the result would be 16. However, if the multiplication is performed before the addition, the result is 11. The correct answer is 11. You can see that the order in which the expression is evaluated is very important. Java addresses this issue by applying a set of *precedence* rules to the expression. It evaluates the parts of an expression starting with operators with the highest precendence. It then moves down the list until the entire expression has been evaluated. Table 6.4 shows the precedence of the various operators.

Table 6.4: Order of Operations

Priority	Operations
Highest	[ ] . (params) expr++ expr--
	++expr –expr +expr –expr ~ !
	new (typecast)
	* / %
	+ -
	<< >> >>>
	< > >= <= instanceof
	== !=
	&
	^
	|
	&&
	||
	?:
Lowest	= += -= *= /= %= >>= <<= >>>= &= ^= |=

Let's look at the way that Java evaluates the expression **`x=5+3*2`**. The operator with the highest precedence is located and evaluated. In our example, the multiplication operator (**`*`**) is higher on the list than the plus (**`+`**) operator. When this is evaluated, the expression becomes **`x=5+6`**. The operator with the next highest precedence is evaluated and the expression becomes **`x=11`**. There is nothing left to evaluate, so Java assigns the value of 11 to the variable **`x`** and moves on to the next line of code.

You can override the evaluation order of an expression by using parentheses. For example, if you wanted the answer to be 16, you could write:

```
x=(5+3)*2
```

When Java encounters operators of equal precedence, it evaluates the operators from left to right in the expression. For example, **`4+2+9`** produces the same result as **`(4+2)+9`**. It's a good coding practice to place parentheses in expressions that has any complexity.

Special Operators

For the most part, the Java operators will be familiar to you if you've used any other programming language. A few, however, may seem odd if you haven't used C or C++ before.

For example, the **`++`** and **`--`** operators can be confusing. These special operators increment or decrement (that is, increase by one or decrease by one) the variable they alter. Instead of writing:

```
foo = foo + 1
```

You might write:

```
foo++;
```

That doesn't seem like a big improvement, but you can also use these operators within other expressions. If the **++** occurs before the variable, the increment occurs before Java uses the value. If it occurs after the variable, the increment occurs after Java uses the value. You'll understand how this works if you consider the following code:

```
int x=10;
int y=3*++x;  // y = 33 and x=11
int z=2*x++;  // z = 22 and x=12
```

If you want to increase the value by more than just one, you can write:

```
x=x+10;
```
or
```
x+=10;
```

This also works with **+**, **-**, **/**, and ***** operators.

Another operator that is unusual is the conditional operator.

boolean expression ? true expression : false expression

This operator requires three arguments. The first is a boolean expression. If the expression evaluates to **true**, the result of the second expression is returned. Otherwise, the result of the third expression is returned. For example, the following statement assigns 0 to **x** if y is equal to 10, otherwise, x is assigned a value of 100:

```
x=(y==10)?0:100;
```

Notice that two equal signs is the operator that tests for equality **(y==10)**. A single equal sign is for assignment only.

You might wonder about the difference between the **&** and **&&** operators (or the | and || operators). The single character operators do bitwise operations. In other words, & takes the bits of its two arguments and ands them together. The double character versions only work on boolean values.

Comments

It is always a good idea to add comments to your code. This helps other people understand your program and might even help you figure out what you were doing when you return to your code a few weeks or months after you wrote it.

Java allows you to start a comment with two slash characters (**//**). After the two slashes, Java ignores everything else on that line. If you want to make multi-line comments, start them with **/*** and end them with ***/**.

However, **/**** is a special type of comment known as a Java Doc comment. A special program (javadoc) can scan Java source code and use special commands embedded in Java Doc comments to automatically create documentation in HTML or other formats.

Control Flow

All programming languages need a way to control the program's flow. Otherwise, your programs would be just a list of commands.

The Javelin Stamp supports decision statements such as **if** and **switch** and loop control statements **for**, **while** and **do**. These work nearly the same as their C counterparts. Program Listing 6.2 shows a simple program that uses a **for** loop. The first expression in the **for** statement sets the initial conditions. The second expression tests for the end of the loop, and the final expression modifies the loop variable at each loop.

Program Listing 6.2 - for Demo

```
public class forDemo {                              // class Declaration

  public static void main() {                       // main Declaration
    int i;                                          // Create 'i' integer
    for (i=0;i<10;i++) System.out.println(i);       // For loop, from 0 to 9
  }                                                 // end main
}                                                   // end class declaration
```

Even if you are used to C or C++, Java's strong typing can throw you a few curves. For example, in C++ you might write:

```
int t;
t = someroutine();
if (t)
  dosomething();     // call if t is not zero
else
  dosomethingelse(); // call if t is zero
```

This won't work in Java. Why not? The variable **t** is an integer but the **if** statement expects a boolean value. You'd have to write:

```
int t;
```

```
t = someroutine();
if (t==0)
  dosomething();     // call if t is not zero
else
  dosomethingelse(); // call if t is zero
```

Another place where Java differs from C is in the **break** and **continue** statements. With the Javelin Stamp, as in C, you use these statements to either end a loop (in the case of **break**) or go directly to the next iteration of the loop (for **continue**). However, these statements have extra features in the Javelin Stamp's language.

Consider this loop:

```
for (i=0;i<10;i++) {
  System.out.println(i);
    if (func(i)==3) break;
      if (i%2==0) continue;  // don't do any more for even
  // numbers
  System.out.println("Odd number");
}
```

The **break** statement, if executed, immediately terminates the loop. The **continue** statement, just moves on to the next iteration of the loop (in this case, that prevents even numbers from getting to the bottom of the loop).

Unlike C, Java allows you to include a label as the target of a **break** or **continue**. This lets you terminate or continue nested loops. For example:

```
Loop0:
for (x=1;x<10;x++) {
  for(y=1;y<20;y++) {
        .
        .
        .
    if (checkexit()==true) break Loop0;
  }
}
```

The **for** loop above, by the way, is functionally the same as this code:

```
int i=0;
while (i<10) {
   .
   .
   .
  i++;
}
```

There are many times when you want to test a value against several constants and take particular actions depending on the value. You could write a series of **if** statements. However, Java provides the **switch** statement, which is more succinct. Program Listing 6.3 shows an example of using **switch**. Notice that once a match occurs, the code executes from that point – even if it encounters another **case** statement. This allows you to cascade several cases that share the same code. However, most often you want each case to be separate and you'll want to write a **break** statement at the end of each **case**.

Program Listing 6.3 - Switch Demo

```
import stamp.core.*;

public class SwDemo {                                   // class declaration

  public static void main() {                           // main declaration

    while (true){                                       // do while loop forever
      System.out.print("Select 1-4: ");                 // Output
      switch (Terminal.getChar() ) {                    // run code based on getChar
        case '1':                                       // execute if '1'
          System.out.println("Number one");             // Output
          break;                                        // exit switch
        case '2':                                       // execute if '2'
        case '3':                                       // execute if '3'
          System.out.println("Either 2 or 3");          // Output
        case '4':                                       // execute if '4'
          System.out.println("Either 2, 3, or 4");      // Output
          break;                                        // exit switch

        default:                                        // execute if no match above
          System.out.println("You didn't enter 1-4!"); // Output

      }                                                 // end switch
    }                                                   // end while
  }                                                     // end main
}                                                       // end class declaration
```

Classes and Objects

Up to this point, we have talked about objects and classes without saying too much about what they are. You already know how to use data types such as int or char. Classes allow you to define new data types, also know as a reference types. In the example below, we have declared a class of type Thermostat. The Thermostat data type has fields, to store data and methods, that can perform operations on that data. Now you can declare a variable that uses this new data type:

```
int counter;
Thermostat myTemp;
```

A class does not actually do any work. That role is reserved for objects. An object is an instance of a class.

For example, consider a class that represents a thermostat used in a building's air conditioning system. The class might have fields to represent the current set point temperature and the current actual temperature. In addition, there might be methods that request an update of the current temperature or a manual override to turn the system on and off. You can see an excerpt of this imaginary class in shown below.

```
Class Thermostat   {
  private int id;
  private int setpoint=20;
  public Thermostat(int _id) { id=_id; }
  public void setTemp(int temp)   {
     . . .
  }
  public int getTemp()    {
     .
     .
     .
  }
```

All by itself, this class does nothing. If you want to represent a particular thermostat, you'll have to instantiate the object. First, you'll declare a variable of the object's type:

```
Thermostat t1;
```

This isn't an object yet; it is simply a reference to an object. What's the difference? The variable **t1** holds a reference (or pointer) to the **Thermostat** object. We haven't actually created the **Termostat** object yet, so the value of **t1** is null or undefined. To create (or instantiate) a new **Thermostat** object, you would write:

```
t1=new Thermostat;
```

This line of code creates a **Thermostat** object and stores a reference to the new object in **t1**. You might also write:

```
Thermostat t2=t1;
```

Now **t2** and **t1** refer to the exact same object. That means you can change the object using either **t1** or **t2** and it will have the same effect.

This has an odd effect when testing for equality. If you test to see if **t1** and **t2** are equal (using **==**) the result will be true if an only if the two references point to the same object. For thermostats, that is probably the right thing to do. On the other hand, consider objects like **String** (the built-in object for handling text strings). You don't care that the strings are the same object. You are more interested to know if the strings have the same contents. Using **==** tests to see if the variable refer to the exact same object, and **s1** and **s2** will not tell you whether the contents of the two strings are the same. Many objects (including **String)** provide an

equals method that tests for logical equivalence. Then, you can use a statement like **s1.equals(s2)** to test and see if the two strings have the same contents.

Methods and Parameters

The **equals** method is a common method that exists in every class. Of course, you can write your own methods. Each method belongs to a class and returns a value. Methods can also take arguments or parameters. You can have two methods in the same class that have the same name as long as they accept different parameters. For example, you might have a method known as **print** that accepts an integer argument and another one that accepts a **String**. From Java's point of view, these are two entirely different methods.

Methods return values (using the **return** statement). If you don't need to return anything, you can define the method as a **void** type. If you don't specify **void**, then you must use a **return** statement or you'll get a compile error.

Classes can contain special methods that have the same name as the class. These special methods are constructors and have no return type. They can, however, accept arguments. You can have multiple constructors with different argument lists.

Consider the simple class in Program Listing 6.4. Here the **construct** object has three fields. The **intval** field can store an integer value and the **strval** field stores a string. The **which** field tells which of the two values were set (if any). Notice there are three constructors. One takes no arguments (the default constructor). The other two take arguments of the appropriate type. Each constructor sets the correct field and the **which** field as appropriate.

Program Listing 6.4 - construct

```
// This program is a Library Class and must be called by another program
public class construct {                              // class Declaration

  // Variable Initialization
  final int NONE=0;
  final int INTEGER=1;
  final int STRING=2;
  int intval;
  String strval;
  int which;

  public construct() {                                 // default construct
    which=NONE;                                        // no value set
  }                                                    // end construct

  public construct(int value) {                        // int construct
    intval=value;
    which=INTEGER;
  }                                                    // end construct(int)

  public construct(String value) {                     // String construct
    strval=value;
```

```
    which=STRING;
  }                                                  // end construct(String)
}                                                    // end class declaration
```

When you use **new** to create a new instance of an object, you can provide arguments, as in:

```
c1 = new construct(10);
```

The Javelin Stamp does not have garbage collection. Once you allocate memory for an object, it remains allocated until you reset the processor. That means you have to be careful allocating objects in response to external events, or timers. A good strategy is to allocate all the objects you will use early in your program and refrain from allocating any more from other points in your program.

Another place to be careful is when Java automatically creates objects on your behalf. For example, consider this:

```
String a = new String("Hello " );
String b = new String("Parallax");
a = a + b;
```

How many objects do you see? Two? The answer is four. There is the object that **a** refers to (it contains "Hello Parallax"). There is also the object that **b** refers to (that string contains Parallax). However, there is also the original string that contains "Hello " – your program no longer refers to it, but it still takes up space in the Javelin Stamp's memory. In fact, the Java interpreter also creates a **StringBuffer** object to perform the actual concatenation, so that's another object for a total of four.

Where are the Pointers?

If you are familiar with C++ or assembler language, you might wonder how the Javelin Stamp handles pointers. A common misconception is that Java doesn't have pointers. This is not really true. In Java, every time you use an object you are using a pointer to the object. That's why you say an object variable is a reference, not the object itself. For example, suppose you want to create a linked list. Each item in the list has a reference to the next element. Program Listing 6.5 shows a simple class that implements the elements. The test **main** method builds a simple list with 4 elements. Notice that the program has only one variable that holds a reference to a list element (**head**).

Program Listing 6.5 - List

```
public class List {

  static List head=null;                             // pointer to first item
  String value;
  List next;

  // create list element (not linked)
  List(String s) {
    value=s;
```

```
    next=null;
  }                                                    // end List

  // insert item in list
  void insert() {
    List ptr, last;
    if (head==null) {
      head=this;
      return;
    }                                                  // end if

    // this code finds the last item in list
    last=head;
    for (ptr=head;ptr!=null;ptr=ptr.next)
    last=ptr;
    last.next=this;
  }                                                    // end insert

  static void printList() {
    List ptr;
    for (ptr=head;ptr!=null;ptr=ptr.next)
      System.out.println(ptr.value);
  }                                                    // end printList

  static public void main() {
    new List("One").insert();
    new List("Two").insert();
    new List("Three").insert();
    new List("Four").insert();
    List.printList();
  }                                                    // end main
}                                                      // end class declaration
```

Every object has a special pseudo reference known as **`this`**. You can use this to refer to the current object. You can see this in Program Listing 6.5. Where the **`List`** object's **`insert`** method sets the **`next`** link.

There are a few more interesting points to Program Listing 6.5. First, notice that **`head`** is static. There is only one head reference no matter how many list items are in use. What's more the **`printList`** method is also static. This is for the same reason – it applies to the list as a whole. The **`for`** statements that scan the list are a good example of using a **`for`** loop in a non-numeric situation. Remember, the first clause initializes the loop (**`ptr=head`**). The second clause tests for the end condition (**`ptr==null`**) and the third clause sets up the next iteration of the loop (**`ptr=ptr.next`**). These clauses are not the usual numeric expressions, but they still work.

In the test **`main`** program, you'll see four **`new`** statements that create objects. They look a bit peculiar because the program doesn't store the object reference anywhere. Instead, it simply calls **`insert`** directly. Since the program no longer needs the objects, there is no need to retain a reference to them. To print the list, the program uses the **`printList`** method.

Arrays

Java also supports array data types. You can create arrays of basic types (like **int**) or you can create arrays that contain object references. All arrays in Java are objects. Create them using syntax similar to an object:

```
int [] x;          // reference to array
x = new int[33];   // create array with 33 elements
```

You can also use an alternate syntax to declare the array reference:

```
int x[];
```

Given the above declaration and *new* statement, you could refer to the first element of the **x** array as **x[0]**. The last element is **x[32]**. You can use these just like any other variable:

```
x[2]=17;
system.out.println(x[2]);
```

Since arrays are really objects, they may have fields. The one you'll find particularly useful is the **length** field. This allows you to determine how many elements the array contains. This is very useful when you want to loop through the entire array with a for loop (see Program Listing 6.6).

Program Listing 6.6 - An Array

```
public class AnAry  {                                  // class declaration
  public static void main() {                          // main declaration
    String [] testary;                                 // Create reference to testary
    String [] testary2 = {"One","Two","Three"};        // Cerate and fill testary2
    testary=new String[5];                             // Create testary with 5 elements
    int i;                                             // Create variable i

    // initialize testary
    for (i=0;i<testary.length;i++)
      testary[i]=String.valueOf(i*2);

    // print both arrays
    System.out.println("testary");
    for (i=0;i<testary.length;i++)
      System.out.println(testary[i]);

    System.out.println("testary2");
    for (i=0;i<testary2.length;i++)
      System.out.println(testary2[i]);

  }                                                    // end main
}                                                      // end class declaration
```

Notice in Program Listing 6.6 that **testary2** uses a set of constants enclosed in brackets as an initializer. This is known as an array constant.

Strings

You usually don't think of strings as relating to microcontrollers, but these days many embedded systems do manipulate strings. You might want to write to an LCD, or receive commands from a PC or to a modem.

Strings are objects, but they are so prevalent in many programs that Java makes a special concession to them. You can create **String** objects using **new** like any other object. You can also assign a string literal to a **String**. For example:

```
String modemprefix = "AT";
```

Like all objects, **String** objects have fields and methods. If you are C programmer, you might think of **String** as similar to an array. However, in Java, strings have very little in common with arrays.

One surprising feature of **String** is that once set, the actual **String** object never changes. That's not to say that the reference can't change, but the actual object stays the same. This can lead to performance problems if you are not careful. For example, suppose you have a method named **getC** that retrieves a character from some source. You might write this code to build a **String** object in the **s** variable:

```
String s = new String();
for (i=0;i<1000;i++) s=s+getC ();
```

This will work, but it is very inefficient. When you compute **s+getC()**, you create another **String** object. Then you set the **String** reference *s* to point to that new object. That means the original string now has no references, and will be lost to the Javelin Stamp. Throughout this loop you'll create and discard 1000 **String** objects! Remember, the Javelin Stamp can't reclaim this memory, so you'll quickly run out of memory.

To prevent this problem, Java also provides a **StringBuffer** object. These objects are similar to **String** objects, but they allow you to manipulate characters in place. Once you are done, you can convert the **StringBuffer** into a proper **String**.

```
StringBuffer sb = new StringBuffer(1000);
String s;
for (i=0;i<1000;i++) sb.append(getC());
s=sb.toString();
```

The String object has several useful methods (see Table 6.5). Most of these are straightforward, although many people have trouble with substring. The substring method has two versions. One takes the starting index and returns the substring from that index to the end of the string. The other version takes a starting index and an ending index. This version returns the string starting at the first index, and ending at the character before the second index. Consider a string that contains "Javelin Stamp". The index arguments start at 0, so if you call substring with arguments of 2 and 4, the call will return "ta", not "tam" as you might expect. You'll find out more details about all of the Javelin Stamp's objects in Chapter 7 and Chapter 8.

Table 6.5: Object Methods

Method	Description
equals	Test objects for equality
hashCode	Returns id number (hash) for this object
toString	Returns a string representation of the object
clone	Duplicates object

Extending Classes

The biggest benefit to object-oriented programming is the ease with which you can reuse code. One thing that makes this possible is inheritance. The idea behind inheritance is that each class extends another class and inherits methods and fields from this base class. Suppose you have a class that represents a temperature probe:

```
public class Probe {
  public Probe(int portnum) { . . . }
  public int getTemp() { . . . }
  public void setOptions(int a) { . . . }
}
```

Later, you update the sensor to include a wind speed indicator. Instead of maintaining two copies of the temperature code, you can create a new class **DeluxSensor** that extends the temperature sensor code. In this way, all the code and fields in the original code are available in the new class. If you make changes to the original code, the new object will inherit the same changes automatically. In this case, the original sensor object is the base class. The new object is said to extend (or derive from) the base class.

```
public class DeluxSensor extends Probe  {
  public DeluxSensor(int portnum) { . . . }
  public int getWindSpeed() { . . . }
  public int getWindDir() { . . . }
  // getTemp and setOptions are inherited from Probe
}
```

It is possible to extend this hierarchy to any number of levels. For example, you might extend **DeluxSensor** into **WeatherStation** that integrates several instruments and an LCD interface. However, unlike some languages, Java only allows you to derive from a single class, it is not possible to derive directly from more than one class.

If you don't specify a base class, your class will extend the default **Object** class. That means that all objects, no matter what their type, will have the basic methods that belong to **Object** (see Table 6.5). Remember, classes that extend other classes (including **Object**) can (and often do) replace methods with custom versions. For example, quite a few classes override **toString** to provide a more meaningful string representation of their contents (the default **toString** doesn't print any of the object's contents). Many objects (like **String**) override **equals** to test the object's contents instead of the actual object.

Usually, you'll want to allow others to extend your classes and inherit members (that is, methods and fields). However, you can control what other classes can access. If you name certain fields or methods **private** they will not be accessible by code in any other class (including classes that extend this class). If you mark members **public**, any code can access them. You can also specify members as **protected**. Classes that extend your class can freely access **protected** members, but other classes have no access. If you don't specify any of these access modifiers (that is, **private**, **public**, or **protected**) then the member is accessible to any code in the same package. You'll read more about packages shortly, but for now consider it as one subdirectory. In addition to making certain members private, you can also mark a class as **final**. This will prevent other classes from extending your class.

Just because a base class provides members doesn't mean the derived class has to use them. You can override methods (or fields) when you want to provide replacements. You can still call the base class version by using the **super** keyword. This can be useful if you want to make a minor modification to an object. For example, suppose your temperature sensor class operates using Fahrenheit temperatures. Later, you decide you want to create a version to do Celsius temperatures. You can simply extend the original class and override the **getTemp** method. Instead of rewriting it totally you can still call the original class method:

```
int getTemp() {
  return 5*(super.getTemp()-32)/9;
}
```

This is a common theme in embedded programming. For example, you might have a base class that represents a serial port. You could extend the class to represent instruments that use the serial port. That way all the serial port code resides in the main class and the other derived classes can share that common code.

