

Javelin Stamp Programming

Introduction

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 Javelin Stamp and Its Features

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

The Javelin Stamp is somewhat of a departure from Parallax's BASIC Stamps. Most notably, the Javelin is programmed using a subset of the Java programming language. Some of the other features that set the Javelin apart from BASIC Stamps are:

- The instruction codes for the Javelin are fetched and executed from a parallel SRAM instead of a serial EEPROM.
- The Javelin has 32k of RAM/program memory with a flat architecture. No more program banks, and no more tight squeezes with variable space.
- The Javelin 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 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 can be placed directly into the circuit and debugged there.
- Delta-sigma 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

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.

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.

The Javelin'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.

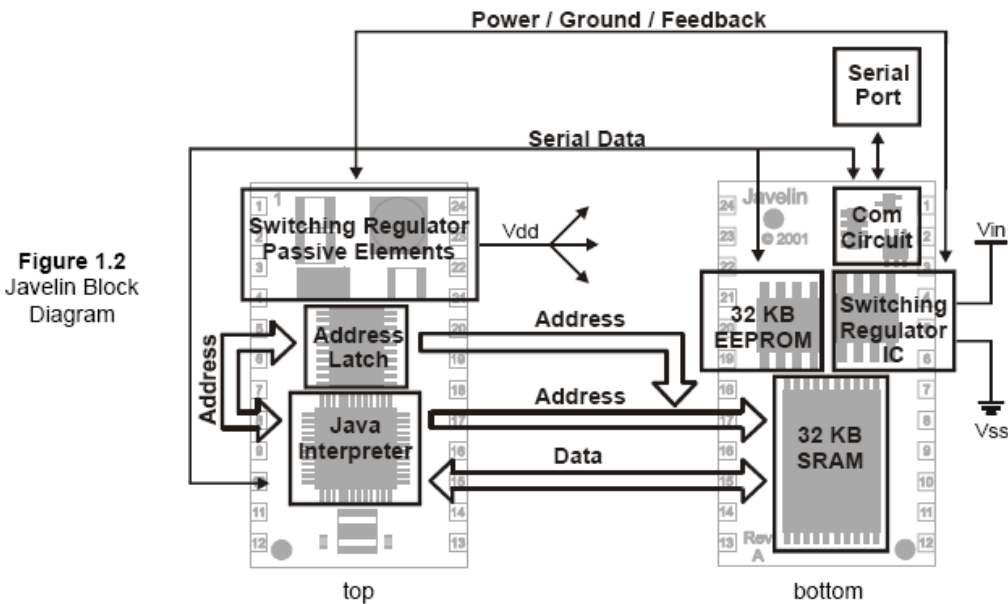


Figure 1.2
Javelin Block
Diagram

Javelin Stamp Hardware

Table 1.1 shows the Javelin Stamp's specifications. Note that the onboard voltage regulator can accept between 6 and 24 VDC and output up to 150 mA of current. Since the Javelin 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'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's voltage regulator output (Vdd) for peripherals. If in doubt, use an external 5 V regulator for your peripherals.

Table 1.1: Javelin 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	0° - 70° C (32° - 158° 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

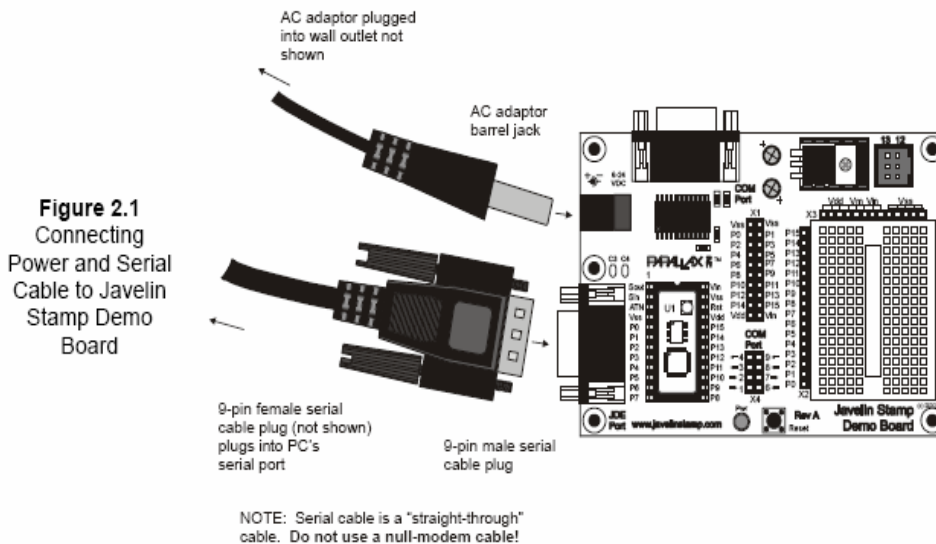
Hardware Setup

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's pins are all lined up with the holes in the socket, press down firmly with your thumb to make sure the Javelin 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.



