**Platform Independent**The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is more closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM.

**Simple**There are various features that makes the java as a simple language. Programs are easy to write and debug because java does not use the pointers explicitly. It is much harder to write the [java programs](http://www.roseindia.net/java/java-introduction/java-features.shtml) that can crash the system but we can not say about the other programming languages. Java provides the bug free system due to the strong memory management. It also has the automatic memory allocation and deallocation system.

**Object Oriented**To be an Object Oriented language, any language must follow at least the four characteristics.

* Inheritance           :       It is the process of creating the new classes and using the behavior of the existing classes by extending them just to                                    reuse  the existing code and adding the additional features as needed.
* Encapsulation:      :       It is the mechanism of combining the information and providing the abstraction.
* Polymorphism:     :       As the name suggest one name multiple form, Polymorphism is the way of providing the different functionality by the       
                                     functions  having the same name based on the signatures of the methods.
* Dynamic binding  :       Sometimes we don't have the knowledge of objects about their specific types while writing our code. It is the way                                   of providing the maximum functionality to a program about the specific type at runtime.

As the languages like Objective C, C++ fulfills the above four characteristics yet they  are not fully object oriented languages because they are structured as well as object oriented languages. But in case of java,  it is a fully Object Oriented language because object is at the outer most level of data structure in java. No stand alone methods, constants, and variables are there in java. Everything in java is object even the primitive data types can also be converted into object by using the wrapper class.

**Robust**Java has the strong memory allocation and automatic garbage collection mechanism. It provides the powerful exception handling and type checking mechanism as compare to other programming languages. Compiler checks the program whether there any error and interpreter checks any run time error and makes the system secure from crash. All of the above features makes the java language robust.

**Distributed**The widely used protocols like HTTP and [FTP](http://www.roseindia.net/java/java-introduction/java-features.shtml) are developed in java. Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing codes on their local system.

**Portable**The feature Write-once-run-anywhere  makes the java language portable provided that the system must have interpreter for the JVM. Java also have the standard data size irrespective of operating system or the [processor](http://www.roseindia.net/java/java-introduction/java-features.shtml). These features makes the java as a portable language.

**Dynamic**While executing the java program the user can get the required files dynamically from a local drive or from a computer thousands of miles away from the user just by connecting with the Internet.

**Secure**Java does not use memory pointers explicitly. All the programs in java are run under an area known as the sand box. Security manager determines the accessibility options of a class like reading and writing a file to the local disk. Java uses the public key encryption system to allow the java applications to transmit over the [internet](http://www.roseindia.net/java/java-introduction/java-features.shtml) in the secure encrypted form. The bytecode Verifier checks the classes after loading.

**Performance**Java uses native code usage, and lightweight process called  threads. In the beginning interpretation of bytecode resulted the [performance](http://www.roseindia.net/java/java-introduction/java-features.shtml) slow but the advance version of JVM uses the adaptive and just in time compilation technique that improves the performance.

**Multithreaded**  
As we all know several features of Java like Secure, Robust, Portable, dynamic etc; you will be more delighted to know another feature of Java which is **Multithreaded.**Java is also a Multithreaded programming language. Multithreading means a single program having different threads executing independently at the same time. Multiple threads execute instructions according to the program code in a process or a program. Multithreading works the similar way as multiple processes run on one computer.    
Multithreading programming is a very interesting concept in Java. In multithreaded programs not even a single thread disturbs the execution of other thread. Threads are obtained from the pool of available ready to run threads and they run on the system CPUs. This is how Multithreading works in Java which you will soon come to know in details in later chapters.

**Interpreted**We all know that Java is an interpreted language as well. With an interpreted language such as Java, programs run directly from the source code.   
The interpreter program reads the source code and translates it on the fly into computations. Thus, Java as an interpreted language depends on an interpreter program.  
The versatility of being **platform independent** makes Java to outshine from other languages. The source code to be written and distributed is platform independent.    
Another advantage of Java as an interpreted language is its error debugging quality. Due to this any error occurring in the program gets traced. This is how it is different to work with Java.

**Architecture Neutral**The term architectural neutral seems to be weird, but yes Java is an architectural neutral language as well. The growing popularity of networks makes developers think distributed. In the world of network it is essential that the applications must be able to migrate easily to different computer systems. Not only to computer systems but to a wide variety of hardware architecture and Operating system architectures as well.  The Java compiler does this by generating byte code instructions, to be easily interpreted on any machine and to be easily translated into native machine code on the fly.The compiler generates an architecture-neutral object file format to enable a Java application to execute anywhere on the network and then the compiled code is executed on many processors, given the presence of the Java runtime system.Hence Java was designed to support applications on network. This feature of Java has thrived the programming language.

**Why Java?**

* Similar to C++ so it is familiar to commercial programmers.
* Does not include the nasty dangerous parts of C++ so it is safe.
* Extensive run-time type information and safe dynamic link-loading is available.
* Includes string and multi-thread support in the language.
* Automatic memory management.
* Data type sizes and arithmetic behaviour are fixed and fully defined for all platforms.
* Has useful standard OO libraries.
* Documentation can be extracted from the source code.
* Security checking is built in to the libraries and virtual machine.
* Supports unicode for ease of internationalisation.
* Write once, run anywhere, any platform (no porting, no client configuration .. well almost!)
* Vast amount of supplier and programmer support and acceptance. It is unkillable.
* Loads and runs over the WWW, 40 million potential clients.

# Java (programming language)

### From Wikipedia, the free encyclopedia

Jump to: [navigation](http://en.wikipedia.org/wiki/Java_(programming_language)#column-one), [search](http://en.wikipedia.org/wiki/Java_(programming_language)#searchInput)

*"Java language" redirects here. For the Indonesian spoken language, see* [*Javanese language*](http://en.wikipedia.org/wiki/Javanese_language)*.*

*Not to be confused with* [*JavaScript*](http://en.wikipedia.org/wiki/JavaScript)*.*

|  |  |
| --- | --- |
| **Java** | |
| [Java logo.svg](http://en.wikipedia.org/wiki/File:Java_logo.svg) | |
| **Usual** [**file extensions**](http://en.wikipedia.org/wiki/Filename_extension) | .java, .class, .jar |
| [**Paradigm**](http://en.wikipedia.org/wiki/Programming_paradigm) | [Object-oriented](http://en.wikipedia.org/wiki/Object-oriented_programming), [structured](http://en.wikipedia.org/wiki/Structured_programming), [imperative](http://en.wikipedia.org/wiki/Imperative_programming) |
| **Appeared in** | 1995 |
| **Designed by** | [Sun Microsystems](http://en.wikipedia.org/wiki/Sun_Microsystems) (Owned by [Oracle Corporation](http://en.wikipedia.org/wiki/Oracle_Corporation)) |
| [**Developer**](http://en.wikipedia.org/wiki/Software_developer) | [James Gosling](http://en.wikipedia.org/wiki/James_Gosling) & Sun Microsystems |
| [**Stable release**](http://en.wikipedia.org/wiki/Software_release_life_cycle) | Java Standard Edition 6 (1.6.0\_18) (January 14, 2010; 20 days ago) |
| [**Typing discipline**](http://en.wikipedia.org/wiki/Type_system) | [Static, strong, safe](http://en.wikipedia.org/wiki/Type_system), [nominative](http://en.wikipedia.org/wiki/Nominative_type_system), [manifest](http://en.wikipedia.org/wiki/Manifest_typing) |
| [**Major implementations**](http://en.wikipedia.org/wiki/Programming_language_implementation) | [Numerous](http://en.wikipedia.org/wiki/Category:Java_virtual_machine) |
| [**Dialects**](http://en.wikipedia.org/wiki/Programming_language_dialect) | [Generic Java](http://en.wikipedia.org/wiki/Generic_Java), [Pizza](http://en.wikipedia.org/wiki/Pizza_(programming_language)) |
| **Influenced by** | [Ada 83](http://en.wikipedia.org/wiki/Ada_(programming_language)), [C++](http://en.wikipedia.org/wiki/C%2B%2B), [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)),[[1]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-0) [Delphi Object Pascal](http://en.wikipedia.org/wiki/Object_Pascal),[[2]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-1) [Eiffel](http://en.wikipedia.org/wiki/Eiffel_(programming_language)),[[3]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-2) [Generic Java](http://en.wikipedia.org/wiki/Generic_Java), [Mesa](http://en.wikipedia.org/wiki/Mesa_(programming_language)),[[4]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-3) [Modula-3](http://en.wikipedia.org/wiki/Modula-3),[[5]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-4) [Objective-C](http://en.wikipedia.org/wiki/Objective-C),[[6]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-5) [UCSD Pascal](http://en.wikipedia.org/wiki/UCSD_Pascal),[[7]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-6)[[8]](http://en.wikipedia.org/wiki/Java_(programming_language)#cite_note-7) [Smalltalk](http://en.wikipedia.org/wiki/Smalltalk) |
| **Influenced** | [Ada 2005](http://en.wikipedia.org/wiki/Ada_(programming_language)), [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)), [Clojure](http://en.wikipedia.org/wiki/Clojure), [D](http://en.wikipedia.org/wiki/D_(programming_language)), [ECMAScript](http://en.wikipedia.org/wiki/ECMAScript), [Groovy](http://en.wikipedia.org/wiki/Groovy_(programming_language)), [J#](http://en.wikipedia.org/wiki/J_Sharp), [PHP](http://en.wikipedia.org/wiki/PHP), [Scala](http://en.wikipedia.org/wiki/Scala_(programming_language)), [JavaScript](http://en.wikipedia.org/wiki/JavaScript), [Python](http://en.wikipedia.org/wiki/Python_(programming_language)), [BeanShell](http://en.wikipedia.org/wiki/BeanShell) |
| [**OS**](http://en.wikipedia.org/wiki/Operating_system) | [Cross-platform (multi-platform)](http://en.wikipedia.org/wiki/Cross-platform) |
| [**License**](http://en.wikipedia.org/wiki/Software_license) | [GNU General Public License](http://en.wikipedia.org/wiki/GNU_General_Public_License) / [Java Community Process](http://en.wikipedia.org/wiki/Java_Community_Process) |
| [**Website**](http://en.wikipedia.org/wiki/Website) | [java.sun.com](http://java.sun.com) |

**Java** is a [programming language](http://en.wikipedia.org/wiki/Programming_language) originally developed by [James Gosling](http://en.wikipedia.org/wiki/James_Gosling) at [Sun Microsystems](http://en.wikipedia.org/wiki/Sun_Microsystems) now a [subsidiary](http://en.wikipedia.org/wiki/Subsidiary) of [Oracle Corporation](http://en.wikipedia.org/wiki/Oracle_Corporation), and released in 1995 as a core component of Sun Microsystems' [Java platform](http://en.wikipedia.org/wiki/Java_(software_platform)). The language derives much of its [syntax](http://en.wikipedia.org/wiki/Syntax_(programming_languages)) from [C](http://en.wikipedia.org/wiki/C_(programming_language)) and [C++](http://en.wikipedia.org/wiki/C%2B%2B) but has a simpler [object model](http://en.wikipedia.org/wiki/Object_model) and fewer low-level facilities. Java applications are typically [compiled](http://en.wikipedia.org/wiki/Compiler) to [bytecode](http://en.wikipedia.org/wiki/Java_bytecode) ([class file](http://en.wikipedia.org/wiki/Class_(file_format))) that can run on any [Java Virtual Machine](http://en.wikipedia.org/wiki/Java_Virtual_Machine) (JVM) regardless of [computer architecture](http://en.wikipedia.org/wiki/Computer_architecture). Java is general-purpose, concurrent, class-based, and object-oriented, and is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere".

# The Introduction to Applet Programming

***by Rogers Cadenhead***

### CONTENTS

* [Viewing Applets](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch17.htm#ViewingApplets)
* [Providing Security](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch17.htm#ProvidingSecurity)
* [The Basics of Applet Programming](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch17.htm#TheBasicsofAppletProgramming)
* [Putting the Applet on the Web](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch17.htm#PuttingtheAppletontheWeb)
* [An Example: The ColorCycle Applet](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch17.htm#AnExampleTheColorCycleApplet)
* [Summary](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch17.htm#Summary)

Although Java is a general-purpose programming language suitable for a large variety of tasks, the task most people use it for is applet programming. An *applet* is a Java program that executes on a World Wide Web page.

When the prerelease versions of the Java Developers Kit were made available in 1995, the demonstration programs that drew international attention to the language were applets. Your first experience with Java might have been one of these demos-spinning heads, the animated Duke character doing cartwheels, a dancing headline, and so on. Those applets are still available on the JavaSoft site at the following URL:

<http://java.sun.com/java.sun.com/applets/applets.html>

Today, applets are being used to accomplish far more than demonstrative goals. There are working examples of applets on Web sites throughout the Internet-a check of the AltaVista search engine finds more than 4,200 Web pages that have applets embedded on them.