An important consequence of using derived classes is polymorphism. Polymorphism is a simple concept for such a fancy word. Suppose you've built the serial port class and extended three other classes from it: **Temp**, **Wind**, and **Humid**. These classes – of course – represent different instruments that all use a serial port for communications. What if you want to keep a list of these items in an array? Since they are all derived from **SerialPort**, you can treat them as if they are **SerialPort** objects. Once you place the objects in the **instruments** array, you can't use members that belong to the derived classes. In other words, calling **instruments[0].getTemp()** is not legal. However, you can access anything that belongs to **SerialPort**. For instance, if **SerialPort** defines an **init** method, you could call it using any (or all) of the elements of the array. If any of the specific objects override the **init** method, Java will call the correct override.

```
public class Instruments {
  public SerialPort[] instruments = new SerialPort[3];
  public Instruments() {
    instruments[0]=new Temp(1);  // on port 1
    instruments[1]=new Wind(2);
    instruments[2]=new Humid(3);
```

```
  }
}
```

This is not true, however, of fields. If the **SerialPort** object defines a field named **port** and **Humid** overrides it, you'll access different fields depending on if you are using a **SerialPort** variable or a **Humid** variable. That's true even if the **SerialPort** variable really refers to a **Humid** object. Remember, variables are just references to objects and it is legal for a base class variable to refer to a derived class object.

If you want to force an object reference into another type of object, you can use a cast, which is simply the name of the object in parentheses. You can only cast an object to a correctly related class. For example, you can cast any object to **Object** since it is a base class of all objects. You can also cast an object back to its original class. However, you can't cast an object to a class that doesn't appear in the object's class hierarchy.

Suppose you have class **B** that extends class **A**. You also have class **C** that doesn't extend any other class (except, of course, **Object** which is the default). Further suppose that you have the following declarations:

```
B b = new A();
B b1;
A a;
C c = new C();
```

The following assignments are legal:

```
a=(A) b;
b1=(B)a;
```

However, the following is not legal:

```
a=(A)c;
```

Because class **A** and class **C** are not related. You also could not make the following assignment:

```
b1=(B)new A();
```

Constructors present a special problem. Each class has to provide its own constructors. Then, if you don't do anything special, Java calls the default constructor for each base class, starting with **Object** (which is the ultimate base class of every object) and working down the class hierarchy until, finally, the most specific constructor executes.

If you think about this, it makes sense. After all, a derived class might need to use methods in the base class that require that the base class' constructor has already executed. However, there are a few cases where this chaining of constructors doesn't work correctly. For example, suppose the base class doesn't have a default constructor? The same situation might arise when the derived class needs to call a non-default constructor.

The answer is to make the first line of the derived constructor a call to **super**. This special keyword calls the base class constructor explicitly.

```
  class BaseClass {
    private int val;
    public BaseClass(int x) { val=x; }
    public int getVal() { return val; }
  }

  class Extender extends BaseClass {
    private int val2;
    public Extender(int a, int b) {
    super(a);
    val2=b;
  }

  public int getAltVal() { return val2; }
}
```

Basic Type Classes

Nearly every data type you can use in Java is an object. Since all objects derive from **Object**, that means you can depend on a certain number of methods being available in all objects. For example, **toString**, a method in **Object**, returns a string representation of any object. Many objects override **toString** so they can return a meaningful representation.

What about the basic types like **int**? Often, it is useful to have a class that represents one of these types. However, you don't want the overhead of using an object just to perform simple operations. Therefore, Java uses simple types for most purposes, but also provides corresponding objects. For example, the **Integer** class wraps an **int** value. This has several benefits. First, you might want to treat a basic type as an object so you can put it in an object array with other objects. Also, these objects act as a central clearinghouse for methods related to the type. Remember, Java has no real global variables or methods – everything has to belong to a class.

Numeric Conversions

You'll often use the wrapper classes to convert strings to the appropriate type. For example, **Integer** has two methods (**parseInt** and **valueOf**) that convert strings to integers. The **parseInt** method returns an **int** whereas the **valueOf** method returns an **Integer** object. You can also specify an optional radix if you want, for example, hex or octal interpretations.

In the opposite direction, you can use **toString** to convert an integer to a string. To convert the basic types, you can use a cast:

```
int n=100;
byte fn = (byte) n;
```

Statics

Numeric conversions are one of the uses of the wrapper classes – Java uses them as containers for what might otherwise be global methods. It does this using static methods. This allows you to refer to a method without having to actually create an instance of an object. Suppose you have an integer variable **x**. You can't call **toString** on an **int** because it isn't an object. You could construct an **Integer** object to contain the **int**, but that's a lot of work just to do a string conversion.

Luckily, **Integer** provides **toString** as a static member, so you can call it like this:

```
String s = Integer.toString(x);
```

You can make methods or fields static. Be aware that a static method can't access any normal fields or methods directly, because there is no object instance associated with the static method. Therefore, there is no **this** reference. That also means, in the case of fields, that there is only one copy of the variable no matter how many object instances exists. That makes static fields useful for creating a kind of global variable. If you make the field **public**, any other part of your program can access the variable (using the class name as a prefix). If you make the field **private** or **protected**, the variable will still be like a global variable, but it won't be accessible from other objects (or unrelated objects in the case of **protected**).

Abstraction

Sometimes, it is useful to write a class that represents an imaginary object that will never exist. For example, suppose you had classes that represented a serial port, a printer port, and an USB port. You'd like to share code between them, but what's the common base class? Printer ports are not serial, nor are they a kind of USB port.

The answer is to make an abstract class that represents ports in general. It doesn't make sense to instantiate this class because there is no such thing as a generic port. Abstract classes can contain reusable code that subclasses can inherit, but they can't be instantiated directly. You must use a derived class.

Program Listing 6.7 - Library Class Example

```
//This program is a Library Class and must be called by another program

abstract class GenericPort {
  protected byte [] buffer;
  protected int buffp;
  protected int bufflen;
  protected int portnum;
  protected int irq;
  public void init();
  public int getData(byte [] data);         // returns bytes read
  public void sendData(byte [] data, int len);
  public GenericPort() { buffer=new byte[256]; buffp=0; bufflen=0;}
  public byte getByte() {
    if (bufflen==0) {
      bufflen=getData(buffer);              // read chunk (assume this never fails)
      buffp=0;
    }                                       // end if
```

```
  return buffer[buffp++];
  }                                          // end getByte
}                                            // end class declaration
```

Exceptions

Java supports a modern idea known as exception handling. Simply put, an exception is a way for your code to signal some event to other parts of your programs. Java uses exceptions frequently in its own library and you may also use them as part of your own programs.

Often, but not always, an exception indicates an error has occurred. Suppose you are writing a general purpose routine that performs a simple calculation based on input parameters. The computation might divide by zero, depending on the input parameters. Of course, you could test for a zero denominator before dividing, but what do you do if you detect this condition? You could print an error message, but that presupposes your program can display a message (remember, I said this routine was general-purpose).

A common solution is to return an error code to the calling method. This is not always good, though. What if the calling program is another general routine? It will have to propagate the error condition somehow. What if the calling program doesn't check for an error condition? You can solve these problems with exceptions.

When an event occurs, like a division by zero, Java throws an exception. Your code can handle the exception by wrapping the code in a **`try`** block see Program Listing 6.8. In this case there isn't much advantage to using exceptions. However, suppose the equation inside the **`try`** block called other methods to do its work.

Program Listing 6.8 - Exceptions Ex1

```
public class Ex1 {
  public static void main() {
    int x=0;
    int y=20;
    int z;
    try {
      z=y/x;
    }                                                  // end try
    catch (Exception e) {
      System.out.println("Divide by zero");
    }                                                  // end catch
  }                                                    // end main
}                                                      // end class declaration
```

Even if code in these other methods divided by zero, the **`catch`** block beneath the **`try`** would be activated (unless, of course, the called methods provided their own **`try`** block. Consider this example.

Program Listing 6.9 - Exceptions Ex2

```
public class Ex2 {
  static int docomp(int a, int b) {
    return a/b;
  }                                                    // end docomp
```

```
  public static void main() {
    int x=0;
    int y=20;
    int z;
    try {
      z=docomp(y,x);
    }                                                    // end try
    catch (Exception e) {
      System.out.println("Divide by zero");
    }                                                    // end catch
  }                                                      // end main
}                                                        // end class declaration
```

This is the real value to exceptions. It allows code that is interested in some event to handle that event, no matter what caused it. Code that doesn't care about an event can simply ignore the event.

Dividing by zero is an example of an unchecked exception. Since it could happen at almost any time, Java does not force you to handle the exception. If you remove the **`try`** and **`catch`** blocks, the code will still compile, but it will cause an abnormal termination of the program.

Many exceptions, however, are checked exceptions. That means that the Java compiler ensures that you handle the exception wherever it may occur. If your code calls a method that may throw an exception, you have to either mark your method as throwing the same exception, or you must handle it yourself. You indicate which checked exceptions your method may throw by using a **`throws`** clause.

You can find an example in Program Listing 6.10. Here, there is a custom exception (**`ScaleError`**) that extends **`Exception`**. When the calculation detects a zero divisor, it throws the custom exception, which can be caught by any of the interested caller. Of course, the **`docalc`** method could catch the divide by zero exception and simply convert it to the special exception by throwing it in the **`catch`** clause.

Program Listing 6.10 - Scale Error (Extends Exception)

```
class ScaleError extends Exception {
  // no methods or fields required
}

public class Ex {
  static int docalc(int a, int b) throws ScaleError {
    if (b==0) throw new ScaleError();
      return a/b;
  }

  public static void main() {
    int x=0;
    int y=20;
    int z;
    try {
      z=docalc(y,x);
    }
    catch (ScaleError e) {
```

```
      System.out.println("Scale Error");
    }
    catch (Exception e) {
      System.out.println("Unknown exception");
    }
  }
}
```

Notice that there are multiple **catch** clauses. The first one is the most specific type of exception. The last one catches any **Exception** object including objects that derive from **Exception**. That's why that clause must come last. If it were first, it would match the **ScaleError** exception and the second **catch** clause would never execute. Try removing the **try** and **catch** block and rebuilding the program. You'll find that the compiler rejects the program because it sees that there is an unchecked exception. Of course, you could mark **main** so that it throws a **ScaleError** exception. Then the exception would terminate the program like an unchecked exception.

Packages and CLASSPATH

When Java must locate a class file, it searches the directories listed in the CLASSPATH environment variable. This is a list of directories separated by semicolons.

Even with multiple directories, you'd quickly clutter each directory with class files. For that reason, Java supports packages. Packages are somewhat like subdirectories that contain class files. For example, suppose your CLASSPATH variable contains a single directory named `C:\Classes`. When you attempt to load an ordinary class, the IDE will search in the `C:\Classes` directory.

However, some classes belong to a package, a group of related classes. For example, you might want to refer to a **Cache** object. That object is in the **stamp.util** package, so to declare it, you could write:

```
stamp.util.Cache  = new stamp.util.Cache();
```

The JVM would look for the **Cache.class** file in a subdirectory of one of the CLASSPATH directories. In this case, there is only one directory (`C:\Classes`) so the class file should be in `C:\Classes\javelin stamp\util\Cache.class`. Of course, if there were more directories listed in the CLASSPATH variable, the IDE would also search those directories, always looking in the `javelin stamp\util` subdirectory.

It wouldn't be very convenient to have to write **stamp.util.Cache** every time you wanted to use it. By default, if you use a class name, it can only reside in one of the top-level CLASSPATH directories or in the special package **java.lang**. However, you can use the **import** statement to mark certain packages that you want to behave as though they were local.

If you wanted to use the name **Cache** instead of **stamp.util.Cache**, you can add the following line at the start of your java source file:

```
import stamp.util.Cache;
```

You can also get all the classes in **stamp.util** by writing:

```
import stamp.util.*;
```

Keep in mind that you never have to use **import**. If you prefer, you can simply use fully qualified class names everywhere. Still, using **import** makes your programs much more readable so you'll want to use it where appropriate. A common mistake beginning Java programmers make is to try something like this:

```
import System.out.println;
println("Hello World");
```

This won't work! That's because **System** is an object (part of the **java.lang** package), but **out** is a static field of this object. This field is an object reference that has a method called **println**. The **import** statement only works with classes. You can't import a field or method.

Summary

You could read an entire book on Java – there are plenty around. However, this chapter, along with the examples in the next few chapters, will give you a lot of practice with Java. You can also find many online tutorials, books, and documentation on Java. Be sure to check out the online resources section for more information. Be aware, though, that many books and other materials will focus on writing graphical programs, not embedded systems.

This chapter may leave you wondering why use Java. In the next chapter, however, you'll see that Java's networking capability is a real winner. And Java's cross platform ability will serve you well in a networked environment.

Online Resources

http://java.sun.com
: Java's home on the Web. Free downloads of the JDK, tutorials, news, and more.

http://www.norvig.com/java-iaq.html
: Java Infrequently Asked Question (IAQ) list.

http://mindprod.com/jgloss/gotchas.html
: Java gotchas

http://uranus.it.swin.edu.au/~jn/java/style.htm
: Automatically format your Java code

Javelin Stamp Keyword Reference

abstract

The **abstract** keyword has two possible methods. You can mark a method as **abstract** to indicate that the class contains no code for the method. That implies that you can't instantiate the class, only extend it. Classes that extend the class must either implement the **abstract** method, or also be an abstract class.

You can also mark an entire class as **abstract** – any class that contains at least one abstract method is an abstract class.

Examples:

```
abstract class AbaseClass {
  abstract void someMethod();
}
```

boolean

The **boolean** data type can contain the values **true** or **false**.

Example:

```
boolean limit = false;
```

break

When you are executing a loop (that is, a **for**, a **do**, or a **while** loop), you may find it useful to exit the loop prematurely. That's the purpose of the **break** statement. You can optionally provide a label that will cause the **break** statement to exit to an outer level of nested loops.

You can also use a **break** statement to stop execution inside a **switch** statement. This is useful to end a block of code that handles one condition (see **switch** for more details).

Example:

```
outside:    // label
for (x=0;x<10;x++) {
  for (y=0;y<10;y++) {
    f(x,y);  // do something with x and y
      if (CPU.readPin(CPU.pin2)) break;           // skip to next X early
      if (CPU.readPin(CPU.pin3)) break outside;   // stop both loops
    }
  }
```

See Also: ***continue, do, for, switch, while***

byte

You can use the **byte** type to store any 8-bit quantity. The **byte** type is signed, so it can store values from –128 to 127. The signed nature of bytes can lead to common compile errors. For example, you can't assign **0xFF** (255) into a byte, because it is out of range and the compiler won't allow it. You can cast the value, however (see the examples below). If you really want an unsigned byte, consider using a **char**.

Examples:

```
byte x = 10;
byte y = 0x55;
byte z = (byte)0xFF;
```

You can't directly assign 0xFF (255) into a *byte*, because it is out of range and the compiler won't allow it. However, you can assign numbers greater than 127 into a byte with a cast:

```
byte fullvalue = (byte) 0xFF;
```

See Also: ***char***

case

See ***switch***

catch

See ***try***

char

The **char** data type stores a single byte character. This type can hold a single ASCII character, or you can use it as an unsigned alternative to **byte**.

Example:

```
char c = '@';
```

See Also: ***byte***

class

Classes are templates that create objects. You'll introduce each class definition with the **class** keyword.

See Also: ***extends, private, protected, throws***

continue

When you are executing a loop (that is, a **for**, a **do**, or a **while** loop), you may find it useful to jump directly to the next iteration of the loop prematurely. That's the purpose of the **continue** statement. You can optionally provide a label that will cause the **continue** statement to exit to an outer level of nested loops.

Example:

```
outside:                                // label
for (x=0;x<10;x++) {
  for (y=0;y<10;y++)   {
    if (CPU.readPin(CPU.pin2)) continue;           // skip this value of Y
      if (CPU.readPin(CPU.pin3)) continue outside; // skip to next X
      f(x,y);                           // do something with x and y
    }//end if
  }//end for y
}//end for x
```

See Also: ***break, do, for, switch, while***

default

See ***switch***

do

Use the **do** loop construct to perform a statement (or statements) a repeated number of times. The **do** construct always executes the loop once before performing the end of loop test.

Example:

```
do {
  CPU.writePin(CPU.pin8,getNext());
} while (CPU.readPin(CPU.pin5));  // continue until pin5 goes low
```

See Also: ***break, continue, for, switch, while***

else

See ***if***

extends

When you define a class, by default, it extends the **Object** class. However, you can make it extend any class you like by specifying **extends**. When an object extends a base class, it can override the base class methods and fields. It can also access the base class **protected** members.

Example:

```
class ParallaxDemo extends GenericDemo {
  . . .
}
```

See Also: ***class, throws***

final

When you declare something **final** you indicate that it can't be changed. For example, declaring a variable **final** makes it a constant. You can also declare a class as **final** to prevent others from extending it.

Example:

```
final int x = 33;

final class TheEnd {
  . . .
}
```

finally

See ***try***

for

The **for** statement allows you to execute a group of statements multiple times. Each **for** statement has three parts separated by semicolons. The first part initializes the loop, the second part tests for the end of the loop, and the third portion specifies code to execute after each loop completes.

There are many variations on the **for** loop. Consider this example:

```
int i;
for (i=0;i<10;i++) System.out.println(i);
```

This initializes the **i** variable to 0 and then executes the loop until **i** is less than 10. At the end of each loop, the **for** statement adds one to **i** (**i++**). In this case, only one statement is part of the loop (**System.out.println(i)**), but often you'll see multiple statements enclosed in braces.

Notice that if the test (the second part of the loop) fails right away, the code never executes. For example:

```
int k;
for (k=0;k>0;k++) System.out.println(k);
```

This loop never executes because **k** is not greater than **0**.

As a special case, the **for** statement allows you to declare the loop variable right in the statement:

```
for (int j=0;j<100;j+=2)
```

In this example, the loop variable (**j**) increases by two on each pass through the loop.

You can omit any (or all) of the portions of the for loop:

```
for (;;j++) { . . .  }
for (j=0;j<10;) { . . . }
for (;;;) { . . . }
```

The first example assumes **j** is already set, and will continue forever (presumably the loop will contain a **break** statement). However, at the end of each loop, **j**'s value increases by one. The second example doesn't change the value of **j** at all. In this case, some code within the loop would probably assign a value to **j**. The final example loops forever (unless something inside the loop uses the **break** statement). This is useful when you want a loop to run without stopping (although you could also use **while(true)** to get this same effect.

The **for** statement is very versatile. You don't have to directly refer to the loop variable in any of the clauses. For example, suppose you want to call a method **f** on each element of an array until you find an array element that contains a –1. You could write:

```
for (j=0;ary[j]!=-1;j++) f(ary[j]);
```

Here's another example that loops until the input on pin 0 is high (and counts the number of seconds it is low):

```
for (ct=0;CPU.readPin(CPU.pin0);ct++) CPU.delay(10000);
```

Notice that the test does not involve the loop variable at all.

Sometimes it is useful to use more than one loop variable. Here is an example that declares two variables (**x** and **y**), initializes them, and changes them on each loop:

```
for (int x=0,y=0;x<10;x++,y+=2) System.out.println(x+y);
```

Notice the commas separate the different portions of the **for** loop. In C this is known as a comma operator and you can use it anywhere. However, in Java there is no general-purpose comma operator – you can only use this syntax in a **for** loop.

Examples:

```
for (n=0;n<ary.length;n++) f(ary[n]);
for (x=0;x<10;x++) f(x);
```

See Also: ***break, continue, do, while***

if

The **if** statement allows you to conditionally execute a statement (or a group of statements surrounded by braces). The **if** statement requires an expression that returns a boolean in parentheses. If this expression evaluates to **true,** the following statement executes. If it is **false**, execution continues with the next statement.

In addition, you can specify an optional **else** clause. The statement (or statements) following the **else** will only execute if the **if** condition is **false**. Often it is useful to use multiple **if**/**else** statements. For example, consider this code:

```
if (x==10) System.out.println("Condition A");
if (x<20)
  System.out.println("Condition B");
else
  System.out.println("Condition C");
```

This code will output both "Condition A" and "Condition B" if **x** is 10. You probably meant to write:

```
if (x==10) System.out.println("Condition A");
else if (x<20) System.out.println("Condition B");
else System.out.println("Condition C");
```

This prints one line, depending on the value of **x**.

Examples:

```
if (x==10 && y<0) break;

if (CPU.readPin(CPU.pin1)) func(100);
```

See Also: ***switch***

import

The **import** statement is a directive to the compiler that tells it to search for class names in different packages. When you name a class, by default the compiler looks in your current package (essentially, the current directory) and in the default **java.lang** package. If you want something from another package, you must fully qualify the name. For example, you might write **java.util.Random** to access random numbers.

This is cumbersome if you need to access the object frequently. You can place any number of **import** statements at the beginning of your file. Each import names a class you plan to use in your code. Then you can refer to the class using an ordinary name. In addition to naming specific classes, you can also use an asterisk to refer to the entire package. So importing **java.util.Random** allows you to use the **Random** class, but importing **java.util.*** imports all classes in the **java.util** package.

