

## 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 on 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.

## 6: Javelin Stamp Programmers Reference

---

### 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.

## 6: Javelin Stamp Programmers Reference

---

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

Variables 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 ( <i>not Unicode</i> ) 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;
```

## 6: Javelin Stamp Programmers Reference

---

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

## 6: Javelin Stamp Programmers Reference

---

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
<code>\b</code>	Backspace
<code>\f</code>	Form Feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\u0013</code>	Clear Screen
<code>\\</code>	Backslash
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\xxx</code>	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).

## 6: Javelin Stamp Programmers Reference

---

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 precedence. It then moves down the list until the entire expression has been evaluated. Table 6.4 shows the precedence of the various operators.

## 6: Javelin Stamp Programmers Reference

---

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 parenthesis. 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 parenthesis 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.

## 6: Javelin Stamp Programmers Reference

---

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.

## 6: Javelin Stamp Programmers Reference

---

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 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
```

## 6: Javelin Stamp Programmers Reference

---

```
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;
    }
}
```

## 6: Javelin Stamp Programmers Reference

---

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

## 6: Javelin Stamp Programmers Reference

---

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 known as 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 `Thermostat` 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;
```

## 6: Javelin Stamp Programmers Reference

---

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 and 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 variables 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
```

## 6: Javelin Stamp Programmers Reference

---

```
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.

## 6: Javelin Stamp Programmers Reference

---

### 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();
    }
}
```

## 6: Javelin Stamp Programmers Reference

---

```
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.

## 6: Javelin Stamp Programmers Reference

---

### 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
```

## 6: Javelin Stamp Programmers Reference

---

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. 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

## 6: Javelin Stamp Programmers Reference

---

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
<code>equals</code>	Test objects for equality
<code>hashCode</code>	Returns id number (hash) for this object
<code>toString</code>	Returns a string representation of the object
<code>clone</code>	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.

## 6: Javelin Stamp Programmers Reference

---

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

## 6: Javelin Stamp Programmers Reference

---

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 parenthesis. 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();
```

## 6: Javelin Stamp Programmers Reference

---

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.

## 6: Javelin Stamp Programmers Reference

---

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 bn = (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.

## 6: Javelin Stamp Programmers Reference

---

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.

## 6: Javelin Stamp Programmers Reference

---

### 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

## 6: Javelin Stamp Programmers Reference

---

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.

## 6: Javelin Stamp Programmers Reference

---

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.

## 6: Javelin Stamp Programmers Reference

---

### 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/gotchas.html>

Java gotchas

<http://www.afu.com/javafaq.htm>

Java programmer's FAQ

<http://www.mindspring.com/~chroma/docwiz/>

Adds java doc comments to your code

<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

## 6: Javelin Stamp Programmers Reference

---

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:

## 6: Javelin Stamp Programmers Reference

---

`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*

## 6: Javelin Stamp Programmers Reference

---

### **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 {
    . . .
}
```

## 6: Javelin Stamp Programmers Reference

---

### 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 (;;) { . . . }
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

## 6: Javelin Stamp Programmers Reference

---

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 parenthesis. If this expression evaluates to **true**, the following statement executes. If it is **false**, execution continues with the next statement.

## 6: Javelin Stamp Programmers Reference

---

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.*;
```

## 6: Javelin Stamp Programmers Reference

---

*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 complement) 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**

## 6: Javelin Stamp Programmers Reference

---

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 parenthesis). 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 parenthesis.

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

## 6: Javelin Stamp Programmers Reference

---

**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
```

## 6: Javelin Stamp Programmers Reference

---

```
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;
}
```

## 6: Javelin Stamp Programmers Reference

---

*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.

## 6: Javelin Stamp Programmers Reference

---

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
    }
}
```

## 6: Javelin Stamp Programmers Reference

---

```
    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; }
}
```

## 6: Javelin Stamp Programmers Reference

---

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).

## 6: Javelin Stamp Programmers Reference

---

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();
    }
}
```

## 6: Javelin Stamp Programmers Reference

---

```
        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

[ ]

## 6: Javelin Stamp Programmers 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:

## 6: Javelin Stamp Programmers Reference

---

```
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 (%). Parenthesis 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 parenthesis, 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; // parenthesis 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
```

```
<<, >>, >>>
```

## 6: Javelin Stamp Programmers Reference

---

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 4<sup>th</sup> 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:

## 6: Javelin Stamp Programmers Reference

---

```
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;

?:
```

## 6: Javelin Stamp Programmers Reference

---

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 {  
    . . .  
}
```

## 6: Javelin Stamp Programmers Reference

---

```
}  
  
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. 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**