The current uses of applets include the following:

* Tickertape-style news and sports headline updates
* Animated graphics
* Video games
* Student tests
* Image maps that respond to mouse movement
* Advanced text displays
* Database reports

Figure 17.1 shows a noteworthy example of an applet: the Instant Ballpark program from   
Instant Sports.

[**Figure 17.1:** *The Instant Ballpark applet (courtesy of Instant Sports).*](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/f17-1.gif)

Instant Ballpark takes real-time data from live baseball games and updates its display to reflect what's happening in the game. Players run the bases, the ball goes to the place it was hit, and sound effects are used for strike calls, crowd noise, and other elements. The program, which was unique enough to qualify for a U.S. patent, is reminiscent of the old-time baseball tradition of presenting the play-by-play for road games by moving metal figures on the side of a building. In addition to the live coverage, Instant Ballpark can be used to review the play-by-play action of past games.

The applet shows one of the advantages of a Web program over a Web page. With HTML and some kind of gateway programming language such as Perl, a Web page can offer textual updates to a game in progress. However, Instant Ballpark offers a *visual* presentation of a live game in addition to text, and the applet can respond immediately to user input. Java can be used to provide information to Web users in a more compelling way, which is often the reason site providers are offering applets.

To try this applet, visit the following Web site:

<http://www.instantsports.com/ballpark.html>

## Viewing Applets

As you know, applets are displayed as a part of a Web page. A special HTML tag, <APPLET>, is used to attach a Java applet to an HTML page. Running an applet requires the use of a Web browser or other software that serves the function of a browser, such as the applet viewer program that ships with the Java Developers Kit from JavaSoft.

The browser acts as the operating system for applets-you cannot run an applet as a standalone program in the same way you can run an executable file.

At the time of this writing, there are three widely available Web browsers that can run Java applets:

* Netscape Navigator version 2.02 or higher
* Microsoft Internet Explorer 3.0
* JavaSoft HotJava 1.0 pre-beta 1

These programs load applets from a Web page and run them remotely on the Web user's computer. This arrangement raises security issues that must be handled by the Java language itself and by Java-enabled browsers. These browsers are covered in detail in [Chapter 3](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch3.htm), "Browsing Java."

## Providing Security

Java applets are programs that run on a Web user's machine. Anything that can execute code is a potential security risk because of the damaging things that can occur. Viruses can damage a computer's file system and reproduce onto other disks, Trojan horses can masquerade as helpful programs while doing harmful things, and programs can be written to retrieve private information surreptitiously. Even Microsoft Word has been a security risk because of Word Basic-an executable programming language that can be used in conjunction with Word documents.

Security is one of the primary concerns of Java's developers, and they have implemented safeguards at several levels. Some of these safeguards affect the language as a whole: The removal of pointers, the verification of bytecodes, and other language issues have been discussed elsewhere in this book.

Some of Java's functionality is not possible when programming applets because of security concerns. The following safeguards are in place:

* Applets cannot read or write files on the Web user's disk. If information must be saved to disk during an applet's execution (as in the case of a video game saving the top 10 scores), the storage of information must be done on the disk from which the Web page is served.
* Applets cannot make a network connection to a computer other than the one from which the Web page is served.
* Pop-up windows opened by applets are identified clearly as Java windows. A Java cup icon and text such as Untrusted Applet Window appear in the window. These elements are added to prevent a window opened by Java from pretending to be something else, such as a Windows dialog box requesting a user's name and password.
* Applets cannot use dynamic or shared libraries from any other programming language. Java can make use of programs written in languages such as Visual C++ by using a native statement from within Java. However, applets cannot make use of this feature because there's no way to adequately verify the security of the non-Java code being executed.
* Applets cannot run any programs on the Web user's system.

As you can see, Java applets are more limited in functionality than standalone Java applications. The loss is a tradeoff for the security that must be in place for the language to run remotely on users' computers.

The security restrictions discussed here are current as of the 1.0.2 release of the Java Developers Kit. JDK version 1.1 is in development as of this writing, but it is expected to add more security rather than lessening any of the existing safeguards. Refer to [Chapter 35](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/ch35.htm), "Java Security," for more information.

|  |
| --- |
| **A word about applications** |
| This chapter focuses on applets, but it's important to make clear the distinction between the two types of Java programs. Applets are programs offered on Web pages that require the use of a Web browser to execute. Applications are everything else: general-purpose programs run by executing the Java interpreter with the name of the Java program as an argument. For example, to run the Java program ReadNews.class, enter the following at a command-line prompt:  java ReadNews  Applications do not have any of the restrictions that are in place for applets. |

## The Basics of Applet Programming

Now that you understand what applets are, it's time to get out the tools and build one. Before starting the project, however, the following sections introduce some basic elements of applet programming.

### The java.applet.Applet Class

Each applet starts out with a class definition such as the following:

public class LearnPigLatin extends java.applet.Applet {

// to do

}

In this example, LearnPigLatin is the name of the applet's class. An applet must be declared as a public class. Applets are subclasses of java.applet.Applet, which is a subclass of the java.awt.Panel class. Figure 17.2 shows the full class hierarchy tree of the Applet class.

[**Figure 17.2:** *The hierarchy of* Java.applet.Applet.](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/f17-2.gif)

The superclasses of Applet give all applets a framework on which you can build user interface elements and mouse events. These superclasses also provide a structure for the applet that must be used when the program is developed.

### Applet Methods

The structure of an applet takes the form of five events that can take place as an applet is running. When the events occur, a method is automatically called. The methods also can be called directly within the applet. The methods are the following:

* **Initialization:** The init() method is called the first time the applet is loaded.
* **Destruction:** The destroy() method is called the final time the applet is exited.
* **Stopping:** The stop() method is called each time an applet is stopped. A *stop* happens automatically when a Web page containing the applet is exited and also when the stop() method is called directly in a program.
* **Starting:** The start() method is called each time an applet is loaded or reloaded. A *start* follows initialization and also takes place each time the applet is restarted. A *start* happens when a Web user comes back to the applet's page after leaving it; you can also call start() directly.
* **Painting:** The paint()method is called any time the applet window must be repainted. This occurs automatically at certain times, such as when the applet window is covered up by another window and then uncovered. It also can be called by using a repaint() call when a program needs a screen update to take place.

The last of the methods, paint(), must take a parameter-an instance of the Graphics class-as in the following method definition:

public void paint(Graphics g) {

g.drawString("One moment, please", 5, 50);

}

A Graphics object is used to indicate where something should be drawn. The Graphics object used as the parameter to paint() is created automatically, and it represents the applet window. The g.drawString() line uses this Graphics object to indicate where a string should be drawn. Every time the repaint() method is called and the applet window must be updated, the string One moment, please is drawn at the x, y position (in this example, 5, 50).

The Graphics object does not have to be declared. However, the .Graphics class must be imported at the beginning of an applet's source code. Here's what that import statement should look like:

import java.awt.Graphics;

Each of these applet methods-init(), destroy(), start(), stop(), and paint()-is inherited by an applet. You do not have to write your own methods for any of these.

However, each of the applet methods is empty by default. If something is supposed to happen in an applet, some or all of these methods must be overridden.

### The <APPLET> Tag

For a Java applet to be run when a Web page is loaded, information about that applet must be put on the page. This requires the use of two special HTML tags: <APPLET> and <PARAM>. This HTML code is included on a Web page along with all other HTML code. In this respect, putting a Java applet on your home page is no different than putting a picture there.

Java applets can be viewed by Web browsers and any other software that is equipped to load applets, such as the applet viewer utility that comes with the Java Developers Kit.

Following is an example of an applet tag:

<APPLET CODE="NowShowing.class" CODEBASE="progdir" WIDTH=376 HEIGHT=104>

<PARAM NAME="speed" value="100">

<PARAM NAME="blink" value="5">

<PARAM NAME="text" value="FREE THE BOUND PERIODICALS!">

<PARAM NAME="fontsize" value="21">

<PARAM NAME="pattern" value="random">

<H5>This applet requires the use of a Java-enabled browser!</H5>

</APPLET>

When included on a Web page, this HTML code causes the following to take place on a Java-enabled browser:

1. An applet called NowShowing.class is loaded from a directory called progdir. The CODE attribute specifies the applet to load, and the optional CODEBASE attribute refers to a directory where the applet can be found.
2. The applet is set to a width of 376 pixels and a height of 104 pixels using the WIDTH and HEIGHT attributes.
3. A parameter named speed is sent to the applet with a value of 100. Four other parameters are sent to the applet: blink, text, fontsize, and pattern. Parameters are optional; you can include as many as you want. The NAME attribute indicates the name a parameter should be given, and the VALUE attribute indicates the value to associate with the parameter.
4. The line <H5>This applet requires a Java-enabled browser!</H5> is ignored.

The HTML code causes the following to take place on a browser that is *not* equipped to run Java programs:

1. The line <H5>This applet requires a Java-enabled browser!</H5> is shown.
2. Everything else is disregarded.

Browsers that do not handle Java programs disregard everything within the <APPLET>, </APPLET>, and <PARAM> tags. As shown in the preceding HTML code of an applet tag, an alternative can be provided for browsers that do not handle Java.

The CODE attribute must be used in conjunction with the <APPLET> tag because it specifies the name of the applet's class file. This is the file that will be run after it has been loaded onto the Web user's computer.

If the CODEBASE attribute is used, it indicates the path from the Web page's directory to the directory containing the applet's class file. For example, CODEBASE="usr" indicates that the applet is in a directory called usr that is a subdirectory of the Web page's directory.

If the applet makes use of class files that are not part of the standard Java API, these class files must be located in the same directory as the applet's class file.

The HEIGHT and WEIGHT attributes should be familiar to anyone who has used them to place an image on a Web page-they work the same with <APPLET> as they do with <IMG>. The ALIGN attribute used with images also can be used with <APPLET>. The ALIGN attribute determines how the applet is positioned in relation to the other parts of the Web page and can have the values TOP, MIDDLE, or BOTTOM.

### Using Parameters

Parameters can be sent to an applet by using the <PARAM> tag and its two attributes: NAME and VALUE. Here's a line from the preceding example:

<PARAM NAME="blink" VALUE="100">

The value of the NAME attribute assigns a name to an applet parameter, and VALUE gives the parameter a value. The preceding statement sends a parameter named blink with a value of 100.

|  |
| --- |
| **Note** |
| The name of a parameter is not case sensitive, so the capitalization of the value assigned to NAME does not matter. |

Parameters are sent to an applet when it is loaded; you can send as many parameters as you want. All parameters are sent to applets as strings and must be converted to other data types if they are needed as integers or other types.

For a parameter to be used by a Java applet, the applet must retrieve the parameter. This requires the getParameter() method, which is available to all applets because it is part of the Applet class.

For example, use the following line in a Java applet to store the blink parameter in a variable called blinkValue:

String blinkValue = getParameter("blink");

If you want to retrieve the value and convert it to an integer, use the following code:

int blinkValue = -1;

try { blinkValue = Integer.parseInt(getParameter("blink")); }

catch (NumberFormatException e) { }

This example uses the parseInt() method of the java.lang.Integer class to convert a String into an int. The try and catch block is used to trap errors if the String cannot be converted into a number.

## Putting the Applet on the Web

When you have created an applet and added it to HTML pages, you easily can make it available on the World Wide Web. Put all .class files required by the applet on your Web site, making sure to put the files in the same directory as the CODEBASE attribute if it has been used. If not, put the .class files in the same directory as the Web page that includes the applet.

That's all it takes. Unlike CGI programming (which requires special access to the computer providing the Web pages), Java applets can be added by anyone who can put pages on a Web site.

## An Example: The ColorCycle Applet

In the next chapter, you will delve into specific details of applet programming, including user interface design and event handling. For now, it is worthwhile to take a look at a working example of an applet to get a clearer picture of how applets are designed.

The ColorCycle applet is a simple applet with one button labeled *Next Color*. When the button is clicked with the mouse, the background color of the applet changes.

The program demonstrates basic applet structure and a simple bit of event handling-how to respond to a mouse click on a button.

### Programming the Applet

Listing 17.1 shows the full source code of ColorCycle.java. It can be found on the book's CD-ROM in the directory \WIN95NT4\SOURCE\CHAP17 (Windows 95 and Windows NT 4 users) or in the directory \SOURCE\CHAP17 (Macintosh users), along with the rest of the Java and HTML source code in this chapter. Windows NT 3.51 users must install the source code on their hard drives or select these files from the zipped source code located on the CD-ROM.