Examples:

```
import java.util.Random;
import java.util.*;
```

See Also: ***package***

int

The **int** data type defines 16-bit signed integers. Integers can hold between –32768 and 32767. Notice that the limit of 32767 precludes directly writing hex constants greater than 0x7FFF. If you need a number greater than 0x7FFF you'll either need to compute the equivalent negative (two's compliment) number or break the number into parts. Moreover, if you break the number into parts, you'll have to keep the compiler from realizing the expression is constant or else it will resolve it at compile time.

As an example of this, suppose you want to pass 0x80A0 to a method named **f**. You might try this:

```
f(0x80A0);
```

However, this won't work because the compiler decides it is too large to be an integer constant. Next, you might try:

```
f(0x80<<8+0xA0);
```

That's the right idea, but the compiler realizes it can calculate 0x80A0 at compile time and you wind up with the same problem. One possible way around this is to define a method to prevent the compiler from resolving the expression. So you might write this method:

```
int bytes2hex(int hi, int lo) {
  return hi<<8+lo;
}
```

Now the call to method **f** is simply:

```
f(bytes2hex(0x80,0xA0));
```

Another solution is to subtract one from the number and invert it. This will give you the magnitude of the equivalent negative number. So 0x80A0 minus 1 is 0x809F. Inverting this results in 0x7F60 (32608 decimal) so -32608 is the same as 0x80A0 and this code will correctly compile:

```
f(-32608);          // 0x80A0
```

Examples:

```
int x;
int num = 100;
```

new

When you declare an object variable, it is simply a reference to an object. Using **new** creates an object that you can assign to an object reference. Following **new**, you'll specify the class name followed by any constructor arguments required (in parentheses). Of course, the class in question must have a visible constructor that matches the arguments. Even if you don't want to supply any arguments (that is, you want to use the default constructor) you'll still need to provide an empty set of parentheses.

Usually, the variable on the left side will have the same type as the argument to **new** as in:

```
SomeObj anObj = new SomeObj();
```

However, it is possible to assign the object to a variable of a base class of the object. Since **Object** is a base class of all objects, for example, you might write:

```
Object anObj = new SomeObj();
```

The object is still a **SomeObj**, but your program will treat it as an **Object** until you cast it to the more specific type.

Examples:

```
pid = new PIDController(10,1,"Unit 1");

LED led = new LED(CPU.pin3);
```

null

Uninitialized object references have the **null** value. You can also assign **null** to an object reference to mark the value as empty once you are done with the object the variable refers to.

Example:

```
if (anObj != null) anObj.doSomething();
```

package

You can organize your classes into packages. By placing code in a package, you not only group similar classes together, but you also avoid the risk of naming a class something that is already in use by other code (that is presumably in another package). In addition, you can specify fields and methods that are only accessible by other code in the package. This is similar to having private data or methods, but any class in your package can access the members.

Each class that belongs to a package must include a **package** statement. This tells the compiler that the class is a member of the package and makes it implicitly search the package for any unresolved class names. The **package** statement must appear before any class declarations in the file (it is usually the first non-comment line in the file). You may only use one **package** statement per file.

Package names may be hierarchical. For example, you might make a package named **robot.wheels** and another named **robot.sensors**. If you expect to widely distribute your packages to the public, you should consider following a widely-used convention to avoid conflict. The idea is to use your Internet domain name (assuming you have one, of course) but with the top-level domain name first (and in upper case). So, a fictitious package from Parallax, might be: *COM.parallax.fictitious*.

Class files that belong to the **package** must reside in a subdirectory that matches the package name (replacing dots with slashes). So the above example would be the **fictitious** subdirectory of the *parallax* directory, which would be in the *COM* directory. The *COM* directory would be a subdirectory of one of the directories in the CLASSPATH environment variable.

You should note that if you are simply writing programs, you don't need to use **package** at all. This statement is for either organizing very large programs or distributing code for others to use. Small programs that only you will use do not really need **package** although you can use it if you like.

Example:

```
package COM.parallax.fictitious;
```

See Also: ***import***

private, protected, public

You can use the **private** keyword to mark fields and methods. A member that is private can't be used except in the class that defines it. Marking a member public has the opposite effect. A **public** member is accessible by all code. A member that uses **protected** is visible only to code in the class that defines the member and any classes that extend that class.

If you don't use any of the three keywords (**private**, **protected**, or **public**) the member has package visibility. That means that the member is public to any class in the same package, but private to all other code.

You can also mark classes as **public**. Any class that is not public will have package access.

Example:

```
public class A {
  int pack_var;            // package visible
  public int pub_var;      // public variable
  protected int prot_var;  // protected variable
  private int keep_out;    // private variable

  // all of the following is OK
  void test() {
    pack_var=0;
    pub_var=0;
    prot_var=0;
    keep_out=0;
  }

  private class B {
    void testB() {
      A aobj = new A();
      aobj.pub_var=10;     // Ok
      aobj.keep_out=3;     // error
      aobj.prot_var=9;     // error
      aobj.pack_var=5;     // OK (same package);
    }
  }

  private class C extends A {
    void testC() {
      prot_var=77;         // ok - C can access A variables directly
      keep_out=33;         // error!
    }
  }
```

return

When you call a method that returns a value (that is, it is not a *void* method) you'll need to return a value. This is the purpose of the **return** statement. It returns control to the calling part of your program and specifies the return value of the method (which the calling program may discard, of course). The expression you use with **return** must match the method's return type. If the method returns **void**, you may use the **return** statement alone to end the method early. Otherwise, a **void** method will return automatically when it encounters the final closing brace.

Examples:

```
void a(int n) {
  if (n==0) return;
  f(n);
}   // automatic return

int b() {
  System.out.println("Processing B");
  return 0;
}
```

See Also: ***void***

short

The **short** data type is the same as an **int**.

Example:

```
short value=33;
```

See Also: ***int***

static

You may declare fields and methods **static**. This means that, unlike other members, they apply to the class as a whole. You don't need to instantiate the object to use a **static** member. Also, a **static** method can't access non-static members of the class directly.

A **static** member is useful for cases where you would normally use a global variable or method. For example, suppose you have a **CommLink** class that handles communications with the outside world. You might have several **CommLink** objects, each representing a different port. However, you want to track the total number of errors for all ports.

You might write:

```
class CommLink {
  private static int errct = 0;
  public static void reportError() { errct++; }
  public static int errorCount() { return errct; }
  private int portNumber;
    .
    .
    .
}
```

Notice that the two **static** methods can access **errct** only because it too is **static**. Any attempt by these two methods to access, for example, the **portNumber** field would cause a compile-time error.

From outside the object, your code could call **CommLink.errorCount** to fetch the error value. There is no need for the program to actually create an instance of **CommLink** using **new** first. One common case where this is useful is in the class' **main** method. It is static because when the program starts, there is no instance of the class. Since **main** is **static**, that doesn't present a problem.

Another common use for **static** fields is to provide named constants for use elsewhere in your program. For example, you might have an entire class consisting of **static** constants:

```
public class Bitmasks {
  final public static int bit0=1;
  final public static int bit1=2;
  final public static int bit2=4;
     .
     .
     .
  final public static int bit7=128;
}
```

Then your program could refer to **Bitmasks.bit7** to retrieve the value 128.

Examples:

```
public static void main(String args[]) {
  . . .
}

static int errct;
```

super

You can use the **super** keyword to call the base class constructor of a class from within a class constructor. If you don't supply the **super** keyword as the first statement of the constructor, Java will call the base class default constructor. This could be a problem if the base class does not have a default constructor, or if you need to pass the base class constructor arguments.

You can also use **super** in any non-static method. In this case, **super** acts like the **this** reference, except that it acts as a reference to the base class. This is useful if you want to call a base class method or access a base class field that you have hidden in the derived class.

Example:

```
class Base {
  int z=10;
}
```

```
class Other extends Base {
  int z=100;
  void test() {
    System.out.println(z);             // prints 100
    System.out.println(super.z);       // prints 10

    // another way to do this
    Base otherBase = (Base)this;
    System.out.println(otherBase.z);  // prints 10
  }
}
```

See Also: ***this***

switch

You'll use the **`switch`** statement to compare a value to a group of constants and execute code based on the value. Within the **`switch`** statement you can place any number of **`case`** statements. When the **`case`** statement matches the test value, execution begins at that point. It continues until the code reaches a **`break`**, **`return`**, or **`throw`**. Notice that execution does **not** stop when reaching another **`case`** statement. You may also specify a **`default`** clause that will match any value.

Example:

```
switch (n) {
  case 1:
    System.out.println("One");
    break;
  case 2:
  case 3:
    System.out.println("Two or Three");
  case 4:
    System.out.println("Two, Three, or Four");
    break;
  default:
    System.out.println("Huh?");
    break;
}
```

See Also: ***break, case, default, if***

this

Every non-static member method has access to a pseudo-variable named **`this`**. The **`this`** variable is simply a reference to the current object. This can be useful if you want to pass a reference to another method, for example. You can also use it to access an object field if it is hidden by a local variable or formal parameter. For example, it is not uncommon to see an object with the following constructor code:

```
class ConstDemo {
  int x;
  ConstDemo(int x) { this.x = x; }
}
```

Example:

```
Object [] objs = new Object[10];
objs[0]=this;
```

See Also: ***super***

throw, throws

The **throw** statement allows you to create an exception. An exception consists of an object that extends **Throwable**. In general, nearly all exceptions extend subclasses of **Throwable**. In particular, exceptions derive from **RuntimeException** or **Error** (for an unchecked exception) or **Exception** (for a checked exception). A method that may throw a checked exception must use the **throws** keyword in the method declaration.

When you call a method that may throw a checked exception, your code must either catch the exception (using **try**) or use the **throws** statement to indicate that your code may also throw the same exception. Unchecked exceptions do not have to be caught, but if one occurs, your program's execution will terminate.

Example:

```
class EmptyArgumentException extends Exception {
  EmptyArgumentException() {
    super("Argument must not be empty");
  }
}

public class SomeClass {
  public void aMethod(String s) throws EmptyArgumentException {
  if (s==null || s.equals("")) throw new EmptyArgumentException();
      .
      .
      .
   }
}
```

See Also: ***try***

try

When you perform an operation, there is always a chance it might throw an exception. Unchecked exceptions (like dividing by zero, for example) can happen at any time, and the Java compiler does not require you to catch them. However, many exceptions are checked – the compiler requires you to either catch the exception, or mark that you may throw the same exception (using the **throws** keyword).

To catch an exception, enclose the code that might cause the exception in a **try** block. If the code executes without any exceptions, nothing special happens. However, if an exception occurs, the compiler scans the adjoining **catch** statements, looking for a matching type. You can catch broad categories of exceptions by writing **catch** statements for a base class (like **Exception**, which will catch all checked exceptions). You can have any number of **catch** statements as long as each **catch** handles a different type. You should place specific **catch** statements before more generic ones.

If there is no appropriate **catch** statement, the exception propagates to the calling method. If it is executing within a **try** block, the search continues. If there is no match, or no **try** block, the exception propagates to the next caller, continuing until a **catch** is found, or there is nowhere else to search (at which point, the program terminates).

You can also place a **finally** block after the **catch** statements. The code in the **finally** block will execute whenever execution leaves the **try** block. That means the **finally** code will execute if no exceptions occur, or if an exception occurs (even if it is not caught), or even if the code within the **try** block executes a **return**.

Example:

```
class BadArgumentException extends Exception {
  BadArgumentException() { super("Bad Argument"); }
}

public class TryTest {
  void test() {
    try {
      test1();
    }
    catch (BadArgumentException e) {
      System.out.println(e);
    }
    finally {
      System.out.println("Done!");
    }
  }

  void test1() throws BadArgumentException {
    test2(-1);  // try changing this value
  }
```

```
  void test2(int n) throws BadArgumentException {
    if (n==-1) throw new BadArgumentException();
      System.out.println(n);
  }

  public static void main(String[] args) {
    TryTest t=new TryTest();
    t.test();
  }
}
```

See Also: ***throw, throws***

void

The **void** type is used for methods that return no value.

Example:

```
void f(int n) {
  .
  .
  .
}
```

See Also: ***return***

while

Use the **while** loop construct to perform a statement (or statements) a repeated number of times. The **while** construct always tests for the end of the loop before it executes the loop. Therefore, it is possible that the loop will never execute.

Frequently, you'll write a **while** loop with no statements simply to wait for some condition. For example: **while (CPU.readPin(CPU.pin5);** will wait for pin 5 to go high.

Example:

```
while (CPU.readPin(CPU.pin5)) {           // continue until pin5 goes low
  CPU.writePin(CPU.pin8,getNext());
}
```

See Also: ***break, continue, do, for, switch***

Javelin Stamp Operator Reference

[]
The bracket operators define or use an array. Javelin Stamp arrays can only be one-dimensional and always start at index 0.

Examples:

```
// The next two lines are the same; you can use either syntax
int [] x = new int[10];
int y[] = new int[10];
x[2]=0;
y[1]=x[2];
```

++, --
The **++** and **–** operators perform slightly different methods depending on their position. Consider **++** first. It always adds 1 to the variable it is next to (known as an increment operation). However, it also returns a value used in the expression. If the **++** precedes the variable, the increment occurs before the value is taken. If the **++** is after the variable, the increment occurs after taking the value. The **--** operator works the same way but it decrements (subtracts 1) instead of incrementing.

Examples:

```
int x=5, y;
y = ++x;      // y=6, x=6
y = x++/3;    // y=2 (6/3), x=7
y = --x*2;    // y=12 (6*2), x=6
```

(type)
You can force a value of one type into another type using the cast syntax (**type**). Some casts don't make sense, and the compiler won't allow them. For example, you can't convert an object type (including **String)** into, say, an integer. You also can't cast a type to another unrelated type.

Often, the compiler will automatically cast values where it can be sure it is safe. For example, while you can cast a **short** to an **integer**, you don't have to, because the compiler knows it can always fit a **short** into an **integer**. On the other hand, you do have to explicitly cast an **integer** into a **short** because it is possible that the integer will be too big, and your program will use the wrong value after the cast. The cast is the compiler's way of making sure you really want to do the conversion. The same holds true for objects. You don't need to make an explicit cast to convert an object to one of its base classes. However, you do need a cast to convert an object to a more specific type. Consider this example:

```
Object o = new SomeObject();      // no cast required, because Object must
                                  // be a base class
SomeObject so = (SomeObject)o;    // cast required here
```

Examples:

```
short s=20;
int n;
n=(int)s;  // cast not required here
n=n*3+7;
s=(short)n; // cast required
```

+, -, *, /, %, ()

These operators represent the usual math operators: addition (+), subtraction (-), multiplication (*), division (/), and remainder from integer division (%). Parentheses can override precedence.

Java evaluates these operators using the normal order of operation (multiplication and division first, followed by addition and subtraction). So 4+3*2 is equal to 10, not 14. You can override the order using parentheses, so (4+3)*2 is 14.

Don't forget that division is integer-only on the Javelin Stamp. So 10/3 is 3 (and 10%3 is 1, the remainder). You may want to rearrange your computations to make sure division occurs in such a way that it doesn't affect your results. For example, suppose you read a value from an A/D converter. To get the correct answer in volts, you need to multiply by 5/256. You don't want to write this so that 5/256 is computed first since that result will always be zero. So don't write:

```
y = 5/256*x;
```

Instead, you want to write:

```
y = (5*x)/256;  // parentheses not necessary, but added for clarity
```

Even writing it this way, any value below 52 will result in a 0 result. You might prefer to compute decivolts (1/10 of a volt units) instead by scaling everything up by 10. For example:

```
y = (50*x)/256;
```

If you need to find the volts, you can use the / operator. The % operator could determine the fractional (1/10) volt units. For example:

```
System.out.println("Volts = " + y/10 + "." + y%10);
```

Examples:

```
y = 10 + 33 / 17 % 3 * 100;  // answer is 110
```

<<, >>, >>>

These operators all shift their left argument to the left (**<<**) or right (**>>** and **>>>**) the number of times specified by their right argument. Shifting to the left is equivalent to multiplying by powers of two, and shifting right is the same as dividing by a power of 2. So writing **100>>4** is the same as writing **100/16** (because 2 to the 4th power is 16). In addition, shifting is typically faster than multiplication and division.

It is possible to rewrite certain common multiplication statements as sums of shifts to realize faster execution. For example, when working with decimal numbers, you'll often need to multiply by 10. Observing that 10 is actually 8+2, you can rewrite **10*x as (x<<3)+(x<<1)**.

The **<<** operator always sets the least-significant bit of the result to zero. The **>>** operator preserves the most significant bit (which represents the sign). This makes positive numbers stay positive and negative numbers stay negative. If you really want to zero fill the most significant bit, use **>>>** which is a true unsigned shift.

Examples:

```
x = 10<<3;                // x = 80
```

<, >, <=, >=, ==, !=

The relational operators allow you to test two values and get a boolean value (**true** or **false**). Each operator makes a particular test:

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

You have to be careful when using **==** and **!=** with objects. For objects, these operators only test that the object references are the same. In many cases, you may want to treat objects that are not the same as though they are equal. For example, suppose you have a **String** object that contains the word "END" and another **String** object that you read from an RS-232 device. Suppose the RS-232 string also contains "END" and you compare them. Since the objects are not the same identical object, they are not equal (as far as == is concerned). Instead, use the **equals** method (part of **Object**) to make the test. The default version of **equals** is no different than **==**, but many classes (including **String** provide different versions of **equals** that behave the way you would expect).

Examples:

```
if (x==10) done();
while (y!=33) {
  perform(y);
}
```

&, |, ^
These operators perform logical operations on binary numbers stored in **int**, **short**, or other integral types.

To understand how these operators work, consider their arguments as binary numbers. For example, 100 decimal is 01100100 binary and 7 decimal is 00000111 binary (assuming **byte** data types). If you and these numbers together, the result will only have a 1 where both arguments have a 1. So the result would be 00000100 (or 4 decimal). An OR operation will have a 1 where either or both arguments have a 1. So using or on these two numbers will result in 01100111 (or 103 decimal). Exclusive or results in a 1 where there is a one in either argument, but not both. So the result for exclusive or would be 01100011 (or 99 decimal).

Examples:

```
int x=100, y=7, z;
z = x & y;
```

&&, ||
These operators are superficially similar to the **&** and | operators. However, while **&** and | operate on integers, **&&** and | | operate on boolean values. This is especially useful in **if**, **do**, and **while** statements.

Examples:

```
if (x==3 && y!=5) doit();

boolean b = x==100;
boolean c = y!=55;

while (b && c) go();
```

~, !
These operators perform the invert method. The **~** operator inverts integers bit by bit. So a **byte** with a value of 100 decimal (which is 01100100 binary) will invert to 10011011 binary. The **!** operator is strictly for boolean data. So it turns **true** into **false** and vice versa.

Example:

```
if (!CPU.readPin(CPU.pin3)) break;
```

?:

The conditional operator, unlike other operators, requires 3 arguments. The first evaluates to a boolean. If this value is **true**, the operator evaluates its second argument. Otherwise, it evaluates the third argument. In other words **q = x==10?-1:2*x;** will set **q** to –1 if **x** is 10. Otherwise, **q** will be set to **2*x**.

It is important to realize that only one of the last two arguments will execute. That means that any side effects (like **++** or **--**) will not occur in the unused argument. For example:

```
int x=10, z=5, q;
q = x == 10?-1:++z;
```

This code will set **q** to –1 and not change **z** at all. On the other hand, this code would change **z**:

```
int x=10, z=5, q;
q = x == 10? ++z: -1;
```

Or:

```
int x=10,z=5,q;
q = x!=10?-1:++z;
```

Examples:

```
x = y!=3?5:10;
=, +=, -=, *=, /=, %=, >>=, <<=, >>>=, &=, ^=, |=
```

The **=** operator, of course, assigns a value into a variable (as in **x=10;**). It is not the equality operator (which is **==**). The other related operators all perform the indicated operation on their left-hand argument and their right-hand argument while storing the result back in the left hand argument. That is to say, **x+=5;** is the same as **x=x+5;** for practical purposes.

Examples:

```
x *=10;
```

instanceof

The **instanceof** operator returns **true** if its first argument is an instance of the class named in the second argument. An object is considered an instance of a class even if the object uses the class as a base class. So, for example, all objects are instances of **Object** (although *null* is not an instance of **Object**).

You can use **instanceof** to ensure safe casting. For example, consider this code:

```
class bar {
  . . .
}
```

```
class foo extends bar {
  . . .
  void f(Object o) {
    if (o instanceof foo) {
      foo fooobj = (foo) o;  // will definitely work
        . . .
    }
    if (o instanceof bar) {           // true even for objects of type foo
      bar barobj = (bar) o;           // will definitely work
        . . .
    }
```

Examples:

```
if (obj instanceof Error) procErrorObject(obj);
```

Unused Keywords

The IDE compiler currently recognizes a group of Java reserved words (keywords) that are unsupported in the Javelin Stamp. Some of these words are reserved for historical reasons and are not currently used in regular Java or the Javelin Stamp. You should not use these keywords in your programs. While the compiler will accept their use, the Javelin Virtual Machine will fail to recognize them.