Section 1

Running the Javelin Stamp IDE and Loading a Test Program

The Javelin 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. 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 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 to 1-million times.

The first example we'll try is a simple "hello world" program (Program Listing 2.1 below). It will cause the Javelin 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
- 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's power supply and serial cables are connected.
Click the Program button.

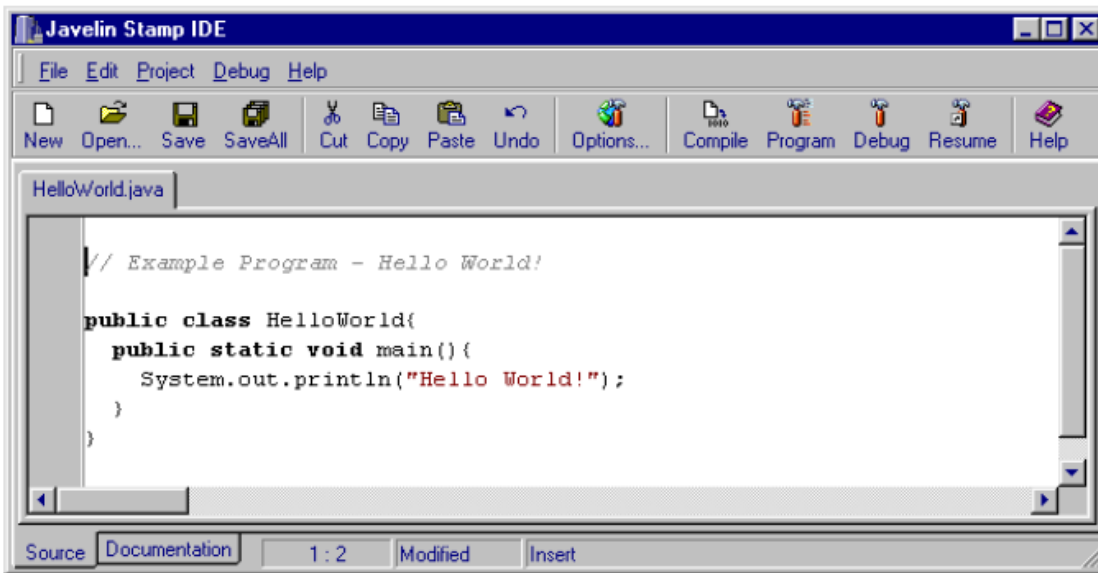
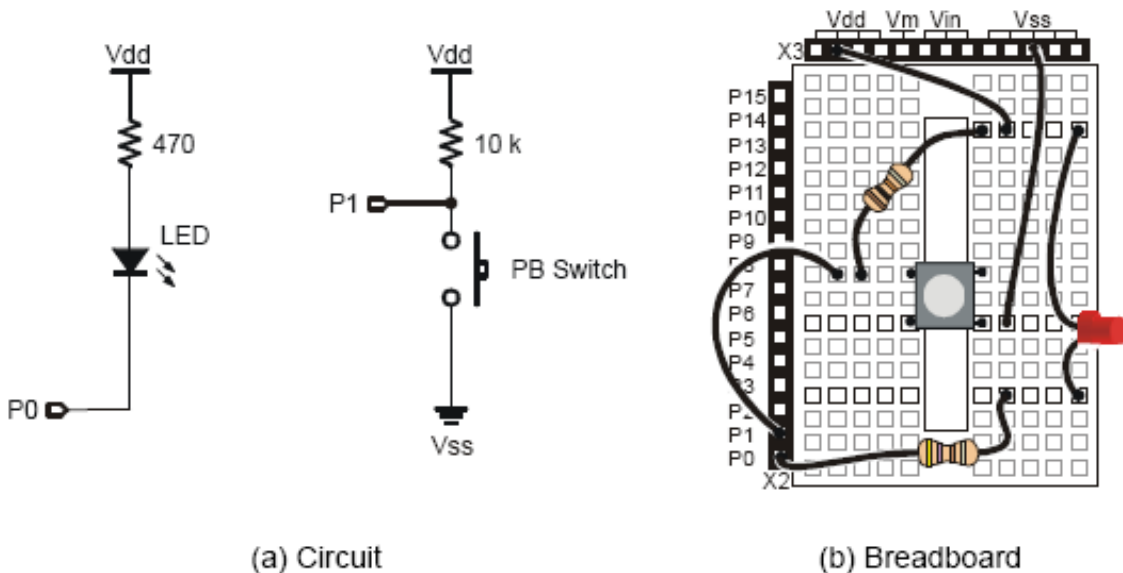