**Listing 17.1. The source code of ColorCycle.java.**

1: import java.awt.\*;

2:

3: public class ColorCycle extends java.applet.Applet {

4: float hue = (float).5;6L5: float saturation = (float)1;

6: float brightness = (float)0;6L7: Button b;

8:

9: public void init() {

10: b = new Button("Next Color");

11: add(b);

12: }

13:

14: public void start() {

15: setBackground(Color.black);

16: repaint();

17: }

18:

19: public boolean action(Event evt, Object o) {

20: if (brightness < 1)

21: brightness += .25;

22: else

23: brightness = 0;

24: Color c = new Color(Color.HSBtoRGB(hue, saturation, brightness));

25: setBackground;

26: repaint();

27: return true;

28: }

29: }

Don't worry if some aspects of this program are unfamiliar to you at this point. Several aspects of this applet are discussed fully in the next chapter, including the creation of user interface components such as buttons and the action() method.

The following things are taking place in the applet:

* **Line 1:** The applet imports several classes by using the wildcard character with java.awt.\*. The awt stands for Abstract Windowing Toolkit, the set of classes used to handle most visual elements of Java programming-graphics, fonts, a user interface-and also to respond to user input from the keyboard and mouse.
* **Lines 4 through 6:** Three instance variables are created to store the HSB values of the color being displayed. HSB (Hue, Saturation, and Brightness) is a method of describing a color as three numeric values from 0 to 1.
* **Line 7:** A Button object is created.
* **Lines 9 through 12:** In the init() method of the applet, which is called automatically when the applet is first run, the Button object b is instantiated and is assigned the label *Next Color*.
* **Lines 14 through 17:** In the start() method of the applet, which is called after init() and whenever a Web user returns to the page containing the applet, the background color of the applet is set to black by using the Color constant Color.black. Additionally, a call to the repaint() method tells the applet that the window must be redrawn because something-in this case, the background color-has changed.
* **Line 19:** The action() method is called whenever a user interface component generates an action event. In this applet, an event occurs when the Next Color button is clicked. There is more on this in the next chapter.
* **Lines 20 through 23:** The value of brightness is changed so that the background cycles through several colors ranging from black to light blue.
* **Lines 24 through 26:** A Color object is created to store the value of the background color, which is created based on the values of the variables hue, saturation, and brightness. The background color is changed and another call to repaint() is made.
* **Line 27:** The boolean value true is returned at the end of the action() method, indicating that the action event generated by clicking the button was taken care of.

### Designing the HTML Page

Once the applet has been written and compiled using your development software, you can put it on a Web page using the HTML tags <APPLET>, </APPLET>, and <PARAM> described earlier in this chapter.

Listing 17.2 shows the full text of an HTML page that loads the ColorCycle.class applet (the source code can also be found on the CD-ROM that accompanies this book). The CODEBASE attribute is not used with the <APPLET> tag, so the ColorCycle.class file must be placed in the same directory as the Web page containing the applet.

**Listing 17.2. The source code of ColorCycle.html.**

1: <html>

2: <body>

3: <applet code=ColorCycle.java height=250 width=250>

4: </applet>

5: </body>

6: </html>

Although the applet loses something in the translation from color to black and white, Fig-ure 17.3 shows how the ColorCycle applet looks when loaded with Netscape Navigator 2.02 for Windows 95, one of the current Web browsers equipped to handle Java programs.

[**Figure 17.3:** *The* ColorCycle *applet.*](http://www.ssuet.edu.pk/taimoor/books/1-57521-197-1/f17-3.gif)

|  |  |
| --- | --- |
|  | Developing Java Beans By Robert Englander 1st Edition June 1997 1-56592-289-1, Order Number: 2891 316 pages, $29.95 |

## Chapter 1. Introduction

|  |
| --- |
| **In this chapter:**  [The Component Model](http://oreilly.com/catalog/javabeans/chapter/ch01.html#36745)  [The JavaBeans Architecture](http://oreilly.com/catalog/javabeans/chapter/ch01.html#16613)  [JavaBeans Overview](http://oreilly.com/catalog/javabeans/chapter/ch01.html#36035)  [Using Design Patterns](http://oreilly.com/catalog/javabeans/chapter/ch01.html#26740)  [JavaBeans vs. ActiveX](http://oreilly.com/catalog/javabeans/chapter/ch01.html#21043)  [Getting Started](http://oreilly.com/catalog/javabeans/chapter/ch01.html#22492) |

As software developers, we are constantly being asked to build applications in less time and with less money. And, of course, these applications are expected to be better and faster than ever before. Object-oriented techniques and component software environments are in wide use now, in the hope that they can help us build applications more quickly. Development tools like Microsoft's Visual Basic have made it easier to build applications faster by taking a building-block approach to software development. Such tools provide a visual programming model that allows you to include software components rapidly in your applications.

The JavaBeans architecture brings the component development model to Java, and that's the subject of this book. But before we get started, I want to spend a little time describing the component model, and follow that with a general overview of JavaBeans. If you already have an understanding of these subjects, or you just want to get right into it, you can go directly to Chapter 2, Events. Otherwise, you'll probably find that the information in this chapter sets the stage for the rest of the book.

## The Component Model

Components are self-contained elements of software that can be controlled dynamically and assembled to form applications. But that's not the end of it. These components must also interoperate according to a set of rules and guidelines. They must behave in ways that are expected. It's like a society of software citizens. The citizens (components) bring functionality, while the society (environment) brings structure and order.

JavaBeans is Java's component model. It allows users to construct applications by piecing components together either programmatically or visually (or both). Support of visual programming is paramount to the component model; it's what makes component-based software development truly powerful.

The model is made up of an architecture and an API (Application Programming Interface). Together, these elements provide a structure whereby components can be combined to create an application. This environment provides services and rules, the framework that allows components to participate properly. This means that components are provided with the tools necessary to work in the environment, and they exhibit certain behaviors that identify them as such. One very important aspect of this structure is containment. A container provides a context in which components can interact. A common example would be a panel that provides layout management or mediation of interactions for visual components. Of course, containers themselves can be components.

As mentioned previously, components are expected to exhibit certain behaviors and characteristics in order to participate in the component structure and to interact with the environment, as well as with other components. In other words, there are a number of elements that, when combined, define the component model. These are described in more detail in the following sections.

### Discovery and Registration

Class and interface discovery is the mechanism used to locate a component at run-time and to determine its supported interfaces so that these interfaces can be used by others. The component model must also provide a registration process for a component to make itself and its interfaces known. The component, along with its supported interfaces, can then be discovered at run-time. Dynamic (or late) binding allows components and applications to be developed independently. The dependency is limited to the "contract" between each component and the applications that use it; this contract is defined by interfaces that the component supports. An application does not have to include a component during the development process in order to use it at run-time; it only needs to know what the component is capable of doing. Dynamic discovery also allows developers to update components without having to rebuild the applications that use them.

This discovery process can also be used in a design-time environment. In this case, a development tool may be able to locate a component and make it available for use by the designer. This is important for visual programming environments, which are discussed later.

### Raising and Handling of Events

An event is something of importance that happens at a specific point in time. An event can take place due to a user action such as a mouse clickwhen the user clicks a mouse button, an event takes place. Events can also be initiated by other means. Imagine the heating system in your house. It contains a thermostat that sets the desired comfort temperature, keeps track of the current ambient temperature, and notifies the boiler when its services are required. If the thermostat is set to keep the room at 70 degrees Fahrenheit, it will notify the boiler to start producing heat if the temperature dips below that threshold. Components will send notifications to other objects when an event takes place in which those objects have expressed an interest.

### Persistence

Generally, all components have state. The thermostat component has state that represents the comfort temperature. If the thermostat were a software component of a computer-based heating control system, we would want the value of the comfort temperature to be stored on a non-volatile storage medium (such as the hard disk). This way if we shut down the application and brought it back up again, the thermostat control would still be set to 70 degrees. The visual representation and position of the thermostat relative to other components in the application would be restored as well.

Components must be able to participate in their container's persistence mechanism so that all components in the application can provide application-wide persistence in a uniform way. If every component were to implement its own method of persistence, it would be impossible for an application container to use components in a general way. This wouldn't be an issue if reuse weren't the goal. If we were building a monolithic temperature control system we might create an application-specific mechanism for storing state. But we want to build the thermostat component so that it can be used again in another application, so we have to use a standard mechanism for persistence.

### Visual Presentation

The component environment allows the individual components to control most of the aspects of their visual presentation. For example, imagine that our thermostat component includes a display of the current ambient temperature. We might want to display the temperature in different fonts or colors depending on whether we are above, below, or at the comfort temperature. The component is free to choose the characteristics of its own visual presentation. Many of these characteristics will be properties of the component (a topic that will be discussed later). Some of these visual properties will be persistent, meaning that they represent some state of the control that will be saved to, and restored from, persistent storage.

Layout is another important aspect of visual presentation. This concerns the way in which components are arranged on the screen, how they relate to one another, and the behavior they exhibit when the user interacts with them. The container object that holds an assembly of components usually provides some set of services related to the layout of the component. Let's consider the thermostat and heating control application again. This time, the user decides to change the size of the application window. The container will interact with the components in response to this action, possibly changing the size of some of the components. In turn, changing the size of the thermostat component may cause it to alter its font size.

As you can see, the container and the component work together to provide a single application that presents itself in a uniform fashion. The application appears to be working as one unit, even though with the component development model, the container and the components probably have been created separately by different developers.

### Support of Visual Programming

Visual programming is a key part of the component model. Components are represented in toolboxes or palettes. The user can select a component from the toolbox and place it into a container, choosing its size and position. The properties of the component can then be edited in order to create the desired behavior. Our thermostat control might present some type of user interface to the application developer to set the initial comfort temperature. Likewise, the choice of font and color will be selectable in a similar way. None of these manipulations require a single line of code to be written by the application developer. In fact, the application development tool is probably writing the code for you. This is accomplished through a set of standard interfaces provided by the component environment that allow the components to publish, or expose, their properties. The development tool can also provide a means for the developer to manipulate the size and position of components in relation to each other. The container itself may be a component and allow its properties to be edited in order to alter its behavior.

## The JavaBeans Architecture

JavaBeans is an architecture for both using and building components in Java. This architecture supports the features of software reuse, component models, and object orientation. One of the most important features of JavaBeans is that it does not alter the existing Java language. If you know how to write software in Java, you know how to use and create Beans. The strengths of Java are built upon and extended to create the JavaBeans component architecture.

Although Beans are intended to work in a visual application development tool, they don't necessarily have a visual representation at run-time (although many will). What this does mean is that Beans must allow their property values to be changed through some type of visual interface, and their methods and events should be exposed so that the development tool can write code capable of manipulating the component when the application is executed.

Creating a Bean doesn't require any advanced concepts. So before I go any further, here is some code that implements a simple Bean:

public class MyBean implements java.io.Serializable

{

protected int theValue;

public MyBean()

{

}

public void setMyValue(int newValue)

{

theValue = newValue;

}

public int getMyValue()

{

return theValue;

}

}

This is a real Bean named MyBean that has state (the variable theValue) that will automatically be saved and restored by the JavaBeans persistence mechanism, and it has a property named MyValue that is usable by a visual programming environment. This Bean doesn't have any visual representation, but that isn't a requirement for a JavaBean component.

JavaSoft is using the slogan "Write once, use everywhere." Of course "everywhere" means everywhere the Java run-time environment is available. But this is very important. What it means is that the entire run-time environment required by JavaBeans is part of the Java platform. No special libraries or classes have to be distributed with your components. The JavaBeans class libraries provide a rich set of default behaviors for simple components (such as the one shown earlier). This means that you don't have to spend your time building a lot of support for the Beans environment into your code.

The design goals of JavaBeans are discussed in Sun's white paper, "Java Beans: A Component Architecture for Java." This paper can be found on the JavaSoft web site at http://splash.javasoft.com/beans/WhitePaper.html. It might be interesting to review these goals before we move on to the technology itself, to provide a little insight into why certain aspects of JavaBeans are the way they are.

### Compact and Easy

JavaBeans components are simple to create and easy to use. This is an important goal of the JavaBeans architecture. It doesn't take very much to write a simple Bean, and such a Bean is lightweightit doesn't have to carry around a lot of inherited baggage just to support the Beans environment. If a Bean does not require the advanced features of the architecture, it doesn't get them, nor does it get the code that goes with them. This is an important concept. The JavaBeans architecture scales upward in complexity, not downward like other component models. This means it really is easy to create a simple Bean. (The previous example shows just how simple a Bean can be.)