Unsupported Reserved Words:

`const`, **`double`**, **`float`**, **`goto`**, **`implements`**, **`interface`**, **`long`**, **`native`**, **`synchronized`**, **`transient`**, **`volatile`**

7: Working With Objects

Objects are a key part of the Java programming language. You can benefit right away from using other people's objects. For example, the Javelin Stamp has quite a few objects that act as UARTs or interact with peripheral devices. You don't need to know anything about how they work internally. That's the power of objects, you just use them. However, you'll get the most out of objects when you understand how to create new ones yourself and reuse them later or share them with others.

You can hardly use Java without some understanding of objects. In the last chapter, you read a little about objects. In this chapter, you'll dive a little deeper into the objects oriented system that makes the Javelin Stamp so flexible and powerful.

What's an Object?

Let's start right at the beginning. What's an object? In simple terms, an **object** is an entity that has "state", meaning that it can store information across sessions. The information isn't lost or reset each time your program code refers to the object.

An object can perform operations on itself, via methods. Consider this example. In a non-object programming language (like PBASIC for the Basic Stamp) you have functions that read and write serial data (**SEROUT** and **SERIN**). These functions are related, but only because you know they are related. PBASIC recognizes no special relationship between them. Setting the baud rate for **SERIN**, for example, doesn't affect calls to **SEROUT**. In fact, since the calls have no state, it doesn't even affect subsequent calls to **SERIN**. You might write:

```
baudrate var word
serialpin var nib
baudrate = 84
serialpin=16
serout serialpin, baudrate, …
```

Your program, in this case is keeping track of the serial port state (the baud rate and pin number to use). With the Javelin Stamp, you'd perform the same operation using a **Uart** object from the **stamp.core** package. This object handles the methods you'd expect using **SERIN** and **SEROUT**, but it also remembers the state of the serial port. Once you create the object, it even listens for serial input while your program is doing other things.

In this case, the **Uart** object contains all the methods and fields required to do serial I/O. When you want to do serial I/O, you know everything you need is within the **Uart** object. When you construct a **Uart** you provide all the initial conditions:

```
Uart xmit = new Uart(Uart.dirTransmit, CPU.Pin0,Uart.invert,
  Uart.speed9600, Uart.stop1);
```

The object then remembers the parameters. To write to the port, you'd simply call:

```
xmit.sendByte(v1);
```

7: Working With Objects

In traditional object-oriented parlance, you send messages to an object to tell it what to do. With Java, sending a message means calling a member method. So you might say, the above example sent the **`sendByte`** message to the **`xmit`** object.

One advantage of objects is that your access to them is strictly through **`public`** members. Therefore, you might replace the **`Uart`** object with, for example, a **`Uart422`** object that you wrote to handle RS422 communications. As long as the new object supports all the same public members as **`Uart`**, you'd only have to change the call to create the object.

Your program creates objects based on a class that you define. Think of a class as a cookie cutter. You don't eat the cookie cutter, but it makes the cookies. A class is just a blueprint for an object. When you instantiate the object (for example, by using the new keyword in Java), you create a particular object that follows the plan laid out by the class.

The real trick to object-oriented design is deciding what constitutes an object. You could write your entire program in one object, but that probably isn't the best idea. A good object is a focused representation of some entity in your design. Good objects have a well-defined and limited scope. For example, suppose you're developing a system that operates an automated testing machine. Developing a single class to represent the machine, the device under test, and the user isn't an example of focus. Instead, you might decide to write one class to represent the testing machine, one for the device under test, and another to represent the user.

Of course, too much focus can be burdensome, too. Creating objects to represent each individual component in the device under test is probably going too far.

Encapsulation

One of the key features of object-oriented programming is encapsulation. Encapsulation is the process of hiding as much internal detail of your object as possible, so that others can use your object without having to know how it works.

Java, along with other object-oriented languages, allows you to declare each method and variable either public or private, with the public and private keywords. Anything public can be used by the rest of your program to manipulate the object. Anything private is used by the object for its own internal workings. If you think of the automated tester example, the switch that starts testing is public but the electrical wiring inside the machine is private. How that switch works is not important. However, it's important that the public part of your object (the switch in this case) always behaves in the same way.

Suppose you develop a data acquisition system that reads flow data from a sensor and displays the results in gallons per minute. However, the manufacturer of the flow sensor goes out of business and you have to switch to another flow meter that reads in liters per minute. With object-oriented design, solving this problem is easy. You can rewrite the way the class communicates with the sensor to do the appropriate conversion. As long as the new class still supports the original public methods and fields (in gallons per minute), you won't have to change anything else in the rest of your program. That's the power of encapsulation.

In a sense, an object acts like a black box to the rest of your program. Objects hide the internal mechanisms of related methods, and expose only the parts that the rest of your program needs. As long as you don't alter the public interface, you're free to change the private methods and variables without fear of breaking the entire system. You can add new things to the public interface, if necessary, but you shouldn't delete or change anything you've already made public.

A common mistake is the temptation just to make everything public. This defeats the real purpose of encapsulation, you should resist this temptation. Hide private implementation details.

Polymorphism

One of the most important parts of an object-oriented system is polymorphism. This is just a fancy word for establishing "is a kind of" relationships between objects. For example, an A/D converter and a shift register are both types of integrated circuits. An 8-bit serial A/D converter is a specific type of A/D converter.

The idea behind polymorphism is to factor the common parts out of a series of objects. For example, A/D converters, regardless of type, often require a lot of the same code. If you place all the code and data relating to A/D converters in a single class, the classes that represent specific types of A/D chips can extend this base class. This allows the specialized classes to reuse the common code without having to duplicate efforts. It also allows your program to treat all A/D converters the same. If your program deals mostly with generic converter objects (as opposed to specific types), you won't have to make many changes to your code when you want to upgrade to a faster or better chip.

Hypothetically, an A/D class might contain methods to perform engineering unit conversions and to read a value from the device. The value reading method could use low-level routines (like **`initialize`**, **`startConversion`**, etc.) that the derived classes supply. The derived classes could then provide the code that directly talks to the chip. Your program could simply call **`readValue`** and know that all the right things will happen.

With Java you typically get polymorphism (and the associated code reuse) by extending one class from another with the extend keyword. However, you can also get polymorphism without code reuse by using an interface. An interface in Java acts like a skeleton class that defines a number of required methods but doesn't actually implement the inner workings of those methods.

When you define an interface, you don't write any code—you simply provide a list of methods that the interface contains. Any class that implements the interface (with the **`implements`** keyword) must provide the actual code for the methods. All classes that implement a particular interface are polymorphic with each other. This is handy for cases where objects are similar to another object, but not enough so that you want to share code among them.

Class Relationships

Recent versions of Java offer several other ways to express class relationships. You can nest classes, make one class a member of another, create local classes, or create anonymous classes. These are not strictly necessary to create an object-oriented program, but they do offer more options for grouping code together and can help in

many common situations (like event handling, for example). You'll find more about these advanced techniques in any recent Java book.

An Object Oriented Example

Amateur radio operators sometimes use Morse code to communicate over the radio airwaves. In other cases, Morse code identifies remote stations, or even sends telemetry from balloons or rockets. Since Morse code is just a series of short and long pulses, it is easy to make the Javelin Stamp generate them using any of a variety of methods. Suppose you want to send Morse code telemetry. Perhaps you are working on a team, and not everyone knows Morse code. You may be working on several payloads that will each send different information back using Morse code. In either case, you should consider creating an object that knows how to send the data.

That's what you'll see in Program Listing 7.1. The **`MorseOut`** object encapsulates the logic required to send numbers via Morse code. As you examine the code, consider these important points:

- The **`MorseOut`** object completely encapsulates the logic required to generate the code. Anyone who wants to send numbers can simply create the object and call **`send`**.
- The templates for each number are private to the **`MorseOut`** object.
- If you wanted to change the telemetry to some other system, you could simply replace the code in **`MorseOut`**.
- The program does not specify the pin number or the speed of the Morse code, which increases the reusability of the object.
- If you wanted to make the class send more characters, you'd only need to provide a new **`sendChar`** routine (and of course a new table, or other method of translating text into Morse code).

Although there are several things about this class that make it easier to reuse it, you could go even further. For example, creating a class that converted a character to another character would disassociate the conversion to Morse code from the output operations. Connect an LED circuit to P0 to view the Morse output.

Program Listing 7.1 - Send Morse Code Example 1

```
import stamp.core.*;
/**
 * Code to send Morse code
 * <p>
 * This object will send Morse code numbers given ASCII text
 */

public class MorseOut {

  int outPin;  // output pin
  int delay;   // speed base

  private final String[] chars = {
    "-----", ".----", "..---", "...--",
    "....-", ".....", "-....", "--...",
    "---..", "----." };
```

```
  /**
   * Constructor
   * @param pin The pin number to send on (example: CPU.pin0)
   * @param dly The element timing in 100us units
   */
  public MorseOut(int pin, int dly) {
    outPin = pin;
    delay = dly;
  }

  /**
   * Send a single character
   * @param c The character to send
   */
  public void sendChar(char c) {
    int idx;
    String s;
    if (c<'0' || c>'9') return;
    idx=(int)c-(int)'0';
    s=chars[idx];
    for (idx=0;idx<s.length();idx++) {
      sendElement(s.charAt(idx));
    }
    CPU.delay(delay*2);  // character spacing
  }

  protected void sendElement(char el) {
    CPU.writePin(outPin,true);
    if (el=='.')
      CPU.delay(delay);
    else
      CPU.delay(delay*3);
    CPU.writePin(outPin,false);
    CPU.delay(delay);
  }

  /**
   * Send a string (must be numbers only -- skips other characters)
   * @param s The string to send
   */
  public void send(String s) {
    int i;
    for (i=0;i<s.length();i++) {
      sendChar(s.charAt(i));
    }
  }

  /**
   * A test main to try out the object if you like
   * Sends on pin0 and uses a delay of 2000 (200mS)
   */
  public static void main() {
    MorseOut mo = new MorseOut(CPU.pin0,2000);
    mo.send("3141");
  }
}
```

7: Working With Objects

Decoupling the Code

The original **MorseOut** object handles two distinct operations: converting characters to Morse code and sending them out on an output bit. If you aren't interested in reuse, this shouldn't be a problem. However, the best object designs strive to provide objects for each important operation. That means you should decouple this object into two objects, one for each operation.

Consider Program Listing 7.1. It is very similar to the original code, but it accepts a third argument in the constructor. This third argument is a reference to a **CharConvert** object. **CharConvert** is an abstract class Program Listing 7.2 that knows how to convert one character into an equivalent string. Notice that it doesn't convert to Morse code. It simply looks up a string given a character. This class is abstract, you can't create it, but you can extend it.

Program Listing 7.2 - Send Morse Code Example 2

```
import stamp.core.*;
import examples.manual_v1_0.*;

/**
 * Code to send Morse code
 * <p>
 * This object will send Morse code numbers given ASCII text
 */

public class MorseOut2 {

  int outPin;      // output pin
  int delay;       // speed base
  CharConvert cvt; // converter object

  /**
   * Constructor
   * @param pin The pin number to send on (example: CPU.pin0)
   * @param dly The element timing in 100us units
   */
  public MorseOut2(int pin, int dly,CharConvert conv) {
    outPin = pin;
    delay = dly;
    cvt=conv;
  }

  /**
   * Send a single character
   * @param c The character to send
   */
  public void sendChar(char c) {
    int idx;
    String s;
    s=cvt.convert(c);
    for (idx=0;idx<s.length();idx++) {
      sendElement(s.charAt(idx));
    }
  CPU.delay(delay*2);  // character spacing
```

```
  }

  protected void sendElement(char el) {
    CPU.writePin(outPin,true);
    if (el=='.')
      CPU.delay(delay);
    else
      CPU.delay(delay*3);
    CPU.writePin(outPin,false);
    CPU.delay(delay);
  }

  /**
   * Send a string (must be numbers only -- skips other characters)
   * @param s The string to send
   */
  public void send(String s) {
    int i;
    for (i=0;i<s.length();i++) {
      sendChar(s.charAt(i));
    }
  }

  /**
   * A test main to try out the object if you like
   * Sends on pin0 and uses a delay of 2000 (200mS)
   */
  public static void main() {
    MorseOut2 mo = new MorseOut2(CPU.pin0,2000, new MorseNumConvert());
    mo.send("3141");
  }
}
```

This is the **CharConvert** base class:

Program Listing 7.3 - Character Convert

```
package examples.manual v1 0;

abstract public class CharConvert {
  abstract protected int transform(char c);          // transform c to integer
  abstract protected String [] getTransformMatrix(); // get array of conversions
  public String convert(char c) {
    int idx=transform(c);
    if (idx==-1) return "";                          // error
    return getTransformMatrix()[idx];
  }
}
```

You'll need to place this file in a separate file (named **CharConvert.java**) and compile it so that the improved Morse code sending program can use it.

The specific class that handles the code conversion appears below:

Program Listing 7.4 - Convert Numbers to Morse Code

```
package examples.manual v1 0;

public class MorseNumConvert extends CharConvert {
  private final String[] chars = {
    ".----", "..---", "...--",
    "....-", ".....", "-....", "--...",
    "---..", "----." };

  protected int transform(char c) {
    if (c<'0' || c>'9') return -1;
    return (int)c-(int)'0';
  }
  protected String[] getTransformMatrix() { return chars; }
}
```

If you extend **CharConvert** with a regular class, that class must implement two methods: **transform** and **getTransformMatrix**. The **transform** method converts a character to an integer index (or returns –1 if the character is illegal). The **getTransformMatrix** method returns a string array. The **convert** routine handles the actual lookup logic. Notice that while a derived class could override **convert**, it doesn't have to do so (and should avoid it if possible). This allows you to make changes to the public part of all **CharConvert**, derived objects by simply changing that one routine in the base class.

For example, suppose you wanted **convert** to throw an exception if you pass in a bad character. Since the derived classes only extend protected members, you can change the one version of **convert** and any new classes will use that logic. Of course, if any derived classes provide their own **convert**, you'd need to change them as well. You could make **convert final**, that would prevent other classes from changing it. However, that would limit the flexibility of the class, some classes may need to provide custom **convert** routines.

The class that does the actual Morse code conversion is **MorseNumConvert** Program Listing 7.4. This class simply provides the **CharConvert** base class with the information it needs to get the job done. The main program creates a new **MorseNumConvert** object and passes it to the constructor of **MorseOut**. The constructor expects a **CharConvert** object, but since **MorseNumConvert** extends **CharConvert**, that isn't a problem.

At first glance this seems much more complex than the single original file. However, consider this: What if you wanted to make a new class that could send letters and numbers (perhaps to use in a different program). You'd have to make a copy of the entire original class and make major changes to it. Now if you fix or add something in one copy, you'll need to remember to change it in the other copy as well.

With the new scheme, you'd have no problem making the change. Just derive a new class from **CharConvert** and provide the appropriate translation. The same applies if you wanted to change **MorseOut's** output device (perhaps you want to change it to a tone instead of just blinking a light, you only need to change (or override) the **sendElement** method so that it reads:

```
CPU.outputSine(el=='.'?delay*10:delay*30,outPin,10000,0);
CPU.delay(delay);
```

Now a speaker on the output pin will create beeps for the Morse code. Separating the conversion from the output made changes easier to make.

Virtual Peripherals

Another important class of objects are *Virtual Peripherals* (VPs). These are special objects that the Javelin Stamp executes at the same time that your program is running. For example, consider the Javelin Stamp **`Uart`** VP. Once you start it, it runs constantly in the background sending and receiving serial data. That means your program won't miss serial data because it is doing something else when the data arrives. It also means that your program isn't tied up waiting while sending serial data.

How does the Javelin Stamp juggle your program and VPs? The same way that personal computers (like a PC running Windows) multitask programs. Your program and each VP share the Javelin Stamp. Every 8.68 µs, the Javelin Stamp allows the VPs to execute. Each VP is designed to execute quickly on each time slice. Once the VPs have had a chance to run, your program resumes right where it left off.

Don't worry about the VP consuming so much time that your program will slow down. Each VP is especially designed to do just a small amount of processing on each time slice. The **`Uart`** VP, for example, does not send or receive an entire byte of data on each time slice (that would take far too long). Instead, the UART might start sending a bit during one time slice and update the output for the next bit value 12 time slices later (remember, there are roughly 12 time slices in a single bit at 9600 baud).

Of course, if you added too many VPs, the total processing time might start to add up. That's why the Javelin Stamp limits you to six VPs at once. If you try to exceed this limit, you'll cause an **`IllegalArgumentException`**.
Most VPs will install themselves when you create them. You can manually install or uninstall a VP using **`CPU.installVP`** and **`CPU.removeVP`**.

The Javelin Stamp includes several VPs, and you can download the latest from the Parallax site. Here's a brief overview of the core VPs:

- **`Uart`** – Send or receive RS232 data (bi-directional communications requires two **`Uart`** VPs).
- **`PWM`** – Creates pulse width modulation on an output pin. You can use PWM to control a motor speed, modulate light brightness, or generate an analog voltage (using an RC filter).
- **`TIMER`** – Time events with 8.68us precision. When you create a **`Timer`** object, the Javelin Stamp installs the Timer VP. However, once the VP is present, you can create more **`Timer`** objects and they will use the same VP.

VPs are a perfect example of how you can take advantage of object orientation. By using these objects you can perform tasks that would be difficult or impossible to do without them.

7: Working With Objects

A Timer Example

Suppose you want to write a program that will flash two LEDs. Making LEDs flash is usually not the eventual goal of a project (unless you are making a holiday display, perhaps). However, what if you want the two LEDs to blink while you are doing something else? Perhaps one LED flashes to let you know your software is running (it could even reset a watchdog timer). The other LED might flash to let you know that data is arriving over a serial port. The question is how to make the LEDs flash at a steady rate while performing other tasks?

By using multiple timers, the job is easy. Remember, the first timer object installs the VP. Subsequent timers use the same VP, so even though you can only have 6 VPs, you can use many timers while only consuming one of your allotted VPs. Program Listing 7.5 shows the timer code:

Program Listing 7.5 - Simple Timer Demo

```
import stamp.core.*;

// This program blinks an LED circuit connected to P0 every 200 mS and
// blinks an LED circuit connected to P2 every 300 mS using the Timer object.

public class SimpleTimer {

  public static void main() {
    Timer t1 = new Timer();                          // timer for first LED
    Timer t2 = new Timer();                          // timer for second LED

    boolean led0=false;
    boolean led2=false;

    t1.mark();                                       // Start timer t1
    t2.mark();                                       // Start timer t2

    while (true) {                                   // do forever…

      String msg = "test\n";
      CPU.message(msg.toCharArray(),msg.length());   // Print Message

      if (t1.timeout(200)) {                         // If 200 mS LED interval
        led0=!led0;                                  // set LED
        CPU.writePin(CPU.pin0,led0);
        t1.mark();                                   // start new time period
      }                                              // end if

      if (t2.timeout(300)) {                         // If 300 mS LED interval
        led2 = !led2;                                // Negate LED
        CPU.writePin(CPU.pin2,led2);
        t2.mark();                                   // start new time period
      }                                              // end if
    }                                                // end while
  }                                                  // end main
}                                                    // end class declaration
```

Notice that the code loops forever doing some work (in this case, just printing a string with **`CPU.Message`**). During the loop, the code examines two **`Timer`** objects to test for a timeout (one for 200 ms and the other for

300 ms). If the timeout occurs, the program flips the state of the LED and then uses the *Timer*'s **mark** method to start a new interval.

Object-Oriented Opportunity

When designing objects, remember to keep each object focused, make each object as self-contained as possible, and factor common code into base classes.

Having a hammer doesn't mean you can build a house. Using an object-oriented tool like Java doesn't mean you're writing object-oriented code. Look for ways to use objects in your coding to make reusing code easier and to make your program easier to understand.

With so many ways to model objects, you're sure to come up with an elegant, succinct representation for nearly any problem. Elegant representations tend to generate elegant implementations. Better still, a great implementation will be more robust and maintainable than an ad hoc solution.

Example 1.

```
public class datanode {
  static class converter {
    // members of converter
  }
    // members of data node
}
```

Example 2.

```
public class mainclass {
  void somemethod(int x) {
    class helper {
      // some class that only
      // somemethod needs
    }
  }
  void anothermethod(int y) {
    // . . .
  }
}
```

Example 3.

```
class obj {
  void amethod(Object o) {
    // some method that
    // requires an object
  }
}

// . . .
```

```
obj.amethod(
  new {
    public void object_func_1(void) {
      // some code here
    }
    public void object_func_2(void) {
      // more code here
    }
  } // end of nested class
);
```

The Javelin Stamp language depends on objects to perform a variety of tasks. That means understanding the details of objects is crucial if you want to get the most from the Javelin Stamp. This chapter will provide a reference for the core language classes – that is, classes that don't directly interact with the hardware. That includes objects in **the java.lang**, **java.io**, **java.util**, and **stamp.util** packages.