Figure 2.8 The Javelin Stamp IDE.

Homework 1: Program the javelin stamp to write your name on one line and your favorite sports team on the second using two separate commands. Save the program as your last name.

Section 2 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 the Figure 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 2.13(b).



Program Listing 2.2 – ButtonLED

```
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**. (case sensitive)
- Click the Program Button.

Homework 2: Program the LED to turn off when the button pushbutton is pressed. When the button is depressed program it to flash once per second by adjusting the delay. Save the file as ButtonLED2.

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.

Compiler Errors

If you did not enter the program correctly (Java is case sensitive), the IDE might display an error message below your program. 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 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.

If you get two messages, one of them stating that there is a possible Javelin found on COM 1, COM 2, etc, and the second stating that there is no Javelin 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 Javelins 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 2.14 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.

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() {  
...  
}
```

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.

Section 3 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 2.3 – DisplayVariables

```
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);
    }
}
```

- Enter this program as shown and save as respective class name
- Click the Program Button.

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 Javaneese, 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);
    }
}
```

- Enter this program as shown and save as respective class name
- Click the Program Button.

The Javelin Stamp supports the following fundamental (primitive) data types: **boolean**, **byte**, **char**, **int**, and **short**. Program Listing 3.4 declares and displays an example of each of these types.

Program Listing 3.4 - Display Primitive Types

```
import stamp.core.*;
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);
    }
}
```

- Enter this program as shown and save as respective class name
- Click the Program Button.

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

```
import stamp.core.*;
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]);
        }
    }
}
```

- Enter this program as shown and save as respective class name
- Click the Program Button.

[Homework3: Modify ExampleArray to count down from 6 and stop. Save the program as ExampleArray2.](#)

Section 4 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).

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. The 5th line multiplies a variable and a constant and stores the result in a variable. You can write arbitrarily complex expressions and use parenthesis 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

```
import stamp.core.*;
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);
    }
}
```

- Enter this program as shown and save as respective class name
- Click the Program Button.

[Homework 4: Design a program that takes in 5 numbers and computes their average. Only use constants and variables to do this. Print the average to the screen. Save the file as average2.](#)

Section 5 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 parenthesis. 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 parenthesis such as a boolean variable.

The statement after the parenthesis will only execute if the expression in parenthesis 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 parenthesis 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

```
import stamp.core.*;
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.");
        }
    }
}
```

- Enter this program as shown and save as respective class name

- Click the Program Button.

Homework 5: Modify the program with a third **if** statement that prints “The value is optimal” when the value is 150. Then modify it to print the optimum value using the **x** constant. Save the file as `DecisionExample2`.

Section 6

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 section 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

```
import stamp.core.*;
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;
        }
    }
}
```

- Enter this program as shown and save as respective class name.
- Click the Program Button.

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

```
import stamp.core.*;
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);
        }
    }
}
```

- Enter this program as shown and save as respective class name
- Click the Program Button.

Homework 6: First modify the code to count to 20 skipping every odd number. Then count to 100 by tens. Save the file as ForLoops2.

Section 7

Sending Messages to the Javelin

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. 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
```

- Enter this program as shown and save as respective class name
- Click the Program Button.

Homework 7: Modify the program to take in your first name (all lower case) and correct it using capitalization using multiple `getChar()` commands. Save the file as `Capitalize2`.

You are now ready for javelin stamp with the RoadRunner version 2!

The Javelin Stamp Manual can be found at <http://www.parallax.com>.