### Portable

Since JavaBeans components are built purely in Java, they are fully portable to any platform that supports the Java run-time environment. All platform specifics, as well as support for JavaBeans, are implemented by the Java virtual machine. You can be sure that when you develop a component using JavaBeans it will be usable on all of the platforms that support Java (version 1.1 and beyond). These range from workstation applications and web browsers to servers, and even to devices such as PDAs and set-top boxes.

### Leverages the Strengths of the Java Platform

JavaBeans uses the existing Java class discovery mechanism. This means that there isn't some new complicated mechanism for registering components with the run-time system.

As shown in the earlier code example, Beans are lightweight components that are easy to understand. Building a Bean doesn't require the use of complex extensions to the environment. Many of the Java supporting classes are Beans, such as the windowing components found in java.awt.

The Java class libraries provide a rich set of default behaviors for components. Use of Java Object Serialization is one examplea component can support the persistence model by implementing the java.io.Serializable interface. By conforming to a simple set of design patterns (discussed later in this chapter), you can expose properties without doing anything more than coding them in a particular style.

### Flexible Build-Time Component Editors

Developers are free to create their own custom property sheets and editors for use with their components if the defaults aren't appropriate for a particular component. It's possible to create elaborate property editors for changing the value of specific properties, as well as create sophisticated property sheets to house those editors.

Imagine that you have created a Sound class that is capable of playing various sound format files. You could create a custom property editor for this class that listed all of the known system sounds in a list. If you have created a specialized color type called PrimaryColor, you could create a color picker class to be used as the property editor for PrimaryColor that presented only primary colors as choices.

The JavaBeans architecture also allows you to associate a custom editor with your component. If the task of setting the property values and behaviors of your component is complicated, it may be useful to create a component wizard that guides the user through the steps. The size and complexity of your component editor is entirely up to you.

## JavaBeans Overview

The JavaBeans white paper defines a Bean as follows:

A Java Bean is a reusable software component that can be manipulated visually in a builder tool.

Well, if you have to sum it up in one sentence, this is as good as any. But it's pretty difficult to sum up an entire component architecture in one sentence. Beans will range greatly in their features and capabilities. Some will be very simple and others complex; some will have a visual aspect and others won't. Therefore, it isn't easy to put all Beans into a single category. Let's take a look at some of the most important features and issues surrounding Beans. This should set the stage for the rest of the book, where we will examine the JavaBeans technology in depth.

### Properties, Methods, and Events

Properties are attributes of a Bean that are referenced by name. These properties are usually read and written by calling methods on the Bean specifically created for that purpose. A property of the thermostat component mentioned earlier in the chapter could be the comfort temperature. A programmer would set or get the value of this property through method calls, while an application developer using a visual development tool would manipulate the value of this property using a visual property editor.

The methods of a Bean are just the Java methods exposed by the class that implements the Bean. These methods represent the interface used to access and manipulate the component. Usually, the set of public methods defined by the class will map directly to the supported methods for the Bean, although the Bean developer can choose to expose only a subset of the public methods.

Events are the mechanism used by one component to send notifications to another. One component can register its interest in the events generated by another. Whenever the event occurs, the interested component will be notified by having one of its methods invoked. The process of registering interest in an event is carried out simply by calling the appropriate method on the component that is the source of the event. In turn, when an event occurs a method will be invoked on the component that registered its interest. In most cases, more than one component can register for event notifications from a single source. The component that is interested in event notifications is said to be listening for the event.

### Introspection

Introspection is the process of exposing the properties, methods, and events that a JavaBean component supports. This process is used at run-time, as well as by a visual development tool at design-time. The default behavior of this process allows for the automatic introspection of any Bean. A low-level reflection mechanism is used to analyze the Bean's class to determine its methods. Next it applies some simple design patterns to determine the properties and events that are supported. To take advantage of reflection, you only need to follow a coding style that matches the design pattern. This is an important feature of JavaBeans. It means that you don't have to do anything more than code your methods using a simple convention. If you do, your Beans will automatically support introspection without you having to write any extra code. Design patterns are explained in more detail later in the chapter.

This technique may not be sufficient or suitable for every Bean. Instead, you can choose to implement a BeanInfo class which provides descriptive information about its associated Bean explicitly. This is obviously more work than using the default behavior, but it might be necessary to describe a complex Bean properly. It is important to note that the BeanInfo class is separate from the Bean that it is describing. This is done so that it is not necessary to carry the baggage of the BeanInfo within the Bean itself.

If you're writing a development tool, an Introspector class is provided as part of the Beans class library. You don't have to write the code to accomplish the analysis, and every tool vendor uses the same technique to analyze a Bean. This is important to us as programmers because we want to be able to choose our development tools and know that the properties, methods, and events that are exposed for a given component will always be the same.

### Customization

When you are using a visual development tool to assemble components into applications, you will be presented with some sort of user interface for customizing Bean attributes. These attributes may affect the way the Bean operates or the way it looks on the screen. The application tool you use will be able to determine the properties that a Bean supports and build a property sheet dynamically. This property sheet will contain editors for each of the properties supported by the Bean, which you can use to customize the Bean to your liking. The Beans class library comes with a number of property editors for common types such as float, boolean, and String. If you are using custom classes for properties, you will have to create custom property editors to associate with them.

In some cases the default property sheet that is created by the development tool will not be good enough. You may be working with a Bean that is just too complex to customize easily using the default sheet. Beans developers have the option of creating a customizer that can help the user to customize an instance of their Bean. You can even create smart wizards that guide the user through the customization process.

Customizers are also kept separate from the Bean class so that it is not a burden to the Bean when it is not being customized. This idea of separation is a common theme in the JavaBeans architecture. A Bean class only has to implement the functionality it was designed for; all other supporting features are implemented separately.

### Persistence

It is necessary that Beans support a large variety of storage mechanisms. This way, Beans can participate in the largest number of applications. The simplest way to support persistence is to take advantage of Java Object Serialization. This is an automatic mechanism for saving and restoring the state of an object. Java Object Serialization is the best way to make sure that your Beans are fully portable, because you take advantage of a standard feature supported by the core Java platform. This, however, is not always desirable. There may be cases where you want your Bean to use other file formats or mechanisms to save and restore state. In the future, JavaBeans will support an alternative externalization mechanism that will allow the Bean to have complete control of its persistence mechanism.

### Design-Time vs. Run-Time

JavaBeans components must be able to operate properly in a running application as well as inside an application development environment. At design-time the component must provide the design information necessary to edit its properties and customize its behavior. It also has to expose its methods and events so that the design tool can write code that interacts with the Bean at run-time. And, of course, the Bean must support the run-time environment.

### Visibility

There is no requirement that a Bean be visible at run-time. It is perfectly reasonable for a Bean to perform some function that does not require it to present an interface to the user; the Bean may be controlling access to a specific device or data feed. However, it is still necessary for this type of component to support the visual application builder. The component can have properties, methods, and events, have persistent state, and interact with other Beans in a larger application. An "invisible" run-time Bean may be shown visually in the application development tool, and may provide custom property editors and customizers.

### Multithreading

The issue of multithreading is no different in JavaBeans than it is in conventional Java programming. The JavaBeans architecture doesn't introduce any new language constructs or classes to deal with threading. You have to assume that your code will be used in a multithreaded application. It is your responsibility to make sure your Beans are thread-safe. Java makes this easier than in most languages, but it still requires some careful planning to get it right. Remember, thread-safe means that your Bean has anticipated its use by more than one thread at a time and has handled the situation properly.

### Security

Beans are subjected to the same security model as standard Java programs. You should assume that your Bean is running in an untrusted applet. You shouldn't make any design decisions that require your Bean to be run in a trusted environment. Your Bean may be downloaded from the World Wide Web into your browser as part of someone else's applet. All of the security restrictions apply to Beans, such as denying access to the local file system, and limiting socket connections to the host system from which the applet was downloaded.

If your Bean is intended to run only in a Java application on a single computer, the Java security constraints do not apply. In this case you might allow your Bean to behave differently. Be careful, because the assumptions you make about security could render your Bean useless in a networked environment.

## Using Design Patterns

The JavaBeans architecture makes use of patterns that represent standard conventions for names, and type signatures for collections of methods and interfaces. Using coding standards is always a good idea because it makes your code easier to understand, and therefore easier to maintain. It also makes it easier for another programmer to understand the purpose of the methods and interfaces used by your component. In the JavaBeans architecture, these patterns have even more significance. A set of simple patterns are used by the default introspection mechanism to analyze your Bean and determine the properties, methods, and events that are supported. These patterns allow the visual development tools to analyze your Bean and use it in the application being created. The following code fragment shows one such pattern:

public void setTemperatureColor(Color newColor)

{

. . .

}

public Color getTemperatureColor()

{

. . .

}

These two methods together use a pattern that signifies that the Bean contains a property named TemperatureColor of type Color. No extra development is required to expose the property. The various patterns that apply to Beans development will be pointed out and discussed throughout this book. I'll identify each pattern where the associated topic is being discussed.

**NOTE:** The use of the term "design pattern" here may be confusing to some readers. This term is commonly used to describe the practice of documenting a reusable design in object-oriented software. This is not entirely different than the application of patterns here. In this case, the design of the component adheres to a particular convention, and this convention is reused to solve a particular problem.

As mentioned earlier, this convention is not a requirement. You can implement a specific BeanInfo class that fully describes the properties, methods, and events supported by your Bean. In this case, you can name your methods anything you please.

## JavaBeans vs. ActiveX

JavaBeans is certainly not the first component architecture to come along. Microsoft's ActiveX technology is based upon COM, their component object model. ActiveX offers an alternative component architecture for software targeted at the various Windows platforms. So how do you choose one of these technologies over the other? Organizational, cultural, and technical issues all come into play when making this decision. ActiveX and JavaBeans are not mutually exclusive of each otherMicrosoft has embraced Java technology with products like Internet Explorer and Visual J++, and Sun seems to have recognized that the desktop is dominated by Windows and has targeted Win32 as a strategic platform for Java. It is not in anyone's best interest to choose one technology to the exclusion of another. Both are powerful component technologies. I think we should choose a technology because it supports the work we are doing, and does so in a way that meets the needs of the customer.

The most important question is how Beans will be used by containers that are designed specifically to contain ActiveX controls. Certainly, all Beans will not also be ActiveX controls by default. To address the need to integrate Beans into the world of ActiveX, an ActiveX Bridge is available that maps the properties, methods, and events exposed by the Bean into the corresponding mechanisms in COM. This topic is covered in detail in Chapter 11, ActiveX.

# Integrating JavaScript and Java

***by Rick Darnell***

**CONTENTS**

* [Setting the Stage](http://docs.rinet.ru/WebPub/ch38.htm#SettingtheStage)
* [Communicating with Java](http://docs.rinet.ru/WebPub/ch38.htm#CommunicatingwithJava)
  + [Calling Java Methods](http://docs.rinet.ru/WebPub/ch38.htm#CallingJavaMethods)
  + [Controlling Java Applets](http://docs.rinet.ru/WebPub/ch38.htm#ControllingJavaApplets)
* [Communicating with JavaScript](http://docs.rinet.ru/WebPub/ch38.htm#CommunicatingwithJavaScript)
  + [Java and JavaScript Values](http://docs.rinet.ru/WebPub/ch38.htm#JavaandJavaScriptValues)
  + [Looking at the JavaScript Window](http://docs.rinet.ru/WebPub/ch38.htm#LookingattheJavaScriptWindow)
  + [Using JavaScript Methods in Java](http://docs.rinet.ru/WebPub/ch38.htm#UsingJavaScriptMethodsinJava)
* [Summary](http://docs.rinet.ru/WebPub/ch38.htm#Summary)

In [Chapters 34](http://docs.rinet.ru/WebPub/ch34.htm) through [37](http://docs.rinet.ru/WebPub/ch37.htm), you learned about how to make Java and JavaScript a part of your Web pages. Standing alone, they are significant developments in their capability to stretch the behavior of your pages far beyond what was ever imagined for the World Wide Web.

They can become even more powerful when harnessed together. As you'll recall from earlier discussions, although Java is powerful enough to add animation, sound and other features within the confines of an applet, it's very cumbersome to directly interact with an HTML page. JavaScript isn't big or powerful enough to match Java's programming power, but it is uniquely suited to work directly with the elements that comprise an HTML document. By combining the best features of both, your applet can interact with your Web page, offering a new level of interactivity for Java and JavaScript.

## Setting the Stage

In order for Java and JavaScript to interact on your Web pages, they both have to be active and enabled in the user's browser. To make sure both features are active in Netscape Navigator, follow these simple steps:

1. From the menu bar, choose Options | Network Preferences.
2. From the Preferences box, select the Languages tab (see Figure 38.1).
3. Both Java and JavaScript are enabled by default. If this has changed, make sure both boxes are selected.