Objects are also critical when controlling hardware, you'll find more about hardware-related objects in Chapter 8. Don't forget that Parallax and third parties can create new objects that will help you perform different tasks or use different hardware features. For an up-to-date list of available objects, be sure to checkout: www.parallax.com

> *It is important to realize that in addition to the methods and fields specified for each object, the object also inherits the public methods and fields of the base classes. Since all classes extend* **Object***, for example, all objects have an* **equals** *method. Some classes override this method and will separately document it, but others use the default and do not list it explicitly in their documentation. In addition, classes that don't specify any constructors have default constructors that take no arguments.*

The *java.lang* Package

The **java.lang** package contains fundamental types that practically all programs will require. Because of this, the compiler always looks in **java.lang** to find object names. So while you can write **java.lang.Boolean**, you don't have to do it that way – a simple Boolean will suffice. In addition, you never have to **import** the **java.lang** package since every program imports it anyway.

Many of the classes in this package represent wrappers for the fundamental types (b**oolean** for example). In addition, you'll find objects that consist of **static** members that are more or less global in scope (like **Math** or **System**). Objects that are exceptions are those that derive from **Throwable** like **Exception** or **Error**.

Boolean

The **Boolean** class provides an object that wraps a basic **boolean** type. Note that the class does not have a default constructor.

Base Class: **Object**

Fields:
static Boolean false – A **Boolean** object that contains **false**.
static Boolean true – A **Boolean** object that contains **true**.

Constructors:
Boolean(boolean value) – Creates a new object with the specified value.
Boolean(String s) – Creates an object that is true if the string's value is "**true**" (the comparison is not case sensitive).

Methods:
String toString() – Converts the object to a string (**true** or **false**).
boolean equals(Object o) – Tests objects for equality.
int hashcode() – Returns the hashcode of the object (1231 if the object value is **true**, otherwise the hashcode is 1237).
boolean booleanValue() – Returns the value of the object.
static boolean valueOf(String s) – Returns boolean value of supplied string.

Error

The **Error** object is the base class for all non-checked exceptions. A non-checked exception is one that can occur at any time, and the compiler does not require you to catch them (contrast this to checked exceptions, which generally derive from **Exception**).

Base Class: **Throwable**

Common derived classes: **OutOfMemoryError**

Methods:
String getMessage() – Returns an error message appropriate to the error.

Exception

The **Exception** object is the base class for all checked exceptions. A checked exception must be explicitly handled in your code via a **try/catch** block or by using the **throws** clause in your method declaration. Contrast this to non-checked exceptions, which derive from **Error**. Although all checked exceptions derive from **Exception**, not all objects that derive from **Exception** are checked exceptions. Notably, those that derive from **RuntimeException** are not checked.

Base Class: **Throwable**

Common derived classes: **RuntimeException**

Methods:
String getMessage() – Returns an error message appropriate to the error.
IllegalArgumentException - The IllegalArgumentException object is what a method throws when it determines that you have passed an illegal argument to it.

Base Class: **RuntimeException**

Methods:
static RuntimeException throwIt() – Throw a run-time exception.

IndexOutOfBoundsException

Your program will throw an **IndexOutOfBoundsException** if you attempt to access an array with an illegal array index.

Base Class: **RuntimeException**

Methods:
static RuntimeException throwIt() – Throw a run-time exception.

Math

The **Math** class consists solely of **static** methods. These methods are effectively global Since you don't need to create the **Math** object to use these methods. For example, to find the absolute value of a number, you don't need to instantiate a **Math** object. Instead, just call **Math.abs()**.

Methods:
static int abs (int a) – Returns the absolute value of the argument.

static int min (int a,int b) – Returns the smallest of **a** and **b**.

static int max (int a,int b) – Returns the largest of **a** and **b**.

NullPointerException

Your program will throw a **NullPointerException** if you attempt to access an object reference that is equal to **null**.

Base Class: **RuntimeException**

Methods:
static RuntimeException throwIt() – Throw a run-time exception.

Object

Object is the top-level base class for all objects, even those that don't explicitly extend anything. Public methods of **Object** are available in all objects, since all objects extend **Object**.

Methods:
boolean equals (Object o) – Compares this object to another object. Returns **true** only if both objects are references to the same actual object. Many classes will provide override versions of **equals** to make comparisons of the object's value (e.g., **String** will test to see if the two strings are equal).

OutOfMemoryError

Your program will throw an **OutOfMemoryError** if you run out of memory during program execution.

Base Class: **Error**

Methods:
static OutOfMemoryError throwIt() – Throw a run-time exception.

RuntimeException

The **RuntimeException** class is the base class for many unchecked exceptions.

Base Class: **Exception**

Common derived classes: **IllegalArgumentException, IndexOutOfBoundsException, NullPointerException, java.util.NoSuchElementException**

Methods:
static RuntimeException throwIt() – Throws a **RuntimeException**.

String

The **String** class represents fixed, unchanging, text data. For the Javelin Stamp, strings consist of 8-bit ASCII bytes. In addition to a constructor, you can form a constant string by simply enclosing characters in double quotes.

Although you can build a string by concatenating two or more strings, it is a better practice to use **StringBuffer** to build a string. Consider this code:

```
String aString = "Hello ";
String bString = "Parallax";
aString = aString + bString;
```

Once compiled, this code creates a new **aString** object and discards the original one. The memory used by the original **aString** is lost until the Javelin Stamp resets. See **StringBuffer** for a better way to manipulate and edit strings at runtime.

Base Class: **Object**

Constructors:
String() – Creates a new, empty string (length=0).
String(char[] data) – Creates a new string and initializes it from the character array.
String(String s) – Creates a new string with the same contents as **s**.

Methods:
char charAt(int index) – Returns the character at the specified index (the first character is at index 0).

boolean equalsIgnoreCase (String s) – Returns **true** if the specified string has the same value as the string object without considering case.

int length () – Returns the length of the string.

setCharArray(char [] ary) – Sets the character array that contains the string's characters. The string will actually use the specified array to hold its characters. It does not make a copy of the characters.

char[] toCharArray () – Converts the string to a character array.

String toString() – Returns the string representation of the string. This is useful for cases where a generic object reference is really a string.

static String valueOf (int v) – Converts the integer provided into a string. Use this method judiciously because it creates a new String object each time it is used. You can avoid unwanted String objects by using the **valueOf (int v, String Result)** method discussed next.

static String valueOf (int v, String result) – Converts the integer provided into a string, storing the result in the **result** argument. The **result** string must have at least 6 characters in it, or an exception will occur. **Result** is a String object of your choosing. You can create one static String that contains six characters and use it to store each conversion you make. This can really come in hand when sending numeric messages to a serial device.

Example:

```
String s = "123";
System.out.println(s.charAt(1));                    // prints "2"
```

StringBuffer

While the contents of a **String** can't change, **StringBuffer** is changeable. You can use a **StringBuffer** to avoid the overhead involved in creating and destroying many string objects when you are making many changes to the contents of a string. This is especially important for the Javelin Stamp since it does not have garbage collection.

Base Class: **Object**

Constructors:
StringBuffer() – Creates a new, empty buffer.
StringBuffer(int length) – Creates a new buffer with the specified length.
StringBuffer(String s) – Creates a new buffer that contains the contents of the specified **String** object.

Methods:
StringBuffer append (int [] str, int length) – Appends characters from an array of characters to the end of the string buffer. The **length** argument determines how many characters to append.

StringBuffer append(String str) – Adds a string to the end of the buffer.

int capacity () – Returns the current size of the buffer. This is the number of potential characters – the actual number of characters currently in the buffer may be less.

char charAt (int index) – Returns the character at the specified index. The first character in the buffer is at index 0.

StringBuffer delete (int start, int end) – Deletes characters from the buffer starting at the **start** index, up to, but not including the character at the **end** index.

StringBuffer insert (int offset, char c) – Inserts a character at the specified offset. The character originally at the offset, and all characters to the right, move over by one position to make room for the new character.

int length () – Returns the length of the string currently in the buffer.

String toString () – Converts the buffer to a **String**.

Example:

```
StringBuffer sb = new StringBuffer("ATDT");
sb.append(getTelephoneNumber());       //getTelephoneNumber returns String
dial(sb.toString());                   //dial expects a String argument
```

System

The **System** object holds **static** items that apply to the system as a whole. It is essentially a holder for global methods and variables.

Base Class: **Object**

Fields:
static PrintStream out – A **PrintStream** used to send data back to the system console on the host PC.

Example:

```
System.out.println("Parallax");
```

Throwable

Throwable is a base class for all exceptions.

Base Class: **Object**

Common derived classes: **Error**, **Exception**

Methods:
String getMessage () – Returns a message appropriate for the exception.

The *java.io* Package

The **java.io** package contains the **PrintStream** class. This is the way that the Javelin Stamp can write data out to a stream. A stream might be a serial I/O port, or any other input and output device that works with characters.

PrintStream

Base Class: **Object**

Methods:
print (boolean b), print(char c), print(int i), print(String s) – These methods write the text representation of the argument to the output stream without any additional white space.

println (boolean b), println(int i), println(String s) – These methods write the text representation of the argument to the output stream followed by a new line character.

The *java.util* Package

This package has useful classes that are not part of the core language.

Random

The **Random** object is useful for creating random integers. The random number algorithm uses a seed value. Two **Random** objects with the same seed will generate the same sequence of randomly distributed numbers.

Constructors:
Random() – Creates an object with the default seed (a fixed number).
Random(int seed) – Creates an object with the specified seed.

Methods:
int next() – Returns the next number in the random sequence.

Example:

```
Random r1=new Random();
Random r2=new Random();
Random r3=new Random(139);
int j;
for (j=0;j<100;j++)
  System.out.println(r1.next() + " " + r2.next() + " " + r3.next());
```

The *stamp.util* Package

This package has useful classes that are not part of the core language and are not part of ordinary Java.

Expect

Expect allows you to wait for particular input from a **Uart** object.

Constructors:
Expect() – Default constructor.

Methods:
static boolean string(Uart input, String string, int timeout) – Wait for the specified string. Returns **true** on success or **false** if the timeout expires. The **timeout** value is in 100us units.

List

The **List** object allows you to create and manage an ordered list of items.

Constructors:
List(int maxsize) – Creates a list of objects with the specified maximum size.

Fields:
protected Object[] list – The items in the list.
protected int numObjects – the total number of objects.

Methods:
void add(int index, Object o) – Adds the specified object at a particular position in the list.
void add(Object o) – Adds the specified object to the end of the list.
Object get(int index) – Retrieves the object at the specified position on the list.
int size() – Retrieves the number of objects in the list.

LinkedList

This object allows you to make a linked list of items.

Constructors:
LinkedList() – Default constructor.

Fields:
protected LinkedListItem list – The start of the list.

Methods:
void addItem(LinkedListItem) – Adds the item to the list.
LinkedListItem getFirst() – Retrieves the first item in the list.
LinkedListItem getLast() – Retrieves the last item in the list.
LinkedListItem getNext(LinkedListItem item) – Retrieves the next item in the list.
LinkedListItem getNextLoop(LinkedListItem item) – Retrieves the next item in the list. At the end of the list, wrap around to the first item in the list.
void removeItem(LinkedListItem item) – Removes the item from the list. This does not reclaim the memory the item uses on the heap. If you want to reuse the object, it is up to you to hold a reference to it.

LinkedListItem

This class represents items in a linked list.

Constructors:
`LinkedListItem()` – Default constructor.

Fields:
`LinkedListItem nextItem` – The next item in the list.

The *stamp.core* Package

When you want to control the Javelin Stamp's hardware, you'll turn to the objects in the **`stamp.core`** package. The methods and fields in this package allow you to directly control the Javelin Stamp's I/O pins. In addition, you'll find classes that can treat I/O in special ways (for example, generate PWM or perform RS-232 input and output).

Many of the objects in this package are ordinary objects. However, some are virtual peripherals (VPs). These VPs operate at the same time your program is running and have special requirements (see Chapter 7 for more information about VPs).

ADC

The ADC class is a VP that performs a delta sigma analog to digital conversion with the aid of two external resistors and one external capacitor (see Figure 9.1). This conversion requires two I/O pins, **`outPin`** pin sends pulses and **`inPin`** monitors the voltage across the capacitor. The VP attempts to keep the capacitor charged to 2.5 V. Since the input voltage will affect how many pulses the VP has to send to keep the capacitor charged to that level, the VP can compute what the input voltage is.

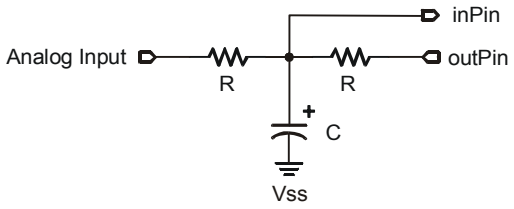


Figure 9.1 Circuit for use with ADC VP

The ADC VP updates the voltage measurement every 2.1 ms. Once it determines the voltage on the input, it stores it until it completes the next conversion is completed.

Base Class: *VirtualPeripheral*

Constructor

`ADC(int inPin, int outPin)` – Constructs an *ADC* object using the specified input pin (**`inPin`**) and output pin (**`outPin`**). The VP runs continuously (and consumes one VP slot) until you uninstall it.

Methods

`int value()` – Returns the value between 0 and 255 that corresponds to the most recently completed complete analog to digital conversion. 0 corresponds to 0 V and 255 corresponds to 5 V. Since a 5 V scale maps to an 8-bit measurement (0 to 255) it means that if 2.5 V is measured, **`value()`** will return 127, and if 1.25 V is measured, **`value()`** will return 63, etc.

Example
Program Listing 9.1 returns analog values that correspond to the analog input shown in Figure 9.1. The code listing starts the ADC, waits so there is time for the VP to complete a conversion, and then displays the reading. Use the following values and I/O pin connections:

- R = 10 kΩ
- C = 1.0 µF
- inPin = P9
- outPin = P8
- Analog Input: For those of you with a variable DC supply, give it a try. You can also try 1, 2, and 3 AAA batteries in series. Connect the negative terminal of the battery or series of batteries to Vss and the positive terminal to the circuit's analog input. You can also use a potentiometer as a voltage divider to generate variable voltage.

Program Listing 9.1 - ADC Demo

```
import stamp.core.*;

public class ADCDemo {

  public static void main() {
    ADC adc = new ADC(CPU.pin9,CPU.pin8);
    CPU.delay(1000);                                  // wait to acquire a value
    System.out.println(adc.value());
  }
}
```

Button

When setting the time on a good alarm clock, it lets you advance its time as follows: You can press and release the button repeatedly to advance the time in increments of one. The circuit for this button might be either of the ones shown in Figure 9.2. To prevent the time from incrementing in increments larger than one due to bouncing mechanical contacts in but button, the microcontroller but it delays a small amount of time between when you press the button an when it acknowledges the button press. In this way, the microcontroller debounces or filter out rapid on/off toggles that can occur before the button state has settled. This delay is governed by the Button object's debounceDelay variable. After that, you can hold the button down for a while to tell the alarm clock that you want it to start advancing the time rapidly. This delay is governed by the Button object's repeatDelay variable. The alarm clock then advances that time at a rate that might seem fast to you, but it is still a repeat rate that involves timed delays for the microcontroller. These repeated timed delays are governed by the Button object's repeatRate variables. All three of these variables specifiy delays in milliseconds.

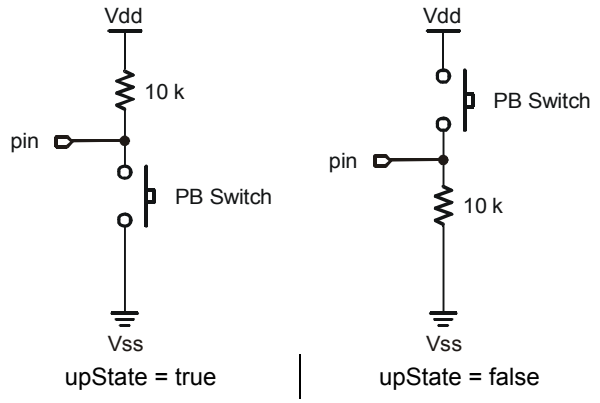


Figure 9.2
Circuit for use with button

The **button** object is typically used inside a loop, and two of its methods should be called each time through: **buttonDown** and **getState**. The **buttonDown** method should be called repeatedly to monitor and update the debounce and auto-repeat timers. It returns true once, when either the debounce or repeat timer has expired; otherwise, it returns false every time it gets called.

The **getState** method returns **BUTTON_UP** whenever the button is not pressed. If the **Button** object's **debounceDelay** variable has been set to a positive, non-zero value, **getState** will return **BUTTON_DEBOUNCE** as soon as the button is pressed, and each time thereafter. After the timer that uses the **debounceDelay** variable has expired, **buttonDown** will toggle false-true-false. Then the **getState** method will return **BUTTON_DOWN** every time it gets called. If the **Button** object's **debounceDelay** variable is left at its default value of zero, when the button is pressed, **buttonDown** will toggle false-true-false immediately, after which **getState** will return **BUTTON_PRESSED**.

If the **Button** constructor with the **repeatDelay** and **repeatRate** arguments is used, here's what will happen if you continue to hold down the button. When the the timer that's governed by **repeatDelay** has expired, **buttonDown**, which has been returning false, will return true, then false again the next time it gets called (and every time thereafter until the next timer expires). After the false-true-false toggle, **getState** will return **BUTTON_AUTO_REPEAT** (instead of just **BUTTON_DOWN**) every time it gets called. If you keep holding the button, each time the **buttonDown** method gets called after the timer governed by **repeatDelay** expires, it will return true, then repeatedly return false again until the next time the repeatDelay timer expires. The **getState** method will continue to return **BUTTON_AUTO_REPEAT**, so the toggle of what **repatDelay** returns is the program's indication that the **repeatDelay** timer has expired again.

At any point during this sequence, the button can be released, to restart the sequence, in which case **buttonDown** returns false, and **getState** returns **BUTTON_UP**.

Base Class: **Object**

Fields:
static int BUTTON_AUTO_REPEAT – The button is down and auto-repeating.

static int BUTTON_DOWN – The button is down.

static int BUTTON_DEBOUNCE – The button has not been down for longer than the debounce period.

static int BUTTON_UP – The button is up.

int debounceDelay – The time for debouncing the switch (in milliseconds).

int repeatDelay – The time from **BUTTON_DOWN** to **BUTTON_AUTO_REPEAT** in milliseconds.

int repeatRate – The time between **buttonDown** method false…false-true-false… in milliseconds.

Constructors:
Button(int pin, boolean upState) – Creates a button on the specified pin (for example, **CPU.pin5**). The buttons default state is specified by upstate. So if the switch is normally open to ground, and there is a pull up resistor on the input pin, you'd specify **true** for this parameter.

Button(int pin, boolean upState, int repeatDelay, int repeatRate) – Creates an autorepeating button. The first two parameters are the same as for the first constructor. The last two parameters set the repeat delay and rate in milliseconds.

Button(int pin, boolean upState, int repeatDelay, int repeatRate) – Creates an autorepeating button. The first two parameters are the same as for the first constructor. The last two parameters set the repeat delay and rate.

Methods:
int getState() – Returns **BUTTON_UP**, **BUTTON_DOWN**, **BUTTON_STILL_DOWN**, or **BUTTON_AUTO_REPEAT** to reflect the current state of the button and the timer governed state it's in.

Example:
Here is a simple program that monitors a button on pin 15. Each time you press and hold the button, the Javelin Terminal will advance from **BUTTON_UP** to **BUTTON_DOWN** and then to the **BUTTON_AUTO_REPEAT** states. By adding the command **blueButton.debounceDelay = 2000;** to the program, you can also observe the **BUTTON_DEBOUNCE** state when pressing and holding the button.

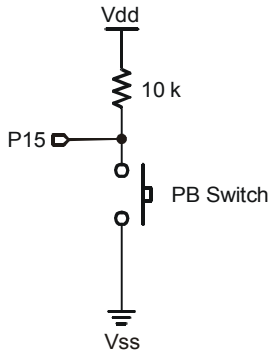


Figure 9.3
Circuit for use with button example

Program Listing 9.2 - Button Demo

```
import stamp.core.*;

public class ButtonTest{
  public static void main() {
    Button blueButton = new Button(CPU.pin15, true, 5000, 10);
    int lastState = Button.BUTTON UP;
    System.out.println("Press & release button.");
    System.out.println("Monitoring button state...");

    while ( true ) {
      int state = blueButton.getState();

      if ( blueButton.buttonDown() )
        System.out.println(" ");
      if ( state != lastState ) {

        System.out.print("State: ");
        switch (state) {
        case Button.BUTTON UP:
          System.out.println("BUTTON UP");
          break;
        case Button.BUTTON DOWN:
          System.out.println("BUTTON DOWN");
          break;
        case Button.BUTTON AUTO REPEAT:
          System.out.println("BUTTON AUTO REPEAT");
          break;
        case Button.BUTTON DEBOUNCE:
          System.out.println("BUTTON DEBOUNCE");
          break;
        default:
          System.out.println("** Unknown state ***");
          break;
        }
```

```
        lastState = state;
      }
    }
  }                                                    //end main
}                                                      // end class declaration
```

CPU

The *CPU* class contains specific calls to help you manage the Javelin Stamp processor resources, including I/O pins. All the members of *CPU* are static, so you don't need to create an instance of the object – you simply call the members, as needed.

Base Class: **`Object`**

Fields: **`static int MAX_NUM_VPS`** – The maximum number of virtual peripherals allowed (6 in the standard version). You can use this to make your software aware of how many VPs are allowed.

`static int pin0`, **`pin1`**, **`pin2`**, **`pin3`**, **`pin4`**, **`pin5`**, **`pin6`**, **`pin7`**, **`pin8`**, **`pin9`**, **`pin10`**, **`pin11`**, **`pin12`**, **`pin13`**, **`pin14`**, **`pin15`** – These constants allow you to access the pins of the Javelin Stamp when using methods like **`readPin`** and **`writePin`**. If you wish to use sequential numbers to access pins, use the **`pins`** array.

`static int[] pins` – This array contains the pin constants (e.g., **`pin0`**, **`pin1`**, etc.) in sequence. This is useful if you want to access bits in a loop. For example:

```
for (int i=0;i<pins.length;i++) CPU.writePin(pins[i],true);
```

`static int POST_CLOCK_LSB`, **`POST_CLOCK_MSB`**, **`PRE_CLOCK_LSB`**, **`PRE_CLOCK_MSB`** – Constants for use with the **`shiftIn`** and **`shiftOut`** methods. See the description for **`shiftIn`** and **`shiftOut`** (below) for more details.

`public static final int PORTA` - A constant representing the first I/O port on the Javelin Stamp. This port contains pins 0-7.

`public static final int PORTB` - A constant representing the second I/O port on the Javelin Stamp. This port contains pins 8-15.

Methods:

carry

`static boolean carry()` – This method returns the internal carry bit's state. After a 16-bit addition or subtraction, this flag will be set if a carry or borrow occurred. For example, here is a 32-bit addition routine:

```
int[] word0 = new int[2];
```

```
int[] word1 = new int[2];
word0[0]=0x5aaa;
word0[1]=0x0020;                    // 205aaa
word1[0]=0x7999;
word1[1]=0x00FF;                    // 00ff7999

// compute word1=word1+word0
word1[0]=word1[0]+word0[0];
if (CPU.carry()) word1[1]++;
word1[1]=word1[1]+word0[1];

// compute word1=word1+word0 again
word1[0]=word1[0]+word0[0];
if (CPU.carry()) word1[1]++;
word1[1]=word1[1]+word0[1];
```

Be careful when mixing **carry** with other expressions, you can't be sure which the compiler will do first. For example, this would be a bad idea:

```
word1[1]=word1[1]+word0[1]+CPU.carry()?1:0;
```

The **carry** call in this case might reflect the carry of the addition that occurs on the same line instead of the previous addition.

count

static int count(int timeout, int pin, boolean edge) – This method counts transitions (or edges) sensed on the specified pin. The **timeout** parameter sets the amount of time the Javelin Stamp will examine the pin (in 100us units). The **pin** argument determines the pin number to use (e.g., **CPU.pin0** or **CPU.pins[3]**). Finally, the **edge** parameter determines if the Javelin Stamp should count rising edges (**true**) or falling edges (**false**).

Example:

Count falling edges for 200 ms on P10

```
CPU.count(2000, CPU.pin10, false);
```

delay

static void delay(int period) – The **delay** method pauses program execution for the specified period in 100us units. For example, to delay for 2 seconds, you'd use a **period** of 20,000. Executing a **delay** does not affect the execution of virtual peripherals. Although the **delay** call accepts signed integers, it treats them as unsigned numbers. So setting **period** to –1, for example, will delay for 65535 time units. If you need a delay greater than 32767 units, you can left shift a variable to obtain the delay you want.

Example:

```
int dlytime;
dlytime=30000;
System.out.println("3 second delay");
CPU.delay(dlytime);
System.out.println("6 second delay");
CPU.delay(dlytime << 1);                              // 60000 x 100us = 6 sec
System.out.println("done");
```

You can not however, use a constant because the compiler will pre-compute the result and you'll get an error since 60000 exceeds the size of a normal signed integer.

installVP

`static void installVP(VirtualPeripheral vp)` – When you create a virtual peripheral object, it will typically install itself. However, you can also use this method to install a VP manually. This might be useful if you've previously unloaded the VP with **`removeVP`** and want to reinstall it.

Example:

The example accompanying the **`removeVP`** method showed how to remove a VP named **`pwm`** to make room for some other VP. Let's say that the VP you want to replace **`pwm`** with is a **`DAC`** object named **`dac`**. This example also assumes that the **`dac`** object has already been declared and removed once. Reinstalling the **`dac`** object involves **`installVP`** plus reassigning it a value:

```
DAC dac = new DAC(CPU.pin2);
dac.update(125);
CPU.removeVP(dac);

// Later in the program after pwm was installed and removed...
CPU.installVP(dac);
dac.update(125);
```

message

`static void message(char [] data, int length)` – Sends a message to the Messages from Javelin window. Note that the text is a **`char`** array, not a string. You can use **`String.toCharArray`** if you want to provide text in a **`String`** object. You can also use **`System.out.println`** to send strings to the Messages from Javelin window.

Example:

```
char [] characters = {'a', 'b'};
CPU.message(characters,2);
String s = "CDE";
CPU.message(s.toCharArray(),3);
```

nap

`static void nap(int period)` – Places the processor in a low power sleep state. All operations cease while the Javelin Stamp is napping.

> *The Javelin Stamp can only call nap as a foreground process with no background processes running. Before calling this method, you must use* **`VirtualPeripheral.removeItem`** *to uninstall any VPs that are installed. See example below.*

The **`period`** argument can range from 0 to 7, depending on how long you want to the Javelin Stamp to nap (see below). However, the **`nap`** time is only approximate and should not be used for timing where accuracy is required. The primary reason you'll use **`nap`** is to conserve power when operating the Javelin Stamp from batteries. Here are the values for the **`period`** argument:

0 – 16 ms
1 – 32 ms
2 – 64 ms
3 – 128 ms
4 – 256 ms
5 – 512 ms
6 – 1024 ms
7 – 2048 ms

Example:

```
System.out.println("Time to take about a 2-second nap");
CPU.nap(7);
System.out.println("Nap completed.");
```

pulseIn

`static int pulseIn(int timeout, int portPin, boolean pinState)` – This method measures a pulse using a 8.68us timer. You can specify a maximum amount of time for this command to run (again, using 8.68us units) with the **`timeout`** parameter. The **`portPin`** parameter specifies which pin to monitor (for example, **`CPU.pin0`**). If the **`pinState`** parameter is **`true`**, the method measures a high pulse, otherwise it looks for a low pulse.

Example:

```
// Count rising clock edges for 173.6 µs.
int pulseValue = CPU.pulseIn(20000, CPU.pin0, true);
```

The Javelin Stamp will not exceed the timeout value while executing this command. Suppose you set the **`timeout`** parameter to 100 (8.68 us). If the Javelin Stamp waits for 500us and senses a pulse, it will still

terminate this command after 868 us, even if the pulse is not complete. In this case, the return value will be –1, indicating that no stop edge was detected. If no starting edge occurs during the timeout period, the method returns 0. Obviously, pulses shorter than 8.68us may escape detection.

The return value and the **timeout** value are actually unsigned. Even though integers nominally range only to 32767, these values actually extend beyond, but the compiler treats them as negative numbers. There are several ways to deal with this problem. One easy way is to use shifts, but it does cut your timing resolution in half. For example, suppose you want to wait for 195.3 ms. This corresponds to a **timeout** argument of 45000 – too big for a signed integer. However, what if you pretended, the actual resolution was not 8.68us, but double that (8.68 us). Now the **timeout** argument for 195.3 ms would be 22500 – an acceptable number.

Of course, the resolution is 8.68us, so you would have to double the **timeout** value using a shift. You can't, however, use a constant, because the compiler is smart enough to pre-calculate the constant and will detect it is "too big". For example:

```
int time_out = 22500;
int pulsed = CPU.pulseIn(time_out << 1, CPU.pin0, true);   // wait 390.6mS
```

However, you can not write:

```
CPU.pulseIn(22500 << 1, CPU.pin0, true);
```

The output in this case, might also be negative, indicating the pulse was longer than 142.2ms. Simply shift the result right, and again, pretend the resolution is 8.68us. For example, suppose the above example detected a pulse of 173.6 ms (40000). This will show up as a negative number (-255). However, you can shift it down to use 8.68us units:

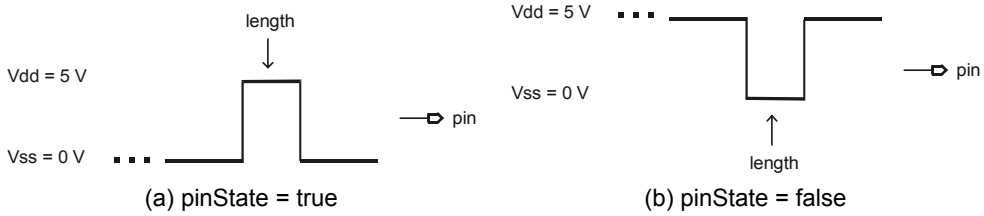
```
pulsed=pulsed>>>1;
```

If you don't want to lose resolution, you'll need to analyze the results as a binary number. For example, using the **&** operator, you can strip the most significant bit (the sign bit). This will provide the amount over 32768. For example:

```
System.out.println(pulsed & 0x7FFF);
```

If **pulsed** contained 40000 (173.6 ms) the display would show 7232 (40000 minus 32768).

Figure 9.4
pulseIn
Measurements



Example
This method takes three arguments: a time out duration, a pin number, and a boolean that indicates if you are looking for a one pulse (**true**) or a zero pulse (**false**).

Program Listing 9.3 - Pulse Class 1

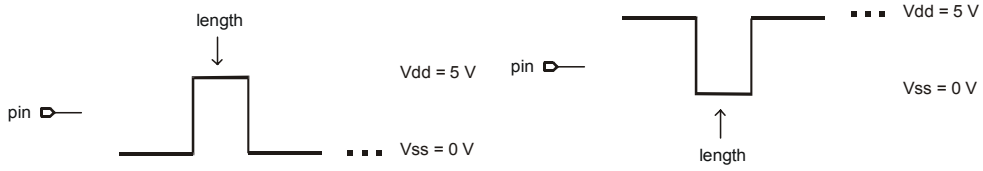
```
import stamp.core.*;

public class PulseClass1 {
  static int n;
  public static void main() {
    while (true) {
      n = CPU.pulseIn(32767,CPU.pin14, false);
      System.out.println(n);
    }
  }
}
```

pulseOut

static void pulseOut(int length, int portPin) – When you call **pulseOut**, the Javelin Stamp inverts the state of the specified **portPin** (for example, **CPU.pin0**) and holds it in that state for the time you specify in **length**. The **length** argument is in terms of 8.68 us units and is unsigned (see the discussion under **pulseIn** for more information on dealing with unsigned numbers). A positive pulse is a low-high-low sequence as shown in Figure 9.5a while a negative pulse is a high-low-high sequence as shown in Figure 9.5b. If you want to deliver a positive pulse, set the pin low (**false**) first using **CPU.writePin**. Likewise, if you want to deliver a negative pulse, set the pin high (**true**) first using **CPU.writePin**.

Figure 9.5
pulseOut Pulses



(a) Positive Pulse (b) Negative Pulse

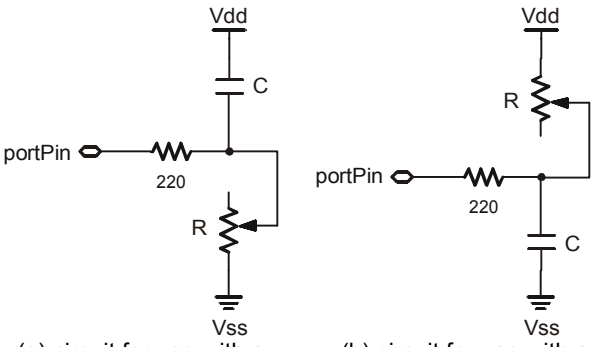
rcTime

static int rcTime(int timeout, int portPin, boolean pinState) – This method measures the amount of time required for the pin you specify to reach a desired state. The time and the **timeout** parameter are in 8.68 µs units. The **portPin** argument is one of the pin constants (like **CPU.pin0**), and **pinState** is the desired ending state (**true** or **false**). If the pin does not reach the specified state before the time out expires, the call returns –1.

The **rcTime** method is useful in applications where you want to measure the charge or discharge of an RC (resistor/capacitor) network. You might want to do this, for example, to read a potentiometer, a thermistor, or any resistive or capacitive sensor.

When **rcTime** executes, it starts a counter that increments every 8.68 µs. It stops this counter as soon as the specified pin reaches **pinState**. If **portPin** is not in the opposite of **pinState** when the instruction is executed, **rcTime** returns 1, since the instruction requires one timing cycle to discover this fact. If pin remains in the opposite of **pinState** longer than the number of 8.68 µs timing cycles specified in the **timeOut** argument, **rcTime** returns 0.

Figure 9.6 RCTime circuits for recommended



Before **rcTime** executes, the capacitor must be put into the state specified in the call. For example, with Figure 9.6a, the capacitor must be discharged until both plates of the capacitor approach 5 V using:

```
CPU.writePin(CPU.pin4, true);
CPU.delay(1000);
```

Then, you can call the **rcTime** method. Assuming **i** is an **int** value, you can use:

```
i = CPU.rcTime(32767,CPU.pin4,false);                  // timeout of 284 ms
```

It may seem counterintuitive that discharging the capacitor is accomplished by sending a true signal, but remember that a capacitor is charged when there is a voltage difference across its plates. When both plates are in the neighborhood of 5 V, there is almost no voltage across the plates, so it is considered discharged. When the **rcTime** method is called, the **portPin** that was output-true changes to input. The **rcTime** method then records the time it takes for the voltage at the capacitor's lower plate to cross the I/O pin's 2.5 V (true-false) logic threshold.

Using **rcTime** is very straightforward, except for one detail: For a given R and C, what value will **rcTime** return? It's easy to figure, based on a value called the RC time constant or tau (τ) for short. You can compute tau by simply multiplying R (in ohms) by C (in farads):

$\tau = R \times C$

The general RC time formula tells you the time required for an RC circuit to change from one voltage to another:

time = $\tau(\ln(V_{begin}/V_{end}))$

In this formula ln is the natural logarithm; it's a key labeled ln on most scientific calculators. V_{begin} is the starting voltage, while V_{end} is the ending voltage. Assume you're interested in a 10 kΩ resistor and 1.0 µF cap. Calculate τ:

$\tau = (10 \times 10^{3}) \times (1.0 \times 10^{-6}) = 10 \times 10^{-3}$

The RC time constant is 10×10^{-3} or 10 milliseconds. Now calculate the time required for this RC circuit to go from 5 V to 2.5 V.

$10 \times 10^{-3} \times \ln(5/2.5) = 6.93 \times 10^{-3}$

In **`RCTime`** units of 8.68 µs, that time (6.93×10^{-3}) works out to 798.4 ≈ 800 units. Since V_{begin} and V_{end} don't change, we can use a simplified rule of thumb to estimate **`rcTime`** results for circuits like the one in Figure 9.6 (a) and (b):

`rcTime` units ≈ 800 × R (in kΩ) x C (in µF)

Another handy rule of thumb can help you calculate how long to charge or discharge the capacitor before **`rcTime`**. A given RC charges or discharges 98 percent of the way in 4 time constants (4 x R x C). In Figure 9.6, the charge/discharge current passes through the 220 ohm series resistor and the capacitor. So if the capacitor were 1.0 µF, the minimum charge/discharge time should be:

Charge time = $4 \times 220 \times (1.0 \times 10^{-6}) = 880 \times 10^{-6}$

It takes only 880 µs (about 1 ms) for the capacitor to charge/discharge. In practice, you could set the pin true and delay for 1 ms to be safe. Here is a code snipped that would work for the rcTime circuit from Figure 9.6.

A couple of final notes about Figure 9.6. You may be wondering why the 220 ohm resistor is necessary at all. Consider what would happen if resistor R were a potentiometer adjusted to 0 Ω. When the I/O pin went high to discharge the cap, it would see a short direct to ground. The 220 Ω series resistor limits limit the short circuit current to 5 V/220 Ω = 23 mA and protect the Javelin Stamp from damage.

The 220 Ω resistor also forms a voltage divider with the 10 kΩ resistor that prevents the voltage from ever getting to 5 V. The formula for calculating the voltage divider created by the two resistors in Figure 9.6 (a) is:

$$V_{divider} = V_{DD} \frac{R_2}{R_1 + R_2}$$

Given values of R_1 = 220, R_2 = 10 k, and VDD the true value of V_{begin} works out to:

$$V_{begin} = V_{divider} = 5\_V \frac{10{,}000}{220 + 10{,}000} = 4.89\_V$$

The **return** value and the **timeout** parameter are actually unsigned integers. See the above discussion for **pulseIn** for more details about using unsigned parameters.

readPin

static boolean readPin(int portPin) – Use **readPin** to determine the state of an input pin. The pin is forced to an input if it isn't already set to input. To use **readPin** use a port number like **CPU.pin0**. If you want to access the pin by its number, use the **pins** array (as in **CPU.pins[2]**, for pin 2, for example).

Example:

Let's assume that you have already declared a boolean variable called **valueP5** using:

```
boolean valueP5;
```

You can use readPin to load the logic value seen by P5 into the **int valueP5** using the command:

```
valueP5 = CPU.readPin(CPU.pin5);
```

You can also display the value of P5 in the Messages from Javelin window using:

```
System.out.println(CPU.readPin(CPU.pin5));
```

readPort

public static byte readPort(int port) - Read the value on a port. Read the value currently on a port.

Parameters:
port - the port to read. Can be either P0 through P7, which is **CPU.PORTA**, or P8-P15, which is **CPU.PORTB**.

Returns:
The value on the port. This value is a binary number that corresponds to the values seen at the I/O pins.

Examples:

If you have an 8-bit parallel device connected to **PORTB** (P8-P15), you can load the data from the parallel device using **readPort**. Let's say you declared a variable named **parallelData** earlier in the program, and you want to load the 8-bit value transmitted by the parallel device into this variable, simply use:

```
int parallelData;
parallelData = CPU.readPort(CPU.PORTA);
```

Let's say that you have a four bit parallel device that is sending data to P4-P7. You can read the value by using **readPort**. Let's also say that you want to load this value into the lower four bits of an **int** variable named **nibble** that you declared earlier in the program.

```
int nibble;
nibble = CPU.readPort(CPU.PORTA);
```

Your **nibble** variable now contains the entire contents of **PORTA**. You can use the **>>** operator to shift the contents of **nibble** four binary slots to the right, for example:

```
nibble = CPU.readPort(CPU.PORTA) >> 4;
```

You can also use the **&** operator to mask data. For example, if you want the 4 bits to stay in bits 4-7 of your nibbleP4_7 variable, you can do this:

```
nibble = CPU.readPort(CPU.PORTA) & 0x00F0;
```

Keep in mind that if you do not apply the mask (or shift the data to the right by four bits), **nibble** will contain the data read from I/O pins P0 through P3 along with the data you want.

removeVP

static void removeVP(VirtualPeripheral vp) – Virtual peripherals typically install themselves when you instantiate them. You can use **removeVP** to unload a currently executing VP. You can have up to six VPs running at any given time, but your program may make use of more than six VPs. You can use this method to remove a VP that you want to halt. You can then load a different VP (or re-load the same one) using the **installVP** method.

The VP halts immediately when using **removeVP**. This could cause a **Uart** object to stop sending mid-byte or a **PWM** object to halt in the middle of the true part of its signal when you wanted the **PWM** signal to go to a **false** resting state. Keep this in mind when writing your code. For example, you can use the **Uart.byteAvailabe** method to make sure the Uart's buffer is empty before removing the VP. Likewise, you can use **CPU.writePin** to make sure the I/O pin is **false** before moving on.

Example:

Let's say you created a **PWM** object named **pwm**, and that you have all six VP slots in use. You can remove the **pwm** VP to make room for another VP using:

```
// Earlier in the program, the pwm object is loaded.
PWM pwm = new PWM(CPU.pin9, 100, 200);

// Later in the program it is removed to make room for a different VP
CPU.removeVP(pwm);
```

See the **installVP** method for information on adding a VP.

setInput

public static void setInput(int portPin) - Make a pin an input. The specified pin will be converted to an input. More than one pin can be specified using the **+** operator, as long as all pins are on the same port (P0 through P7 or P8 through P15).
Parameters:
portPin - the pin to make into an input.

Note, if you want to change an I/O pin back to an output, use **CPU.writePin**.