[**Figure 38.1 :** *The Languages tab from Network Preferences controls whether Java applets and JavaScript commands are processed for HTML documents*.](http://docs.rinet.ru/WebPub/f38-1.gif)

The process to make sure both languages are active in Microsoft Internet Explorer is similar to the steps for Navigator.

1. From the menu bar, choose View | Options.
2. From the Preferences box, select the Security tab (see Figure 38.2).
3. Make sure Enable Java Programs is selected. The scripting languages available in Internet Explorer, JavaScript, and Visual Basic Script are automatically enabled. There is no way to disable them.

[**Figure 38.2 :** *Internet Explorer controls which language features are enabled from the Security tab in the Options dialog box*.](http://docs.rinet.ru/WebPub/f38-2.gif)

Netscape Navigator also includes a Java Console for displaying applet-generated messages (see Figure 38.3). In addition to system messages such as errors and exceptions, it is where any messages generated by the applet using the java.lang.System package, including System.out.println, are displayed. To display the console, select Options | Show Java Console from the menu bar.

[**Figure 38.3 :** *The Java Console displays any system messages generated by the applet*.](http://docs.rinet.ru/WebPub/f38-3.gif)

Microsoft Internet Explorer can show the results of system messages also but not in real time like the Navigator Java Console. All messages are saved in javalog.txt in C:\Windows\Java. To make sure this feature is active, select View | Options. In the Advanced tab of the dialog box that appears, make sure that the Java Logging box is selected.

## Communicating with Java

The first and most commonly used feature of communication is to modify applet behavior from JavaScript. This is really quite easy to do with the right information, and allows your applet to respond to events on the HTML page, including interaction with forms. JavaScript-to-Java syntax is identical to other JavaScript object syntax, so if you're already familiar with this scripting language, adding Java control is an easy step.

### Calling Java Methods

Using the new JavaScript Packages object, JavaScript can invoke native Java methods directly.

|  |
| --- |
| **Note** |
| Groups of related classes are combined in a construct called a package. Classes from a package are usable by outside classes by using the import command.  An example in all applets is the java package. One section of the package, java.awt.Graphics, is imported into every applet to give the paint method the additional methods it needs to add items to the applet screen. Because all applets are outside of the java package, its classes or subsets of those classes are imported into the applet for local use. |

|  |
| --- |
| **Note** |
| Invoking native Java methods from JavaScript is only possible within Netscape Navigator 3.0 or later. It doesn't work on Microsoft Internet Explorer 3.0. |

The syntax to call a Java package directly is

[Packages.]*packageName.className.methodName*

The object name is optional for the three default packages-java, sun, and netscape. These three can be referenced by their package name alone:

java.*className.methodName*

sun.*className.methodName*

netscape.*className.methodName*

Together with the package name, the object and class names can result in some unwieldy and error-prone typing. This is why you can also create new variables using the Package product:

var System = Packages.java.lang.System;

System.out.println("Hello from Java in JavaScript.");

### Controlling Java Applets

Controlling an applet with a script is a fairly easy matter, but it does require a knowledge of the applet you're working with. Any public variable, method, or property within the applet is accessible through JavaScript.

|  |
| --- |
| **Tip** |
| If you're changing the values of variables within an applet, the safest way is to create a new method within the applet for the purpose. This method can accept the value from JavaScript, perform any error checking, then pass the new value along to the rest of the applet. This helps prevent unexpected behavior or applet crashes. |

You need to know which methods, properties and variables are public. Only the public items in an applet are accessible to JavaScript.

|  |
| --- |
| **Tip** |
| There are two public methods which are common to all applets which you can always use-start and stop. These provide a handy means to control when the applet is active and running. |

|  |
| --- |
| **Note** |
| There are five basic activities common to all applets, as opposed to one basic activity for applications. An applet has more activities to correspond to the major events in its life cycle on the user's browser.  None of the activities have any definitions. You must override the methods with a subclass within your applet.  Initialization-Occurs after the applet is first loaded. This can include creating objects, setting state variables, and loading images.  Starting-After initialization or stopping, an applet is started. The difference between initialization and starting is that initialization only happens once, while starting can occur many times.  Painting-This method is how the applet actually gets information to the screen, from simple lines and text to images and colored backgrounds. Painting can occur a lot of times in the course of an applets life.  Stopping-Stopping suspends the applet execution and stops it from using system resources. This can be important since an applet continues to run even after a user leaves the page.  Destroying-This is the extreme form of stop. Destroying an applet begins a clean-up process in which running threads are terminated and objects are released. |

With this information in hand, getting started begins with the <APPLET> tag. It helps to give a name to your applet to make JavaScript references to it easier to read This isn't absolutely necessary as JavaScript creates an array of applets when the page is loaded. However, it does make for a much more readable page.

<APPLET CODE="UnderConstruction" NAME="AppletConstruction" WIDTH=60 HEIGHT=60>

</APPLET>

To use a method of the applet from JavaScript, use the following syntax:

document.*appletName.methodOrProperty*

|  |
| --- |
| **Tip** |
| Netscape Navigator 3.0 includes an applets array which is used to reference all of the applets on a page. These are used according to the following syntax.  document.applets[index].methodOrProperty document.applets[appletName].methodOrProperty  These two methods also identify the applet you want to control, but the method using the applet's name without the applets array is the easiest to read and requires the least amount of typing.  Like the other arrays, a property of applets is length, which returns how many applets are in the document.  This array of applets is not currently available in the Microsoft Internet Explorer implementation of JavaScript. |

One of the easy methods of controlling applet behavior is starting and stopping its execution. This can be accomplished using the start and stop methods-common to every applet. Use a form and two buttons to add the functions to your Web page (see Figure 38.4):

<FORM>

<INPUT TYPE="button" VALUE="Start" onClick="document.*appletName*.start()">

<INPUT TYPE="button" VALUE="Stop" onClick="document.*appletName*.stop()">

</FORM>

[**Figure 38.4 :** *One of the simplest methods of controlling an applet is to use buttons to start and stop it*.](http://docs.rinet.ru/WebPub/f38-4.gif)

You can also call other methods, depending on their visibility to the world outside the applet. Any method or variable with a public declaration can be called by JavaScript.

|  |
| --- |
| **Tip** |
| Any variable or method within the applet which doesn't include a specific declaration of scope is protected by default. If you don't see the public declaration, it's not. |

The syntax to call applet methods from JavaScript is simple and can be integrated with browser events such as the preceding button code snippet:

document.*appletName.methodName*(*arg1,arg2,arg3*)

## Communicating with JavaScript

With the addition of a new set of classes provided with Netscape Navigator 3.0, Java can take a direct look at your HTML page through JavaScript objects. This requires the use of the netscape.javascript.JSObject class when the applet is created.

|  |
| --- |
| **Tip** |
| netscape.javascript.JSObject is included with the other class files under the Netscape directory. In Windows, this is \Program Files\Netscape\Navigator\Program\java\classes\java\_30. In order for your Java program to compile, create a folder set called \netscape\javascript elsewhere on your hard drive, such as under the \java\lib folder. Copy the file to the new folder, and make sure your CLASSPATH variable includes C:\java\lib\ in its list. After you restart the computer, the Java compiler should be able to find the new classes. This new package extends the standard Java Object class, so the newly created JSObjects are treated the same as other Java objects are. |

|  |
| --- |
| **Tip** |
| For normal use, the java package is all you need to use. The netscape package includes methods and properties for Java to reach out to JavaScript and HTML and is covered later in this chapter. The last package, sun, includes platform-specific and system utility classes. |

|  |
| --- |
| **Classes in the java Package** |
| There are five subsets of classes within the java package:  lang-These classes and interfaces are the core of the Java language. This subset includes the Runnable interface (used for threading) and the basic data types (Boolean, character, class, integer, object, string, etc.). It also includes the System class, which provides access to system-level behaviors.  util-This group of utility interfaces and classes aren't crucial to running Java, but they provide ways to make programming easier. It includes utilities to generate random numbers, stacks, hash tables, and dates.  awt-The Abstract Windowing Toolkit (also known as Another Windows Toolkit) contains the graphical items to help create user interfaces and other graphical items. It includes interfaces for a layout manager and menu container, along with classes for form elements, colors, keyboard and mouse events, fonts, images, menus, and windows.  Io-Used for passing information in and out of applets and applications, this subset includes classes for sending and receiving input streams and files, not including networking activity (see net).  Net-This subset of classes has the tools and operations for working over a network. This group includes methods and interfaces to handle URLs, URL content, and socket  connections. |

To include the JSObject class as part of your applet, use the import command as you would normally include any other class package.

import netscape.javascript.JSObject;

An important addition is also necessary in the applet tag-MAYSCRIPT. This is a security feature which gives specific permission for the applet to access JavaScript objects.

<APPLET CODE="colorPreview.class" WIDTH=50 HEIGHT=50 NAME="Preview" MAYSCRIPT>

Without it, any attempt to access JavaScript from the applet results in an exception. If you wish to exclude an applet from accessing the page, simply leave out the MAYSCRIPT parameter.

### Java and JavaScript Values

JSObject gives Java the ability to look at and change objects defined through JavaScript. This requires certain assumptions, especially when passing or receiving values from Java. Every JavaScript value is assigned some form from java.lang.Object to ensure compatibility.

* Objects-Any object sent or received from Java remains in its original object wrapper.
* Java numbers-Since JavaScript doesn't support the variety of numerical types as Java (byte, char, short, int, long, float, and double), they lose their specific type and become a basic JavaScript number.

|  |
| --- |
| **Note** |
| A Java float is a 32-bit floating point number. A version for larger numbers or greater precision behind the decimal point is the double, which is 64 bits long. Bytes, shorts, ints, and longs are all integers of various bit lengths, beginning with 8 bits for the byte and going up to 64 bits for the long. A char is a 16-bit number representing a single Unicode character. |

* JavaScript numbers-There's no way to tell what kind of number Java may be receiving from JavaScript. So, all JavaScript numbers are converted to Java floats.
* Booleans and Strings-These are passed essentially unchanged. Java Booleans become JavaScript Booleans and vice versa. The same occurs with strings.

### Looking at the JavaScript Window

In order to get a handle on JavaScript objects, including form items and frames, you must create an object to hold the current Navigator window first. getWindow provides the means.

First, you'll need to create a new: variable of type JSObject:

JSObject jsWin;

Then, using the JSObject class, assign the window to the variable:

jsWin = JSObject.getWindow(this);

This type of work is typically accomplished within the applet's init() method.

After you have a handle on the window, you can start to break it apart into its various components with getMember. This method returns a specific object from the next level of precedence. For example, to get a handle on a form on a Web page with a form called response, the following set of statements can be used:

jsWin = JSObject.getWindow(this);

JSObject jsDoc = (JSObject) jsWin.getMember("document");

JSObject responseForm = (JSObject) jsDoc.getMember("response");

In JavaScript, this form is referred to as window.document.response. Note that each JavaScript object is assigned to its own variable in Java and is not a property of a parent object. The form in Java is contained in responseForm, not jsWin.jsDoc.responseForm.

|  |
| --- |
| **Note** |
| All parts of an HTML document exist in JavaScript in set relationships to each other. This is called instance hierarchy since it works with specific items on the page rather than general classes of items.  At the top of the pyramid is the window object. It is the parent of all other objects. Its children include document, location, history, which share a precedence level. document's children include objects specific to the page, such as forms, links, anchors, and applets.  The Java netscape package recognizes and uses this hierarchy through its getWindow and getMethod methods. getWindow gets the window object (the highest object), while getMethod returns individual members of the next level. |

So far, you've only retrieved broad objects, such as windows and forms. Getting a specific value from JavaScript follows the same principles, although now you need a Java variable of the proper type to hold the results instead of an instance of JSObject.

|  |
| --- |
| **Tip** |
| Don't forget about passing numbers between JavaScript and Java. All JavaScript numbers are converted to a float. You can cast it to another Java type if needed once it's in the applet. |

Using the preceding form, let's say there's a text field (name), a number (idNum), and a checkbox (member). Each of these values is retrieved from JavaScript using the following commands:

jsWin = JSObject.getWindow(this);

JSObject jsDoc = (JSObject) jsWin.getMember("document");

JSObject responseForm = (JSObject) jsDoc.getMember("response");

JSObject nameField = (JSObject) responseForm.getMember("name");

JSOBject idNumField = (JSObject) responseForm.getMember("idNum");

JSOBject memberField = (JSObject) responseForm.getMember("memberField");

String nameValue = (String) nameField.getMember("value");

Float idNumValue = (Float) idNumField.getMember("value");

Boolean memberValue = (Boolean) memberField.getMember("checked");

This chunk of code becomes a bit unwieldy, especially when there are several values needed from JavaScript. If you need to access more than several elements on a page, it helps to create a new method to handle the process.

protected JSObject getElement(String formName, String elementName) {

JSObject jsDoc = (JSObject) JSObject.getWindow().getMember("document");

JSObject jsForm = (JSObject) jsDoc.getMember(formName);

JSObject jsElement = (JSObject) jsElement.gerMember(elementName);

return jsElement;

}

This simple method creates the intervening JSObjects needed to get to the form element, making the retrieval as easy as knowing the form and element name.