Examples:

Lets' say you want to change P15 from output to input, just use:

```
CPU.setInput(CPU.pin15);
```

You can toggle more than one I/O pin from output to input using the + operator. For example, if you want to change P5, P6, P7, P8, and P9 from output to input, keep in mind that you are dealing with two different ports. Use two separate commands:

```
CPU.setInput(CPU.pin5 + CPU.pin6 + CPU.pin7);
CPU.setInput(CPU.pin8 + CPU.pin9);
```

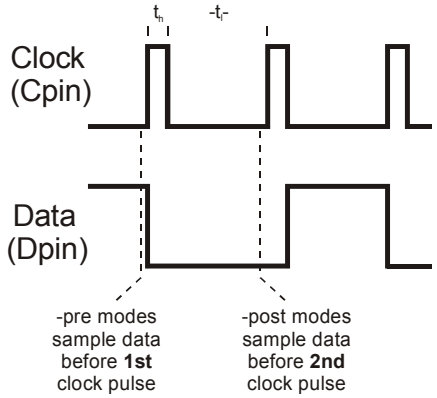
shiftIn

public static int shiftIn(int dataPortPin, int clockPortPin, int bitCount, int mode)

The **shiftin** method is used to read data from a synchronous serial device. The **clockPortPin** is used to send clock pulses to the synchronous serial device. The **dataPortPin** is used to read the binary output values sent by the synchronous serial device in response to each clock pulse. If the **dataPortPin** is set to output before the method is called, **shiftIn** will change the **dataPortPin** to input and leave it that way.

Likewise, if the **clockPortPin** is set to input before the method is called, **shiftIn** changes the **clockPortPin** to output and leaves it that way. The initial output state of **clockPortPin** will determine the polarity of the pulses delivered. If the **clockPortPin** is set to **false** before the method is called, the pulses will be positive (false – true – false). Occasionally, you will find a synchronous serial device that requires negative pulses. You can deliver negative pulses by setting the **clockPortPin** to **true** before calling the **shiftOut** method. Then the clock pulses will be negative (true – false – true). The **mode** parameter selects pre/post clock sampling as shown in Figure 9.7. The figure also shows the pulse durations, which are t_h = 8.68 µs and t_l = 17.36 µs.

Figure 9.7 shiftIn PRE/POST_CLOCK_LSB/MSB



Post clock sampling makes more sense at first glance since the pulse is applied, then the data value is checked. On the other hand, when pre-clock sampling is used, the **dataPortPin** is sampled to obtain the first data bit before delivering any pulses on **clockPortPin**. The most common reason that a data bit is already available is because the last pulse from a previous **shiftOut** call caused the first data bit to appear at the synchronous serial device's output. Keep this in mind if you are using a device that requires a write (**shiftOut**) prior to a read (**shiftIn**). In some cases, the device must be read without first being written to. You can apply an extra pulse simply by modifying the **bitCount** to request an extra bit, then use post clock sampling. Note that the **bitCount** can be set to higher than 16 (up to 256) if desired. When a number greater than 16 is used, only the last 16-bits will be returned.

Parameters:
dataPortPin - I/O pin that reads the data sent by the peripheral device.
clockPortPin - I/O pin that delivers clock pulses to the external device.
bitCount - the number of binary values to be shifted in (from 1 to 16). **bitCount** must not be 0.
mode - the shifting and clocking mode to employ. This parameter is used to tell the Javelin Stamp whether the binary values read by the **dataPortPin** are sent in ascending order starting with the least significant bit (LSb-first) or in descending order starting with the most significant bit (MSb-first). The clocking **mode** also

determines whether the data is sampled before or after the first clock transition. The **mode** arguments are summarized in Table 9.1.

Table 9.1: shiftIn Mode Arguments

Field	Mode Function
CPU.POST_CLOCK_LSB	Post-clock sampling and LSB-first transmission
CPU.POST_CLOCK_MSB	Post-clock sampling and MSB-first transmission
CPU.PRE_CLOCK_LSB	Pre-clock sampling and LSB-first transmission
CPU.PRE_CLOCK_MSB	Pre-clock sampling and MSB-first transmission

Returns:
This method returns an int value that contains the bits that were received by the data pin. The result contains **bitCount** bits and is justified according to the mode setting. If MSb-first (**CPU.POST_CLOCK_MSB** or **CPU.PRE_CLOCK_MSB**) is chosen, the data will be shifted left into the least significant bit, and the data bits will be right justified. For example, if four bits (the binary value 1011) are shifted in, they will be loaded into an int value as: 0000000000001011.

If LSb-first (**CPU.POST_CLOCK_LSB** or **CPU.PRE_CLOCK_LSB**) is chosen, the data will be shifted right into the most significant bit, and the data will be left justified. Let's take a look at shifting in the same four bits (binary 1011) again. This time, your `int` value would contain: 1011000000000000. Instead of a small positive number, you would end up with a large negative number. The `>>` operator is used to shift the contents of the `int` value twelve more bits to the right using `>>12` to get the LSb where it should be in the rightmost bit. The final result is then 0000000000001011.

Examples:
Here are a few code snippets that demonstrate how to use the **shiftIn** method. For executable code examples, see Communicating with Peripheral ICs section of Chapter 4. The first statement uses **writePin** to set the I/O pin delivering the clock signal to **false**. This causes the **shiftIn** method to deliver positive pulses.

```
// Initialize clock pin
CPU.writePin(CPU.pin5, false);
```

The **shiftIn** method call shown below uses the **CPU.PRE_CLOCK_MSB** mode. The data is shifted in MSb-first, and it is already "right justified". This means the data does not have to be shifted any further to correct the value even though only 8-bits were shifted into value.

```
int value;
//…
// Shift in data
value = CPU.shiftIn(CPU.pin6, CPU.pin5, 8, CPU.PRE_CLOCK_MSB );
```

This next **shiftIn** call shifts in a 9-bit value from a DS1620. Remember, if you are shifting a value that's less than 16 bits and storing it in a variable using either **CPU.POST_CLOCK_LSB** or **CPU.PRE_CLOCK_LSB**, you have to use the **>>** operator to "right justify" the value. Before storing the value in the **tempIn** field, the value was shifted right by another 7-bits using the **>>** operator.

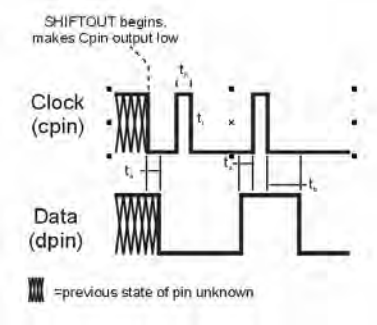
```
int tempIn;
// …
// Shift in 9-bits then shift right by another 7-bits
tempIn = (CPU.shiftIn(CPU.pin6,CPU.pin5,9,CPU.POST_CLOCK_LSB) >> 7);
```

shiftOut

```
public static void shiftOut(int dataPortPin, int clockPortPin, int
bitCount, int mode, int data)
```

The **shiftOut** method sends data to a synchronous serial device by making a data value available on the **dataPortPin** then applying a clock pulse on the **clockPortPin** as shown in Figure 9.8. The durations shown in the figure are $t_h = t_a = t_b = 8.68$ µs, and $t_l = 17.36$ µs. The number of data values sent is determined by **bitCount**. The mode determines whether the most significant bit (MSb) or the least significant bit (LSB) is sent first. The **data** parameter is the **int** field that contains the value to be transmitted to the synchronous serial device.

Figure 9.8 shiftOut
PRE/POST_CLOCK_LSB/MSB



Both the **dataPortPin** and **clockPortPin** are changed to outputs when the **shiftOut** method is called, and they remain outputs after the method is executed. The initial output state of **clockPortPin** determines the polarity of the pulses delivered. If the **clockPortPin** is set to **false** before the method is called, the pulses will be positive (false – true – false). Occasionally, you will find a synchronous serial device that requires negative pulses. You can deliver negative pulses by setting the **clockPortPin** to **true** before calling the **shiftOut** method. Then the clock pulses will be negative (true – false – true).

Note: If you are shifting out less than 16-bits using **CPU.SHIFT_MSB**, make sure to shift your valid data to the left using the **<<** operator. See the examples below.

Parameters:
dataPortPin - I/O pin that sends the **data** to the peripheral device.
clockPortPin - I/O pin that delivers clock pulses to the peripheral device.
bitCount - the number of binary values to be shifted in (from 1 to 16). **bitCount** must not be 0.
mode - the shifting mode to employ. The mode specifies whether to shift the data MSb first using **CPU.SHIFT_MSB** or LSb first using **CPU.SHIFT_LSB**.

Examples:
The code snippets below demonstrate using the **shiftOut** method. For executable code examples, see Communicating with Peripheral ICs section of Chapter 4.

Before calling **shiftIn**, remember to initialize the value of the **clockPortPin** using **CPU.writePin**, for example:

```
// Initialize P1 for delivering positive pulses.
CPU.writePin(CPU.pin1, false);
```

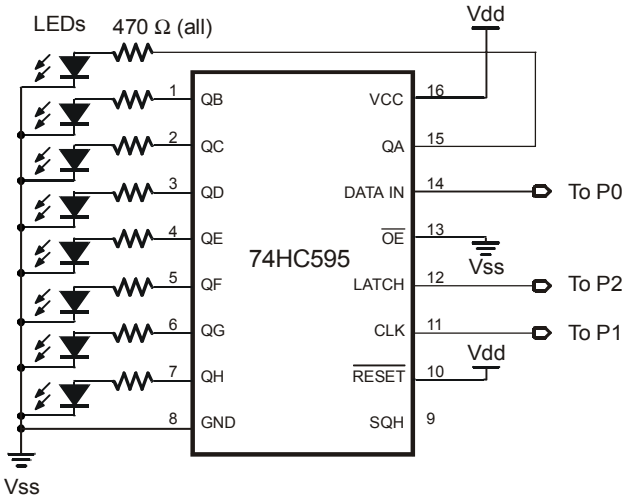
Let's say the **int** value number stores the value 254. In terms of an **int** field, this is really the decimal number +00254, which will appear as a hexadecimal value 0x00FE if viewed with the Debugger. In binary, this value is: 0000000011111110. If you want to shift LSb-first using **CPU.SHIFT_LSB**, the data is ready to be justified to the right, so simply use the **shiftOut** method.

```
int number = 254;
CPU.shiftOut(CPU.pin0,CPU.pin1,8,CPU.SHIFT_LSB,number);
```

If you are shifting out MSb-first using **CPU.SHIFT_MSB**, the value needs to be first shifted left by 8-bits using the **<<** operator. Otherwise, all that will get shifted out are the leading zeros. After the **<<** shift, your binary value will be 1111111000000000. Now the correct values will be shifted out when **shiftOut** is called.

```
CPU.shiftOut(CPU.pin0, CPU.pin1, 8, CPU.SHIFT_MSB, number <<8 );
```

Figure 9.9 shiftOut example using the 74HC595



Program Listing 9.4 - Using shiftOut on 75xx595 shift register

```
import stamp.core.*;

public class Shift74595 {

  final static char   HOME      = 0x01;
  final static int    DATA_PIN  = CPU.pin0;
  final static int    CLOCK_PIN = CPU.pin1;
  final static int    PULSE_595 = CPU.pin2;

  static void write595(int number){
    CPU.shiftOut(DATA_PIN, CLOCK_PIN, 8, CPU.SHIFT_MSB, number <<8 );
    CPU.pulseOut(5,PULSE_595);
  }                                                    // end write595

  public static void main() {

    CPU.writePin(PULSE_595,false);
    CPU.writePin(CLOCK_PIN,false);

    while(true){

      for (int i = 0; i <= 255; i++){
        write595(i);
        CPU.delay(2500);
      }                                                // end for
    }                                                  // end while
  }                                                    // end main
}                                                      // end class
```

writePin

`public static void writePin(int portPin, boolean value)` - Set a pin to a logic state. Makes the selected pin high or low as selected by the value parameter. If the pin is an input then it will be changed to be an output and remain as an output when complete.

Parameters:
portPin - the I/O pin to write.
value - the state the set the pin to.

Examples:
You can change the state of one pin or of multiple pins on the same port (P0-P7 or P8-P15).

```
// Set P5 to output-true.
CPU.writePin(CPU.pin5, true);

//Set P11 and P15 to output-false.
CPU.writePin(CPU.pin11 + CPU.pin15, false);
```

writePort

`public static void writePort(int port, byte value)` - Output a value onto a port. The lower 8 bits of value will be written to the port. Pins on the port will not be converted to outputs first. This method does not affect the direction of the port. It will disturb any virtual peripherals which are using the port.

You can use this method to change the output values of groups of I/O pins on a given port. Since this does not change the direction of an I/O pin, I/O pins that are inputs remain inputs. You can use this method to write values to PORTA (I/O pins P0 through P7) or PORTB (I/O pins P8-P15).

Parameters:
`port` - the port to control. Can be either **`CPU.PORTA`**, or **`CPU.PORTB`**.
`value` - the value to write to the port (**`true`** or **`false`**).

Examples:
Let's say that P4 and P5 are inputs, and the rest of the I/O pins on **`CPU.PORTA`** are outputs (P0-P3 and P6-P7). If you want to set all the I/O pins that are already outputs to true regardless of their current state, you can use this command:

```
CPU.writePort(CPU.PORTA, (byte)0xFF);
```

Since **`0xFF`** is binary 11111111, it writes binary 1s to all **`CPU.PORTA`**. This sets all the I/O pins that are already outputs to **`true`**. Since writePort does not make any changes to the I/O pin's, direction, the I/O pins that were inputs, are still inputs.

Let's say that now you want to change the output values of P0 through P3 to **`false`**. But leave the upper I/O pins true. You can do this using 0xF0, which corresponds to binary 11110000.

```
CPU.writePort(CPU.PORTA, (byte) 0xF0);
```

DAC

The **DAC** class is a VP that performs single bit digital to analog conversion with the aid of an external resistor and capacitor (see Figure 9.10). The DAC generates a train of pulses that the resistor and capacitor integrates into a constant voltage. The values or R and C depend on the application. The example code works fine with values of:

- R = 1 kΩ
- C = 1 µF

When R × C is larger, the voltage will be steadier, but it will respond less quickly when you want to change it. When R × C is smaller, the output voltage will respond more quickly but you will see more fluctuations in the analog voltage because of the pulses that are delivered to maintain that voltage. In general, it's better to place an op-amp in between the analog output if you plan on driving a load that has any appreciable current draw. If the current draw is small, yet appears to have an effect, use a larger value of C, which will store more electrons to supply the small current draw.

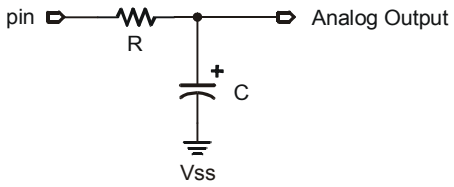


Figure 9.10
Circuit for use with DAC object

DAC is really s special case of PWM. DAC is designed to allow the Javelin Stamp (a purely digital device) to generate an analog voltage. The basic idea is this: If you make a pin output high, the voltage at that pin will be close to 5 V. Output low is close to 0 V. What if you switched the pin rapidly between high and low, so that it was high half the time and low half the time? The average voltage over time would be halfway between 0 and 5—2.5 V. This is the idea behind **DAC**; that you can produce an analog voltage by outputting a stream of digital 1s and 0s in a particular proportion. The proportion of 1s to 0s in **DAC** is called the duty cycle. The duty cycle controls the analog voltage in a very direct way. The higher the duty cycle the higher the voltage. In the case of the Javelin Stamp, the duty cycle is the ratio between the high time and the low time. To determine the proportional **PWM** output voltage, use this formula: (highTime/(lowTime+highTime)) * 5 V. For example, if **highTime** is 49 and **lowTime** is 13, the duty cycle is 0.79. The output, then, is .79*5 V = 3.95 V.

Base Class: **VirtualPeripheral**

Constructors
DAC(int pin) – Creates a **DAC** object that uses the specified **pin.**

Methods
`void update(int value)` – Sets the **`DAC`**'s output. The **`value`** argument must be between 0 and 255.

Example: See the Digital to Analog Conversion section of Chapter 4.

Once you set the **`DAC`** it continues to output the requested voltage until you change it or until you stop or remove the VP. The example below cycles the output voltage from 0 to 5V in a steady ramp.

EEPROM

The Javelin Stamp contains an EEPROM onboard that stores your program. You may use unused portions of EEPROM to write data that persists even when the Javelin Stamp loses power. Like all EEPROMs, there is a limit to how many times you can write to the EEPROM before it will fail (usually in the neighborhood of 1 million cycles).

Although 1 million writes sounds like a lot, you should be careful when writing programs that take advantage of EEPROM. For example, if you wrote a data logging program that wrote to the same EEPROM cell every second, you'd write 1 million times in less than 12 days. Even at once a minute, you could wear out the EEPROM in less than 2 years.

EEPROM storage is best for configuration options and other data that does not frequently change.

The Javelin Stamp organizes its EEPROM so that address 0 is always in the same place regardless of the program you have loaded. This allows you to load the EEPROM with values using one program and retrieve them with another program you load later. Of course, when you share EEPROM like this, the largest program determines the maximum address you can use.

Base Class: *Object*

Methods:
`static byte read(int address)` – The **`read`** method reads a single byte from the specified address. You may use addresses ranging from 0 to the amount returned by *size* –1. If you specify an invalid address, the method will throw an **`IndexOutOfBoundsException`**.

`static int size()` – Use the **`size`** method to determine how much EEPROM is available. You may use addresses from 0 up to, but not including the value of **`size`**.

`static void write(int address, byte value)` – This method writes the indicated byte to the specified address. If you specify an invalid address, the method will throw an **`IndexOutOfBoundsException`**.

Example:

Program Listing 9.5 - EEPROM Test

```
import stamp.core.*;

public class EETest {

  static void setEEProm(int n) {
    // have to chop n into bytes
    EEPROM.write(0,(byte)(n&0xFF));
    EEPROM.write(1,(byte)(n>>8));
  }

  static int getEEProm() {
    int x;
    x=EEPROM.read(1);
    x=(x<<8)+EEPROM.read(0);
    return x;
  }

  public static void main() {
    setEEProm(2300);
    System.out.println("Bytes available in EEPROM:");
    System.out.println(EEPROM.size());
    System.out.println("The value you wrote to EEPROM:");
    System.out.println(getEEProm());
  }                                                    // end main
}                                                      // end class
```

Memory

Base Class: Object

Methods: **`static int freeMemory()`**

Returns: The number of bytes available in the SRAM.

Example:

```
int mem;
mem = Memory.freeMemory();
System.out.println(mem);
```

PWM

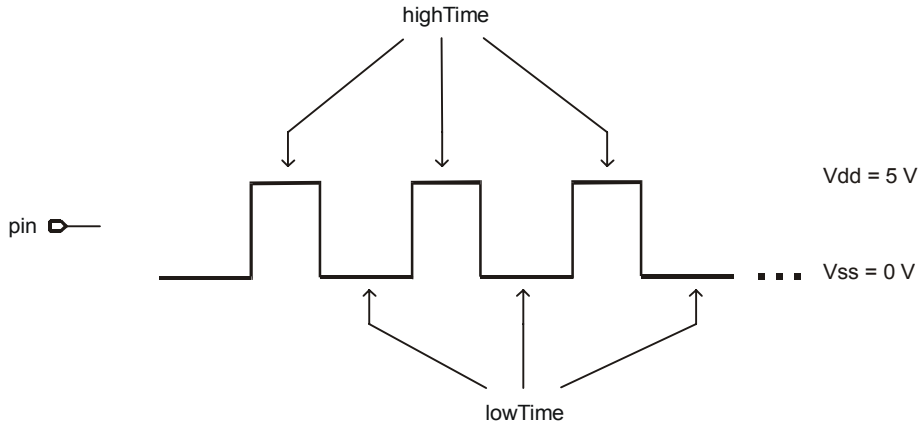
The **PWM** class is a Virtual Peripheral (VP) that can generate a train of pulses. You can use these pulses to control motor speeds, LED or lamp brightness, or – with the addition of a simple filter circuit – generate analog voltages. You can think of **PWM** as a programmable pulse generator. By setting the ratio of time the pin is on to the time the pin is off, you can control the average power sent to a device.

Since **PWM** is a VP, once you set a channel to output **PWM** (short for pulse width modulation) it will continue to do so until you stop it. You specify the amount of time you want the PWM generator to output a high pulse and the amount of time you want the pin to be low. The time units are 8.68uS. To set the duty cycle to 50%, for example, you'd set the high time and low time to be 1 (or any two equal integers).

The **PWM** will accept integer values from 0 to 65535. The Javelin's int field can hold values from –32768 to 32767. To enter **PWM** values above 32767 use the following map:

PWM value:	0	1	2	…	32767	32768	32769	32770	…	65533	65534	65535
Integer value:	0	1	2	…	32767	-32768	-32767	-32765	…	-3	-2	-1

Figure 9.11 Pulse train generated by PWM object



Base Class: **VirtualPeripheral**

Constructors:

PWM(int pin, int highTime, int lowTime) – Creates a **PWM** object on the specified pin (for example, **CPU.pin1**) with the specified high and low times (in 8.68us units).