To change a JavaScript value, use the JSObject setMember method in Java. The syntax is setMember(*name*, *value*), with the name of the JavaScript object and its new value.

JSObject nameField = getElement("response","name");

nameField.setMember("name","Your Name Here");

This snippet uses the getElement method just defined to get the name element from the response form, and then uses the JSObject method setMember to set its value to Your Name Here. This is equivalent to this.*name* = *newValue* in JavaScript.

The two methods covered in this section (getWindow, getMember), are the basic methods used when interfacing with JavaScript. Together, it makes receiving values from an HTML page by way of JavaScript a straightforward task, even if it is a little cumbersome in the number of statements needed to accomplish it.

#### Getting Values Using Indexes

If your applet is designed to work with a variety of HTML pages which may contain different names for forms and elements, you can use the JavaScript arrays with the JSObject slot methods. If the desired form is always the first to appear on the document and the element is the third, then the form name is forms[0] and the element is elements[2].

After retrieving the document object using getWindow and getMember, use getSlot(*index*) to return a value within it. For example, in an HTML document containing three forms, the second is retrieved into Java using the following commands:

JSOBject jsWin = JSObject.getWindow(this);

JSObject jsDoc = (JSObject) jsWin.getMember("document");

JSObject jsForms = (JSObject) jsDoc.getMember("forms");

JSObject jsForm1 = (JSObject) jsForms.getSlot(1);

Using setSlot, the same process is used to load a value into an array. The syntax is

*JSObject*.setSlot(*index*,*value*);

where the index is an integer and the value is a string, Boolean or float.

|  |
| --- |
| **Tip** |
| The one rule which must stand firm in this case is the placement of the form and elements within it. When the applet is used with more than one document, the forms and elements must be in the same relative place every time to avoid exceptions and unpredictable results. |

### Using JavaScript Methods in Java

The netscape class package provides two methods to call JavaScript methods from within an applet-call and eval. The syntax between the two is slightly different, but the outcome is the same. Note that you need a handle for the JavaScript window before you can use these methods.

There are two ways to invoke these methods. The first uses a specific window instance, while the second uses getWindow to create a JavaScript window just for the expression.

jsWin.*callOrEval*(*arguments*)

JSOBject.getWindow().*callOrEval*(*arguments*)

The call method separates the method from its arguments. This is useful for passing Java values to the JavaScript method. The syntax is call("*method*", *args*), where the method is the name of the method you want to call and the arguments you want to pass are contained in an array.

eval, on the other hand, uses a string which appears identical to the way a method is called within JavaScript. The syntax is eval("*expression*"), where the expression is a complete method name and its arguments, such as document.writeln("Your name here.'"). Including it in the eval expression results in eval("document.writeln(\"Your name here.\");").

|  |
| --- |
| **Tip** |
| To pass quotation marks as quotation marks to JavaScript within a Java string, use the backslash character before each occurrence. |

Now you have a whole set of tools to get from JavaScript to Java and back again. The marriage of these two Web technologies can open up a whole new world of how to interact with your users. Using simple statements and definitions-already a part of both languages-a previously static Web page can communicate with an applet imbedded in it, and in return react to the output of the applet. It's just one more set of capabilities in your toolbox that you can use to meet your users' needs.

## Summary

The last logical step with Java and JavaScript is allowing them to work and communicate with each other. Accessing an applet's methods and public variables from JavaScript and allowing the Java applet to take a look and modify HTML through JavaScript objects extends the capability of both languages, although it still doesn't allow either one to be the complete equal.

As described in [Chapter 36](http://docs.rinet.ru/WebPub/ch36.htm), "Including Java Applets in Your Web Pages," Java and JavaScript are tailored to different types of applications. But by being able to coordinate activities between the two, you can extend the interactivity of your Web pages to more powerful and useful levels.

# Chapter 16

# Integrating JavaScript with Java

**CONTENTS**

* [An Overview of Java](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#AnOverviewofJava)
  + [How Java Works](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#HowJavaWorks)
  + [The Java Language](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#TheJavaLanguage)
  + [Java Objects and Classes](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#JavaObjectsandClasses)
  + [Integrating Java with HTML](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#IntegratingJavawithHTML)
  + [Creating Your Own Java Applets](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#CreatingYourOwnJavaApplets)
* [Installing the Java Development Kit](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#InstallingtheJavaDevelopmentKit)
* [Creating a Simple Java Applet](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#CreatingaSimpleJavaApplet)
* [Compiling and Viewing a Java Applet](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#CompilingandViewingaJavaApplet)
* [Using Java Classes with JavaScript](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#UsingJavaClasseswithJavaScript)
  + [Calling Java Methods](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#CallingJavaMethods)
  + [The applet Object](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#TheTTFONTSIZEFACECourierappletFONTTTFONTSIZEObjectFONT)
  + [Making the Java Applet Accessible](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#MakingtheJavaAppletAccessible)
* [Controlling a Java Applet](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#ControllingaJavaApplet)
* [Calling JavaScript Functions from Java](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#CallingJavaScriptFunctionsfromJava)
  + [Steps for the JavaScript Programmer](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#StepsfortheJavaScriptProgrammer)
  + [Steps for the Java Programmer](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#StepsfortheJavaProgrammer)
* [Workshop Wrap-Up](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#WorkshopWrapUp)
  + [Next Steps](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#NextSteps)
  + [Q&A](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch16.htm#QampABR)

As I mentioned in [Chapter 1](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch1.htm) there's not much relation between Java and JavaScript-except for the name. However, they can both be useful on your Web pages-sometimes at the same time. This chapter begins with an introduction to the Java language and its capabilities.

JavaScript also includes functions to control Java applets, and Java applets can access JavaScript functions. By integrating these two Web languages, you can have the best of both worlds, allowing for many complicated applications.

## An Overview of Java

Let's begin with a tour of the Java language, the development process for Java applets, and the way Java and HTML interact. If you're unfamiliar with Java, this will give you a quick start and help you understand how Java and JavaScript can work together.

|  |
| --- |
| **Note** |
| Java is a complex language-much more so than JavaScript. This chapter introduces Java, but can't explain it in detail. If you want to learn more about Java, consult the online resources listed in appendix C, or pick up a copy of the bestselling *Teach Yourself Java in 21 Days*, by Laura Lemay, also from Sams.net publishing. |

### How Java Works

Some languages, such as C, are compiled into executable programs before they can be run. Others, such as JavaScript, are interpreted directly from the source code. Java is something in between.

A Java program starts out as source code and is then run through the Java compiler. The compiler doesn't produce true machine code, though-instead, it produces code for a *virtual machine*. This virtual machine code can then be interpreted by an implementation of the virtual machine, such as Netscape.

|  |
| --- |
| **Note** |
| To complicate things even more, the latest versions of Netscape and MSIE include Java compilers. Called *just-in-time compilers*, these compile Java virtual machine code into executable code before running the applet. This greatly increases the speed of applets. |

There are actually two types of Java programs:

* A Java *application* is a complete program, such as a Web browser.
* A Java *applet* is a program that is meant to run within a Web browser. All Java programs you can run from the Web are applets. The next sections focus on applets.

### The Java Language

The Java language is similar to C++. You may also notice a similarity to JavaScript; Netscape loosely based the JavaScript syntax on that of Java.

Like JavaScript, Java includes statements and functions. It also includes objects, properties, and methods. You will look at a simple example of a Java applet in the section titled Creating Your Own Java Applets later in this chapter.

### Java Objects and Classes

Java is an object-oriented language-much more so than JavaScript. A Java applet is actually a class; running an applet means creating an instance of the class. Technically, applets are subclasses of the Applet object.

There are also several object classes you can import into a Java program. These provide functions such as graphics, fonts, working with strings, and so on. You'll look at a class later that enables Java to access JavaScript objects.

### Integrating Java with HTML

Unlike JavaScript, Java code is never included in the HTML file itself. Instead, a special HTML tag, <APPLET>, is used to embed the applet in the Web page. Here is a simple example of an applet:

<applet code="Applet1.class" width=55 height=68>

</applet>.

Notice that both opening and closing <APPLET> tags are required. Between them, you can use optional <PARAM> tags to give parameters to the applet. The parameters required depend on the applet. Each <PARAM> tag includes a variable name and value:

<PARAM name=height value=100>

|  |
| --- |
| **Tip** |
| You can also use JavaScript to choose between a Java or non-Java version of a page. Use the Navigator.javaEnabled property, described in [Chapter 4](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch4.htm) "Using Built-In Objects and Custom Objects." |

The next section presents a complete HTML file with an embedded applet. For examples of embedding public domain Java applets into your pages, see [Chapter 19](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch19.htm), "Real-Life Examples IV."

|  |
| --- |
| **Note** |
| Currently, most complicated Java applets are a bit slow. If you include one in your Web page, be sure you test it carefully; you may also wish to warn users that an applet is loading. |

### Creating Your Own Java Applets

To create your own Java applets, you use the Java Development Kit (JDK). This includes the Java compiler, example applets, and an applet viewer. The JDK is available at no charge from Sun.

|  |
| --- |
| **Tip** |
| The JDK is also included on the CD-ROM that accompanies this book, along with a variety of sample applets. |

The JDK is available for the following platforms:

* Windows 95
* Windows NT
* UNIX (SPARC or x86 Solaris only)
* Macintosh (Power Mac only)

|  |
| --- |
| **Note** |
| Currently, there is no version of the JDK for Windows 3.1 or 68000-series Macintosh. You will need to use a different platform to compile Java applets, although you can still view applets in Netscape. If you have a shell account with your Internet provider, you may be able to use the JDK from there. |

The following tasks explain how to download and install the JDK, use it to create a simple Java applet, and compile and view the new applet.

## Installing the Java Development Kit

To download the JDK, visit Sun's Java Web page at the following address:

<http://java.sun.com/>

and follow the appropriate links. At this writing, the latest version of the JDK is 1.02. Once you've downloaded it, the installation process depends on your platform:

* For Windows 95 or NT, start a DOS session. Copy the self-extracting archive file to the directory or drive you wish to install the JDK in, and execute it. It will be unpacked into a java directory with several subdirectories. The JDK needs about 5M of space to unpack.
* For Solaris, the JDK comes in a compressed tar file. You can expand it into the directory of your choice with this command:  
  /usr/bin/zcat JDK-1\_0\_2-solaris2-sparc.tar.Z | tar xf -
* For Macintosh, the JDK is available in MacBinary or hqx format. Expand the file with a utility such as StuffIt, DeHQX, or BinHex4. Run the installer program, which will create a folder on your desktop for the JDK.

## Creating a Simple Java Applet

Now let's try creating a simple Java applet. Listing 16.1 shows a Java applet that displays a message in a large font.

**Listing 16.1. (JavaTest.java) A simple Java applet that displays a message.**

import java.applet.Applet;

import java.awt.Graphics;

import java.awt.Font;

public class JavaTest extends Applet {

Font f = new Font("TimesRoman", Font.BOLD, 60);

public void paint(Graphics g) {

g.setFont(f);

g.drawString("Text from Java", 15, 50);

}

}

Here is a brief explanation of the program. The first three lines are Java import statements. These specify three classes (libraries) used by the applet:

* java.applet.Applet is a standard class that must be imported for any applet.
* java.awt.Graphics is part of the awt API, which handles the GUI for Java. This library includes functions for graphics.
* java.awt.Font includes functions to handle text with fonts.

Next, the class definition begins with the public class JavaTest statement. You're defining a class called JavaTest, which is a subclass of the Applet class (as all applets are).

The next statement, Font f = new Font, defines an instance of the Font object, an object used to store information about a font and size. The new Font object is called f. (Notice the similarity to the new keyword for defining objects in JavaScript.)

The next statement begins the paint function, which is the main program. This also defines a new Graphics object called g. The next two statements are methods of the Graphics object; methods in Java work much like JavaScript.

The g.setFont() method sets the font to be used when drawing text, and the g.drawString() method draws the text on the screen in the applet's area.

## Compiling and Viewing a Java Applet

Now that you have a simple Java applet, you need to compile it and use it in an HTML page. Be sure you have placed the applet source code in a file called JavaTest.java.

First, compiling the applet is simple. Type the following command from the directory where you installed the JDK:

bin\javac JavaTest.java

The compilation process should take only a few seconds. If no errors were found, the compiler won't display any messages. After the compilation, you should have a file in the same directory called JavaTest.class. This is the Java class file that you can use on a Web page.

|  |
| --- |
| **Note** |
| Just about everything in Java is case-sensitive-source files, class files, and the <APPLET> tag. Be sure you use the exact names. |

To test the new applet, you need to embed it in an HTML page. Listing 16.2 shows a simple HTML page that includes the applet created previously. Create this HTML file in the same directory as the class file or copy the class file into its directory.

**Listing 16.2. (JAVA1.htm) An HTML file to test the new Java applet.**

<HTML>

<HEAD>

<TITLE>Simple Java Test</TITLE>

</HEAD>

<BODY>

<h1>test</h1>

<hr>

An applet is included below to display some text.

<hr>

<APPLET CODE="JavaTest.class" WIDTH=450 HEIGHT=125>

</APPLET>

<hr>

applet finished... here's the rest of the page.

</BODY>

</HTML>

The <APPLET> tag in this file includes the filename of the JavaTest class and a width and height. Be sure to include the closing </APPLET> tag, although there may be nothing between the two tags.

Now that you have an HTML file, you need to test it. A simple way to do this is with the *applet viewer*, included with the JDK. You can start the applet viewer by typing the following command in the JDK directory:

bin\appletviewer file:\Java\JavaTest.html

|  |
| --- |
| **Tip** |
| You may need to modify this command to work on your system. You can also add the java\bin directory to your path to avoid typing bin\ before each command. |

Although you started the applet viewer from a command line, the actual output is shown in a graphical window. Figure 16.2 shows the applet viewer in action, using the example applet.

[**Figure 16.1 :** *The Java applet viewer in action*.](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/f16-1.gif)

Notice that the applet viewer shows the output of the Java applet, but does not include the text from the HTML file itself. This may be useful when you're debugging a Java applet. To see the entire HTML file including the applet, you'll need to use a browser, such as Netscape.

To test the applet in Netscape, simply use the Open command to load the HTML file created previously. Figure 16.2 shows the applet example as viewed by Netscape.

[**Figure 16.2 :** *The Java applet in action, as viewed by Netscape*.](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/f16-2.gif)

|  |
| --- |
| **Note** |
| Once you've created and debugged a Java applet, you'll probably want to publish it on the Web. To do this, simply place the class file for the applet in the same directory as the HTML file on the Web server. |

## Using Java Classes with JavaScript

In Navigator 3.0b4 Netscape introduced a new feature called LiveConnect, which provides the following capabilities:

* JavaScript programs can communicate with and control Java applets.
* Java applets can access JavaScript objects, properties, and methods.
* JavaScript programs can control Netscape plug-ins.

You will look at the first capability in this section: accessing Java classes (applets) from within JavaScript. You will look at the opposite method in the section titled Calling JavaScript functions from Java, later in this chapter.

### Calling Java Methods

You can call Java methods directly from JavaScript. This means you can treat methods as if they are JavaScript statements themselves. For example, this statement prints a message to the Java console:

java.lang.System.err.println("This is a test.");

This will be most useful if you are an experienced Java programmer. If you are not, you can use JavaScript to take advantage of features of existing Java applets, as described in the next section.

### The applet Object

Each Java applet you embed in a Web page is made available to JavaScript as an applet object, with the same name as the applet's class name. The applet object resides in the object hierarchy under the document object. For example, a Java applet called Scroll would be accessed through an object called document.Scroll.

The objects, properties, and methods of the applet are then available to JavaScript, provided the Java programmer has made them public. You will use this technique to control a Java applet in the task later in this section.

|  |
| --- |
| **Note** |
| There is an exception to the rule: any Java method that communicates over the network can't be called from JavaScript. This limitation exists for security reasons. |

### Making the Java Applet Accessible

From the Java programmer's point of view, there are a few things that need to be done to make an applet accessible to JavaScript:

* Define the methods you want to make accessible as public methods.
* Be sure the Netscape package is included in your CLASSPATH environment variable when the applet is compiled.

Once you've made sure of these things, you should be able to access the applet from within JavaScript. The next section gives an example of an applet that is controllable by JavaScript.

## Controlling a Java Applet

Let's create a Java applet that can be manipulated from within JavaScript. Listing 16.3 shows the Java source code. This is an expanded version of the example in Listing 16.1.

**Listing 16.3. (ControlJava.java)A Java applet that can be controlled via JavaScript.**

import java.applet.Applet;

import java.awt.Graphics;

import java.awt.Font;

public class ControlJava extends Applet {

Font f = new Font("TimesRoman", Font.BOLD, 60);

String Message;

public void init() {

Message = new String("Java Test");

}

public void SetMessage(String MsgText) {

Message = MsgText;

repaint();

}

public void paint(Graphics g) {

g.setFont(f);

g.drawString(Message, 15, 50);

}

}

This applet now includes a SetMessage() method to change the text in the display. Listing 16.4 shows the HTML and JavaScript document used to control the applet.

**Listing 16.4. (CJAVA.htm) The JavaScript program to control the Java applet.**

<HTML>

<HEAD>

<TITLE>Control a Java Applet</TITLE>

</HEAD>

<BODY>

<H1>Control a Java Applet</H1>

<HR>

The Java applet below displays text in a large font. You can enter

new text to display in the form below, and JavaScript will call the

Java applet to change the text.

<HR>

<FORM NAME="form1">

<INPUT TYPE="TEXT" NAME="text1">

<INPUT TYPE="BUTTON" VALUE="Change Text"

onClick="document.ControlJava.SetMessage(document.form1.text1.value);">

</FORM>

<HR>

<APPLET NAME="ControlJava" CODE="ControlJava.class" WIDTH=450 HEIGHT=125>

</APPLET>

<HR>

End of page.

</BODY>

</HTML>

This uses a simple event handler to call the Java applet's SetMessage() method. The string you enter in the text field is passed to the applet and displayed in place of the original string. Figure 16.3 shows this application in action after the text has been changed.

[**Figure 16.3 :** *JavaScript has changed the text by accessing the Java applet*.](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/f16-3.gif)

## Calling JavaScript Functions from Java

It's also possible to call JavaScript functions, and access JavaScript objects and properties, from within Java. This enables you to use JavaScript's unique capabilities, such as reading the values of form elements, in powerful Java applications.

To make JavaScript functions accessible from Java, you need to configure things both in JavaScript and in Java. You'll look at the required steps in the next sections.

### Steps for the JavaScript Programmer

It's possible for a Java applet to do things you don't want it to, so you must give permission for it to access your JavaScript program and objects. To do this, add the MAYSCRIPT attribute to the <APPLET> tag that embeds the applet:

<APPLET CODE="Script.class" NAME="TestApp" MAYSCRIPT>

</APPLET>

### Steps for the Java Programmer

For the Java programmer, there are also some important steps before you can access JavaScript. First, you need to include the netscape.javascript package, which provides these functions, in your imported classes:

import netscape.javascript.\*

Next, you need to create a handle for the JavaScript window. To do this, define a variable of type JSObject and use the getWindow method to assign it:

JSObject js = new JSObject;

js = JSObject.getWindow(this);

#### Accessing JavaScript Objects

Once you have a handle for the JavaScript window, you can get the objects you need to access. To do this, you need to call the getMember method for each property. For example, to make the text1 field on the form1 form accessible:

js = JSObject.getWindow(this);

JSObject document = (JSObject) js.getMember("document");

JSObject form1 = (JSObject) document.getMember("form1");

JSObject text1 = (JSObject) form1.getMember("text1");

You are creating an object of type JSObject for the document, then for each object underneath it.

#### Calling JavaScript Functions

You can also call JavaScript functions and methods from within Java, using the same technique. The two methods you use for this purpose are call and eval. For example, this statement calls a JavaScript method to display an alert message:

js = JSObject.getWindow(this);

js.call("window.alert('This is a test.');");

## Workshop Wrap-Up

In this chapter you learned about Java and how it can work with JavaScript:

* The basics of how the Java language and compiler work
* Where to get the Java Development Kit (JDK) and how to install it
* How to create and compile a simple Java applet
* How to embed a Java applet in an HTML page
* How to use JavaScript and LiveConnect to control a Java applet
* Using JavaScript objects, functions, and methods from within a Java applet

### Next Steps

You can continue exploring JavaScript and Java with one of the following chapters:

* To learn how to combine JavaScript with other popular Web languages-CGI and SSI-turn to [Chapter 17](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch17.htm), "Combining JavaScript, CGI, and SSI."
* To learn about Microsoft Internet Explorer, ActiveX, and VBScript, turn to [Chapter 18](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch18.htm), "Using ActiveX and Microsoft Internet Explorer."
* To see examples of using Java applets and controlling them via JavaScript, see [Chapter 19](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch19.htm), "Real-Life Examples IV."
* To learn about the future of JavaScript, Java, and other Web languages, turn to [Chapter 20](http://www.ssuet.edu.pk/taimoor/books/1-57521-141-6/ch20.htm), "The Future of JavaScript."

### Q&A

|  |  |
| --- | --- |
| **Q:** | **Can I access third-party Java applets from JavaScript, particularly those that were created without JavaScript in mind?** |
| **A:** | Usually not. The applet must include the Netscape package, which includes the required functions for JavaScript and Java communication. |
| **Q:** | **Can I use more than one Java applet in a Web page?** |
| **A:** | Yes. However, note that this may cause severe slowdowns (and sometimes crashes) in some versions of Netscape. |
| **Q:** | **Judging by the examples in this chapter, Java isn't very similar to JavaScript. Why on earth do they have similar names?** |
| **A:** | JavaScript was originally called LiveScript, but was renamed after Java. Although some of the syntax is based on Java, it's a very different language. |

# Introduction

This book is intended for web designers, entrepreneurs, students, teachers, and anyone who is interested in learning CGI programming. You do not need any programming experience to get started; if you can write HTML, you can write CGI programs. If you have a website, and want to add guestbook forms, counters, shopping carts, or other interactive elements to your site, then this book is for you.

## What is CGI?

"CGI" stands for "Common Gateway Interface." CGI is one method by which a web server can obtain data from (or send data to) databases, documents, and other programs, and present that data to viewers via the web. More simply, a CGI is a program intended to be run on the web. A CGI program can be written in any programming language, but Perl is one of the most popular, and for this book, Perl is the language we'll be using.

## Why learn CGI?

If you're going to create web pages, then at some point you'll want to add a counter, a form to let visitors send you mail or place an order, or something similar. CGI enables you to do that and much more. From mail-forms and counter programs, to the most complex database programs that generate entire websites on-the-fly, CGI programs deliver a broad spectrum of content on the web today.

## Why use this book?

This book will get you up and running in as little as a day, teaching you the basics of CGI programs, the fundamentals of Perl, and the basics of processing forms and writing simple programs. Then we'll move on to advanced topics, such as reading and writing data files, searching for data in files, writing advanced, multi-part forms like order forms and shopping carts, using randomness to spice up your pages, using server-side includes, cookies, and other useful CGI tricks. Things that you've probably thought beyond your reach, things you thought you had to pay a programmer to do . . . all of these are things you can easily write yourself, and this book will show you how.

You can also try it out before buying the book; the first six chapters are available online, free of charge, at http://www.cgi101.com/book/.

## What do you need to get started?

You should already have some experience building web pages and writing HTML. You'll also need Perl and a web server (such as Apache) that is configured to allow you to run your own CGI programs.

The book is written towards CGI programming on Unix, but you can also set up Apache and Perl on Mac OS X and Windows. I've written several online tutorials that will show you how to get started:

* [**Windows XP:**](http://www.cgi101.com/book/connect/winxp.html) how to set up Apache and Perl; how to configure Apache; where to write your programs; differences between CGI programs on XP and Unix
* [**Mac OS X:**](http://www.cgi101.com/book/connect/mac.html) how to configure Apache (which you already have installed); where to write your programs
* [**Unix:**](http://www.cgi101.com/book/connect/index.html) How to upload programs to your Unix-based ISP; [Unix tutorial](http://www.cgi101.com/help/unixhelp.html); where to write your programs; Unix permissions.

If you need an ISP that offers CGI hosting, visit [http://www.cgi101.com/hosting](http://www.cgi101.com/hosting/). CGI101 offers Unix shell access, CGI programming, a MySQL database, and all of the Perl modules used in this book. It's an easy, hassle-free way to get started writing your own CGI programs.