Methods:
`void update(int highTime, int lowTime)` – Changes the pulse widths associated with the **`PWM`** object.

Terminal

When debugging a Javelin Stamp program, you may want to supply input to your program using the debugging terminal. You can do this with the **`Terminal`** class, which allows you to read keystrokes from the debug terminal. You can also determine if any characters are waiting to be read. The **`Terminal`** class contains **`static`** members. There is no need to instantiate a copy of **`Terminal`**.

Base Class: **`Object`**

Methods
`static boolean byteAvailable()` – Returns **`true`** if one or more characters are available to be read.

`static char getChar()` – Returns a character from the terminal. If no character is available, this call waits until there is something to return.

Example
The example program reads characters from the **`Terminal`** until the user presses Enter. As the user types, the program compares the input with the **`password`** string. At the end, if the characters match, the program prints a welcome message. Otherwise, it prints an access denied message.

You can send the Javelin Stamp messages by typing them into the field below the area where the messages are displayed in the messages from Javelin window.

Program Listing 9.6 - Password Gate

```
import stamp.core.*;

public class PasswordGate {

  static String password = "Parallax";

  public static void main() {
    int i=0;
    int c;                                          // character
    boolean access=false;
    System.out.println("Enter password: ");
    do {
      c=Terminal.getChar();
      if (c==13) {
        access=(i==password.length());
        break;
      }                                             // end if
      if (i==-1) continue;                          // already blew it!
      if (password.charAt(i)==c)
```

```
        i++;
      else
        i=-1;
    } while (c!=13);
    if (access)
      System.out.println("Welcome!");
    else
      System.out.println("Unauthorized access forbidden!");
  }                                                    // end main
}                                                      // end class declaration
```

Timer

The **Timer** virtual peripheral (VP) provides you with a 32-bit timer with 8.68us resolution. The first time you create a **Timer** object, the class installs itself as a virtual peripheral. However, any subsequent timers use the same VP.

Each timer has a mark that can store the current time (by calling the **mark** method). Once marked, you can test to see if any number of milliseconds (or timer ticks) has passed since the **mark** call. You can also simply read the tick values if you like.

Base Class: **VirtualPeripheral**

Methods:
void mark() – Use the **mark** method to remember the current time. You can later test to see if a given number of milliseconds have elapsed (or you can test timer ticks). See **timeout** for more information.

static void start() – This method starts the master timer VP. The timer automatically starts when you first create it, so you won't need to use this method unless you've previously called **stop**.

static void stop() – This method stops the master timer VP. Notice that calling **stop** will make any previous **mark** calls to any timer inaccurate.

int tickHi(), **int tickLo()** – These two methods return the high-order 16 bits and low-order 16 bits of the current timer value. ntegers are signed, values greater than 32767 will appear negative.

boolean timeout(int milliseconds), boolean timeout(int hi, int lo) – The **timeout** method returns **true** if the specified period has elapsed since the last call to *mark*. The single integer argument specifies the number of milliseconds. If you use two arguments, they are the high and low parts of the period in 8.68 us units. The method will return **true** if the current time is the same or greater than the marked time plus the period. Otherwise, the method returns **false**.

Example:

Program Listing 9.7 - Timer Example

```
import stamp.core.*;

public class TickTock {

  static boolean tick=true;

  public static void main() {
    Timer clock = new Timer();
    while (true) {
      for (clock.mark(); !clock.timeout(1000);) ;     // wait for one second
        System.out.println(tick?"tick":"tock");
      tick=!tick;
    }                                                 // end while
  }                                                   // end main
}                                                     // end class
```

Uart

The **Uart** object is a Virtual Peripheral that can act as a serial receiver or transmitter. Using the **Uart** does not cause your program to stop, so you can receive or transmit serial data while your program continues to execute. If you want to send and receive, no problem, just use two **Uart** objects, one for each communication channel. Each **Uart** object has a 256 byte buffer for receive or transmit operations. You can specify a handshaking pin to send a stop signal to the sender when the buffer has 16 bytes or fewer remaining.

Base Class:
VirtualPeripheral

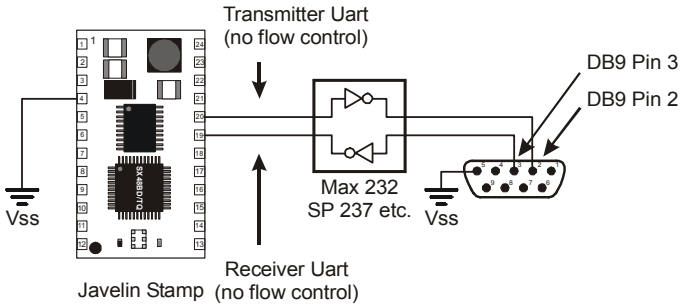
Fields:
final static int dirReceive – Set the **Uart** to receive mode (see Constructors).
final static int dirTransmit – Set the **Uart** to transmit mode (see Constructors).

These constants allow you to easily set the baud rate to any standard baud rate:
final static int speed2400
final static int speed4800
final static int speed7200
final static int speed9600
final static int speed14400
final static int speed19200
final static int speed38400
final static int speed57600

final static boolean dontInvert – selects non-inverted mode. Non-inverted mode allows you to connect the Javelin Stamp to an RS232 device. To do this you will need to boost the Javelin's TTL signal (0/5

V) to ±12 V as required by the RS232 specifications. This can be accomplished by using a MAX232 or the SP237 Uart transceivers. Either of these transceivers will invert the TTL signal as it boosts them to ±12 V, which is why we use the non-inverted mode. The Javelin Stamp Demo board has an SP237 that you can use by connecting the I/O pins of the Javelin Stamp to the 8-socket COM header (X4) on the Demo Board (See Figure 4.8b). The figure below shows you this as a 2-wire connection, without flow control.

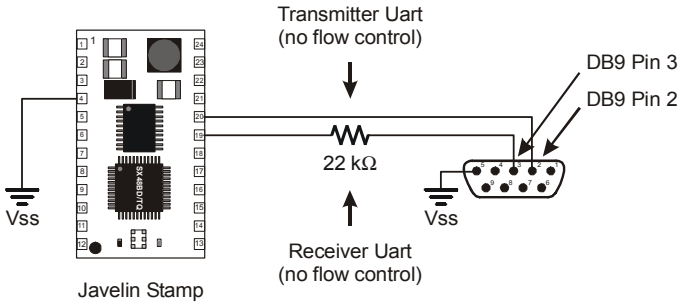
Figure 9.12
UART Not-Inverted



Important	The pins on the male/female DB-9 connectors are different. They are mirror images of each other; care is needed when making these connections that you are connecting to the appropriate pin.

`final static boolean invert` – selects inverted mode. Inverted mode allows you to connect to a computer's RS232 port without using a MAX232 or an SP237 RS232 Uart transceiver. This can be accomplished by using a 22 k resistor to connect the Javelin Stamp I/O pin (that you are using as the receiver) to pin #3 on a 9-pin serial port (DB-9). This will allow you to receive data. To send data to a PC, connect a Javelin Stamp I/O pin to pin #2 of the serial port on your computer. This method will create a voltage that is not to the Uart specifications; some receivers do not accept this nonstandard voltage. If this is your situation you will need to use a Uart transceiver in non-inverted mode. The figure below shows you this as a 2-wire connection, without flow control.

Figure 9.13
UART Inverted



```
final static int stop1
final static int stop2
final static int stop3
final static int stop4
final static int stop5
final static int stop6
final static int stop7
final static int stop8
```

These constants allow you to select the number of stop bits the **Uart** object will expect. Remember, stop bits are not true bits, but rather idle line time. Most devices require 1 stop bit as a minimum and some older devices expect 2 stop bits. Adding extra stop bits increases the amount of time between characters. The exact amount of time depends on the baud rate. For example, at 9600 baud, each bit is about 1.04 ms long. So four stop bits would create 4.16 ms between each character. If the device you are connecting to expects a certain number of stop bits, you'll need to specify at least that many stop bits in the **Uart** constructor. Specifying more will still work, but will slow communications.

TIP	For devices that need pacing, you can get breaks longer than eight stop bits by using the **Timer** VP in conjunction with the **sendByte** method. You can also use **sendByte** in conjunction with **CPU.delay()**.

Constructors:

```
Uart (int direction, int dataPin, boolean dataInvert, int baudRate, int stopBits)
```

The constructor is used to create and start a **Uart** object. Many of the parameters to the constructor should be from the list of the **final static** fields just discussed. The **direction** parameter specifies if the **Uart** is a transmitter (**Uart.dirTransmit**) or a receiver (**Uart.dirReceive**). The **dataPin** parameter specifies the pin you wish to use for serial communications (for example, **CPU.pin2**). For inverted mode, set **dataInvert** to **Uart.invert**, for non-inverted mode, use **Uart.dontInvert**. Finally select a **baudRate** and **stopBits** constant (for example, **Uart.speed9600** and **Uart.stop1**).

Uart(int direction, int dataPin, boolean dataInvert, int hsPin, boolean hsInvert, int baudRate, int stopBits) – This constructor takes the same arguments as the first constructor (see above) but it also allows you to select a handshaking pin (e.g., **CPU.pin3**) and the polarity for that pin. If the **hsInvert** parameter is **Uart.dontInvert** and if the **Uart** object is a receiver, the pin will be **true** if there are at least 16 bytes available in the buffer. Otherwise, the flow control pin will be **false**. If the **Uart** object is a transmitter, using **Uart.dontInvert**, the **Uart** object will transmit serial data when the receiver sends a **false** signal on the handshaking pin. When you use **Uart.Invert**, it expects a **true** signal on the handshaking pin.

Methods:
`void start()` – starts the **`Uart`** virtual peripheral. By default, the **`Uart`** starts when you create it, so you'll only need to use **`start`** after you've stopped it.

`void stop()` – stops the **`Uart`** virtual peripheral. This method stops the virtual peripheral immediately. It does not check to see if there are any bytes remaining in the buffer or if the Uart is in the middle of transmitting or receiving a byte.

TIP	Use **`byteAvailable()`** to find out if there is a byte in the buffer before calling the **`stop()`** method.

`void sendByte(int data)` – stores a byte in the outgoing buffer for transmission as soon as possible.

`void sendString(String data)` – stores a string in the outgoing buffer for transmission as soon as possible.

`int receiveByte()` – reads the next byte from the serial input buffer. If no bytes are present in the buffer, this call will wait until a character arrives.

Caution	If the buffer is empty, this method blocks your foreground code until a byte is received. You can use **`byteAvailable()`** to be sure there is a byte in the buffer before calling **`receiveByte()`**. If a byte is not available, the code can do something else and check to see if there is a byte available later.

`boolean byteAvailable()` – returns **`true`** if there are characters available for **`receiveByte()`** to return.
Listing 9.5

Example:

See the following sections in Chapter 4 for Examples:

- Communicating with Computers
- Communicating with Peripheral Devices.

Summary of Java Differences

This section explains the differences (listed below) between workstation Java and the subset of Java used by the Javelin Stamp. Recommendations for how to approach writing code for the Javelin Stamp are made for each difference.

- Single Thread
- No Garbage Collection
- Subset of Primative Data Types
- Subset of Java Libraries
- Strings are ASCII
- No Interfaces
- One Dimensional Arrays

Single Thread

The Javelin Stamp only supports one thread. However, you can schedule multiple tasks to execute on the same thread with the Timer object. You can also run up to six background VP objects. See Chapter 7 for a **`Timer`** example. Background VP objects are first introduced in the Javelin Stamp Features section of Chapter 1, and examples making use of these VPs can be found in Chapters 4 and 9.

No Garbage Collection

Once memory has been allocated for an object, that memory is never reclaimed for use with another object. For embedded systems, this is not usually a big limitation, especially since garbage collection can wreak havoc with real time system performance. In a PC based Java system, garbage collection may occur at any time. This can cause problems when you are trying to do processing in real time.

Most embedded applications involve allocating memory or buffers at the start of a program and not allocating more memory as the program progresses. When writing programs for the Javelin Stamp, make sure that your programs do not allocate memory every time an event occurs, or at regular intervals.

If you want to reuse memory, it is up to you to program that behavior. Even though the **`obj`** variable is a single variable, it will hold 100 different objects. Each object will persist for the life of the program. When possible reuse objects instead of creating new ones.

There are several strategies you can use to make sure your programs are efficient:

- Use **`static`** variables whenever possible.
- Use the **`StringBuffer`** object instead of **`String`** objects if you have significant amounts of text information that change frequently.

Avoid creating new object instances whenever possible. Here are a few ways that unwanted new objects are created that you should watch out for when writing applications for the Javelin Stamp:

- Creating new objects in loops or based on recurring events.
- Concatonating data, example: **`System.out.println(a + b)`**.
- Concatonating **`String`** objects: **`a = a + b;`**

Particularly, be wary of allocating memory within loops. For example:

```
// Avoid writing code that creates new objects in loops!

for (int i=0;i<100;i++) {
  SomeObject obj = new SomeObject();
      .
      .
      .
}
```

A similar problem can arise when you use string concatenation. If **`a`** and **`b`** are **`String`** object references, you might write:

```
a = a + b;
```

This creates a temporary **`StringBuffer`** (internally handled by the compiler) and it orphans the original contents of **`a`** (assuming there are no other references to that string). So now you have two objects taking up space that you can't possibly use.

Subset of Primitive Data Types

Table 10.1 lists the primitive data types supported by the Javelin Stamp. Note that the **`int`** type is 16 bits wide. Therefore, the largest signed value you can place in an **`int`** is 32,767 (0x7FFF). Values above 0x7FFF appear negative. The **`byte`** type is 8 bits, and the **`short`** type is 16 bits, just as in Java on your PC. With the **`byte`** data type, the values range from –128 to 127. If you need unsigned bytes, use **`char`**, which can range from 0 to 255.

Table 10.1: Primitive Data Types Supported by the Javelin Stamp

Type	Support
`boolean`	Yes
`byte`	Yes
`char`	Yes
`short`	Yes
`int`	16-bit instead of 32
`float`	No
`double`	No
`long`	No

The Javelin Stamp firmware does not support floating point types such as **float** and **double**. The **long** data type is also unsupported. If you need numbers larger than 16-bits, you may be able to build your own routines to handle the larger numbers. Of particular use is the **CPU.carry** method. This method allows you to read the overflow result after a 16-bit addition or subtraction. This makes it straightforward to add and subtract 32-bit (or larger) words. See the article on **CPU.carry** in Chapter 9 for an example.

The Javelin Stamp's **int** type spans –32,768 to 32,767, but there is a way around this if you need a positive value between than 32,767 and 65535 (0x7FFF to 0xFFFF). For example, if you need a delay of 6.4 seconds, you need to pass **CPU.delay** a value of 0xFA00. You can set a variable to 0x7D00 and shift it left:

```
int dly=0x7D00;
CPU.delay(dly<<1);
```

You can't simply use **0xFA00** or **0x7D00<<1** because the compiler will attempt to do the math at compile time and determine that the result is not a valid integer.

Subset of Java Libraries

A subset of the standard Java libraries is available to the Javelin Stamp. A PC Java installation provides a large number of libraries in the java.* packages. Many of these libraries are only applicable to workstation based programs. You should not assume that all of the same methods are available to your Javelin Stamp programs. Also, some methods that are available, they may have different behavior than you are used to because of data type differences, for example. In addition, the Javelin Stamp has many additional packages to provide support for various hardware and peripheral devices. These packages and how they are used are discussed in Chapters 8 and 9.

Strings are ASCII

Javelin Stamp Strings and characters are composed of single-byte ASCII characters, not double-byte Unicode characters. For embedded system programming, you'll usually prefer to handle ASCII characters. If necessary, you can store Unicode characters in an **int** variable.

No Interfaces

Interfaces are not available; however, you can create abstract classes that other classes can extend.

One Dimensional Arrays

The Javelin Stamp only supports single dimensioned arrays. You can always unwrap your multi-dimensional array into a single dimension. For example, suppose you have an 8 by 8 matrix that represents a checkerboard. You might write:

```
contents = checkboard[x*8+y];
```

You could even wrap the array in a method, to make the syntax clearer. For example:

```
int checkerboard(int x, int y) {
  return checkboard[x*8+y];
```

```
}

void checkerboard(int x, int y, int value) {
  checkboard[x*8+y]=value;
}
```

You can also create an array of arrays. The key to making this work, is to make the containing array (or arrays) contain **Objects**. Since all objects derive from **Object** and arrays are objects, you can store arrays in an **Object** reference.

Here's how you might implement the checkerboard using this strategy:

```
Object checkboard[] = new Object[8];
for (int i=0;i<checkboard.length;i++)
  checkboard[i] = new int[8];
```

Referencing a particular cell on the board is a bit cumbersome:

```
((int [])checkboard[x])[y]=1;
```

You can also decompose the array into individual rows:

```
int row2[] = (int [])checkboard[2];
row2[y]=0;
```

Understanding the Javelin Stamp's Memory Management

The Javelin Stamp stores your program in EEPROM. That means that your program remains in the Javelin Stamp even when the power is off. When you apply power to the Javelin Stamp, it copies this EEPROM to high-speed RAM memory for speedy execution.

The 32K of RAM also holds the stack and the heap which are both areas used to store your program's data and variables. When you create objects with **new** or define static variables, they consume space on the heap. You must carefully manage the heap since once you create an object on the heap, there is no way for the Javelin Stamp to reclaim it (until you reset or cycle the power, of course). Strings are specially treated and have their own heap. You can examine the size of the various memory areas using the Memory Usage tab of the debugger window (see Figure 10.1).

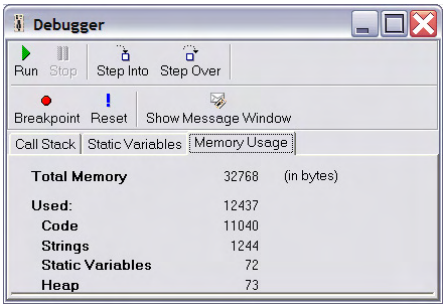


Figure 10.1
Javelin Stamp IDE Debugger memory usage page

The stack, on the other hand, is used for local variables and method parameters. The stack can grow and shrink at will, so there are fewer concerns with managing its size. However, you can't create objects on the stack – only simple variables and object references.

For example, consider this example:

```
void somemethod(String s) {
  StringBuffer work = new StringBuffer(s);
  . . .
}
```

Here, the variables **s** and **work** are on the stack. However, these variables are simply object references – not the objects themselves. The **new** operator creates a **StringBuffer** object on the heap and presumably, the actual **String** that **s** refers to is also on the heap.

A problem arises if you write a method like **somemethod** because once it returns to its caller the object in **work** is effectively lost. Your program no longer has a reference to this **StringBuffer** so it is effectively unusable. In a workstation Java, the system software would eventually notice that it was out of memory and reclaim this variable using a process called garbage collection. However, garbage collection happens at unpredictable times and may take a long time to complete. This makes it unsuitable for embedded programs where you need to know when things occur and how long they take to execute.

The problem isn't so much that the object is on the heap. Rather, it is that the object's only reference is on the stack. When the Javelin Stamp reclaims the **work** variable, the object it refers to is still on the heap, but you no longer have a way to access it.

The answer, then, is to store object references in fields that will exist for the life of your program. In many cases, these will be per instance fields or static fields in an object. There is one case where you might as well use stack variables to hold object references in your main program loop.

For example, consider this bit of code:

```
static void main() {
  StringBuffer buff = new StringBuffer();
  while (true) {
    . . .
    }
}
```

In this case, when **buff** goes out of scope, the program will stop running anyway. Therefore, there is no harm that you will lose access to the underlying **StringBuffer** object. The same principle applies anywhere you have code that you will effectively execute forever (or, at least, until your program terminates).

A common strategy for dealing with this problem is to create a pool of objects and keep them for the entire time your program is running. Then, different parts of your program can check out a few of these objects and return them to the pool when you are done with them. For more information on pools refer to the Javelin Stamp's IDE on-line documentation.

Another important consideration is where the compiler generates objects on your behalf. Since your program doesn't know about these objects, it is impossible for you to ever reclaim them. For example, consider this code:

```
int t=33;
System.out.println("The value is " + t);
```

For a simple example, or for debugging purposes, this might be acceptable. However, it is wasting memory. Why? Because the compiler is automatically generating a **StringBuffer** and a **String** object that it uses to build the concatenated string. Since your program doesn't directly work with these objects, they consume space in the heap that you can't recover. It would be better to create a **StringBuffer** as part of the object or in a static variable and then use it to synthesize the **String**.

```
StringBuffer buf=new StringBuffer(32);                // 32 byte string

void display() {
buf.append("The value is ");
buf.append(t);
System.out.println(buf.toString());
}
```

Memory and Variable Types

This information may come in handy if your application is running short on memory space:

- All types (including **char** and **byte**) use 2 bytes of memory.
- Arrays require the amount of space to store their elements (that is, 2 times the number of elements) plus 4 additional bytes.

- Exception: **byte** and **char** arrays use one byte per element.
- Local variables deduct from your stack and are reclaimed when they go out of scope.
- Objects require space for their non-static fields, plus two bytes of overhead.

10: Technical Details