## Working Code

All of the code examples in this book are available on the web at <http://www.cgi101.com/book/>. You can download any or all of them from there, but do try writing the programs yourself first; you'll learn faster that way.

## Conventions Used in this Book

Perl code will be set apart from the text by indenting and use of a fixed-width font:

print "This is a print statement.\n";

Unix shell commands are shown in a bold font: **chmod 755 filename**

Each program in the book is followed by a link to its source code:

http://www.cgi101.com/book/img/doc.gifSource code: http://www.cgi101.com/book/chX/program-cgi.html

In most cases, a link to a working example is also included:

http://www.cgi101.com/book/img/arrow.gifWorking example: http://www.cgi101.com/book/chX/demo.html

Each chapter has its own web page at http://www.cgi101.com/book/chX, where X is the chapter number. The full text of chapters 1-6 are online; other chapters include an index of the CGI programs and HTML forms from that chapter, links to online resources mentioned in that chapter, questions and answers relating to the chapter material, plus any chapter errata.

# Chapter 1: Getting Started

Our programming language of choice for this book is Perl. Perl is a simple, easy to learn language, yet powerful enough to accomplish very difficult and complex tasks. It is widely available, and is probably already installed on your Unix server. You don't need to compile your Perl programs; you simply write your code, save the file, and run it (or have the web server run it). The program itself is a simple text file; the Perl interpreter does all the work. The advantage to this is you can move your program with little or no changes to any machine with a Perl interpreter. The disadvantage is you won't discover any bugs in your program until you run it.

You can write and edit your CGI programs (which are often called *scripts*) either on your local machine or in the Unix shell. If you're using Unix, try **pico** — it's a very simple, easy to use text editor. Just type **pico filename** to create or edit a file. Type **man pico** for more information and help using pico. If you're not familiar with the Unix shell, see Appendix A for a Unix tutorial and command reference.

You can also use a text editor on your local machine and upload the finished programs to the web server. You should either use a plain text editor, such as Notepad (PC) or BBEdit (Mac), or a programming-specific editor that provides some error- and syntax-checking for you. Visit <http://www.cgi101.com/book/editors.html> for a list of some editors you can use to write your CGI programs.

If you use a text editor, be sure to turn off special characters such as "smartquotes." CGI files must be ordinary text.

Once you've written your program, you'll need to upload it to the web server (unless you're using pico and writing it on the server already). You can use any FTP or SCP (secure copy) program to upload your files; a list of some popular FTP and SCP programs can be found at <http://www.cgi101.com/book/connect/>.

It is imperative that you upload your CGI programs as plain text (ASCII) files, and not binary. If you upload your program as a binary file, it may come across with a lot of control characters at the end of the lines, and these will cause errors in your program. You can save yourself a lot of time and grief by just uploading everything as text (unless you're uploading pictures — for example, GIFs or JPEGs — or other true binary data). HTML and Perl CGI programs are not binary, they are plain text.

Once your program is uploaded to the web server, you'll want to be sure to move it to your cgi-bin (or public\_html directory — wherever your ISP has told you to put your CGI programs). Then you'll also need to change the permissions on the file so that it is "executable" (or runnable) by the system. The Unix shell command for this is:

**chmod 755 filename**

This sets the file permissions so that you can read, write, and execute the file, and all other users (including the webserver) can read and execute it. See Appendix A for a full description of **chmod** and its options.

Most FTP and SCP programs allow you to change file permissions; if you use your FTP client to do this, you'll want to be sure that the file is readable and executable by everyone, and writable only by the owner (you).

One final note: Perl code is case-sensitive, as are Unix commands and filenames. Please keep this in mind as you write your first programs, because in Unix "perl" is not the same as "PERL".

## What Is This Unix Shell?

It's a command-line interface to the Unix machine — somewhat like DOS. You have to use a Telnet or SSH (secure shell) program to connect to the shell; see <http://www.cgi101.com/class/connect.html> for a list of some Telnet and SSH programs you can download. Once you're logged in, you can use shell commands to move around, change file permissions, edit files, create directories, move files, and much more.

If you're using a Unix system to learn CGI, you may want to stop here and look at Appendix A to familiarize yourself with the various shell commands. Download a Telnet or SSH program and login to your shell account, then try out some of the commands so you feel comfortable navigating in the shell.

Throughout the rest of this book you'll see Unix shell commands listed in bold to set them apart from HTML and CGI code. If you're using a Windows server, you can ignore most of the shell commands, as they don't apply.

## Basics of a Perl Program

You should already be familiar with HTML, and so you know that certain things are necessary in the structure of an HTML document, such as the <head> and <body> tags, and that other tags like links and images have a certain allowed syntax. Perl is very similar; it has a clearly defined syntax, and if you follow those syntax rules, you can write Perl as easily as you do HTML.

The first line of your program should look like this:

#!/usr/bin/perl -wT

The first part of this line, #!, indicates that this is a script. The next part, /usr/bin/perl, is the location (or *path*) of the Perl interpreter. If you aren't sure where Perl lives on your system, try typing **which perl** or **whereis perl** in the shell. If the system can find it, it will tell you the full path name to the Perl interpreter. That path is what you should put in the above statement. (If you're using ActivePerl on Windows, the path should be /perl/bin/perl instead.)

The final part contains optional flags for the Perl interpreter. Warnings are enabled by the -w flag. Special user input taint checking is enabled by the -T flag. We'll go into taint checks and program security later, but for now it's good to get in the habit of using both of these flags in all of your programs.

You'll put the text of your program after the above line.

## Basics of a CGI Program

A CGI is simply a program that is called by the webserver, in response to some action by a web visitor. This might be something simple like a page counter, or a complex form-handler. Shopping carts and e-commerce sites are driven by CGI programs. So are ad banners; they keep track of who has seen and clicked on an ad.

CGI programs may be written in *any* programming language; we're just using Perl because it's fairly easy to learn. If you're already an expert in some other language and are just reading to get the basics, here it is: if you're writing a CGI that's going to generate an HTML page, you must include this statement somewhere in the program before you print out anything else:

print "Content-type: text/html\n\n";

This is a content-type header that tells the receiving web browser what sort of data it is about to receive — in this case, an HTML document. If you forget to include it, or if you print something else before printing this header, you'll get an "Internal Server Error" when you try to access the CGI program.

## Your First CGI Program

Now let's try writing a simple CGI program. Enter the following lines into a new file, and name it "first.cgi". Note that even though the lines appear indented on this page, you do not have to indent them in your file. The first line (#!/usr/bin/perl) should start in column 1. The subsequent lines can start in any column.

**Program 1-1: first.cgi - Hello World Program**

#!/usr/bin/perl -wT

print "Content-type: text/html\n\n";

print "Hello, world!\n";

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch1/first-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch1/first.cgi>

Save (or upload) the file into your web directory, then chmod 755 first.cgi to change the file permissions (or use your FTP program to change them). You will have to do this every time you create a new program; however, if you're editing an existing program, the permissions will remain the same and shouldn't need to be changed again.

Now go to your web browser and type the direct URL for your new CGI. For example:

http://www.cgi101.com/book/ch1/first.cgi

Your actual URL will depend on your ISP. If you have an account on cgi101, your URL is:

http://www.cgi101.com/~youruserid/first.cgi

You should see a web page with "Hello, world!" on it. (If it you get a "Page Not Found" error, you have the URL wrong. If you got an "Internal Server Error", see the "Debugging Your Programs," section at the end of this chapter.)

Let's try another example. Start a new file (or if you prefer, edit your existing first.cgi) and add some additional print statements. It's up to your program to print out all of the HTML you want to display in the visitor's browser, so you'll have to include print statements for every HTML tag:

**Program 1-2: second.cgi - Hello World Program 2**

#!/usr/bin/perl -wT

print "Content-type: text/html\n\n";

print "<html><head><title>Hello World</title></head>\n";

print "<body>\n";

print "<h2>Hello, world!</h2>\n";

print "</body></html>\n";

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch1/second-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch1/second.cgi>

Save this file, adjust the file permissions if necessary, and view it in your web browser. This time you should see "Hello, world!" displayed in a H2-size HTML header.

Now not only have you learned to write your first CGI program, you've also learned your first Perl statement, the print function:

print "somestring";

This function will write out any string, variable, or combinations thereof to the current output channel. In the case of your CGI program, the current output is being printed to the visitor's browser.

The \n you printed at the end of each string is the *newline* character. Newlines are not required, but they will make your program's output easier to read.

You can write multiple lines of text without using multiple print statements by using the here-document syntax:

print <<endmarker;

line1

line2

line3

etc.

endmarker

You can use any word or phrase for the end marker (you'll see an example next where we use "EndOfHTML" as the marker); just be sure that the closing marker matches the opening marker exactly (it is case-sensitive), and also that the closing marker is on a line by itself, with no spaces before or after the marker.

Let's try it in a CGI program:

**Program 1-3: third.cgi - Hello World Program, with here-doc**

#!/usr/bin/perl -wT

print "Content-type: text/html\n\n";

print <<EndOfHTML;

<html><head><title>Test Page</title></head>

<body>

<h2>Hello, world!</h2>

</body></html>

EndOfHTML

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch1/third-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch1/third.cgi>

When a closing here-document marker is on the last line of the file, be sure you have a line break after the marker. If the end-of-file mark is on the same line as the here-doc marker, you'll get an error when you run your program.

## The CGI.pm Module

Perl offers a powerful feature to programmers: add-on modules. These are collections of pre-written code that you can use to do all kinds of tasks. You can save yourself the time and trouble of reinventing the wheel by using these modules.

Some modules are included as part of the Perl distribution; these are called *standard library modules* and don't have to be installed. If you have Perl, you already have the standard library modules.

There are also many other modules available that are not part of the standard library. These are typically listed on the Comprehensive Perl Archive Network (CPAN), which you can search on the web at <http://search.cpan.org>.

The CGI.pm module is part of the standard library, and has been since Perl version 5.004. (It should already be installed; if it's not, you either have a very old or very broken version of Perl.) CGI.pm has a number of useful functions and features for writing CGI programs, and its use is preferred by the Perl community. We'll be using it frequently throughout the book.

Let's see how to use a module in your CGI program. First you have to actually include the module via the use command. This goes after the #!/usr/bin/perl line and before any other code:

use CGI qw(:standard);

Note we're not doing use CGI.pm but rather use CGI. The .pm is implied in the use statement. The qw(:standard) part of this line indicates that we're importing the "standard" set of functions from CGI.pm.

Now you can call the various module functions by typing the function name followed by any arguments:

functionname(arguments)

If you aren't passing any arguments to the function, you can omit the parentheses.

A *function* is a piece of code that performs a specific task; it may also be called a *subroutine* or a *method.* Functions may accept optional *arguments* (also called *parameters*), which are values (strings, numbers, and other variables) passed into the function for it to use. The CGI.pm module has many functions; for now we'll start by using these three:

header;

start\_html;

end\_html;

The header function prints out the "Content-type" header. With no arguments, the type is assumed to be "text/html". start\_html prints out the <html>, <head>, <title> and <body> tags. It also accepts optional arguments. If you call start\_html with only a single string argument, it's assumed to be the page title. For example:

print start\_html("Hello World");

will print out the following\*:

<html>

<head>

<title>Hello World</title>

<head>

<body>

You can also set the page colors and background image with start\_html:

print start\_html(-title=>"Hello World",

-bgcolor=>"#cccccc", -text=>"#999999",

-background=>"bgimage.jpg");

Notice that with multiple arguments, you have to specify the name of each argument with -title=>, -bgcolor=>, etc. This example generates the same HTML as above, only the body tag indicates the page colors and background image:

<body bgcolor="#cccccc" text="#999999" background="bgimg.jpg">

The end\_html function prints out the closing HTML tags:

</body>

</html>

So, as you can see, using CGI.pm in your CGI programs will save you some typing. (It also has more important uses, which we'll get into later on.)

|  |
| --- |
| **The Other Way To Use CGI.pm or "There's More Than One Way To Do Things In Perl"**  As you learn Perl you'll discover there are often many different ways to accomplish the same task. CGI.pm exemplifies this; it can be used in two different ways. The first way you've learned already: function-oriented style. Here you must specify qw(:standard) in the use line, but thereafter you can just call the functions directly:  use CGI qw(:standard);  print header;  print start\_html("Hello World");  The other way is object-oriented style, where you create an object (or instance of the module) and use that to call the various functions of CGI.pm:  use CGI; # don't need qw(:standard)  $cgi = CGI->new; # ($cgi is now the object)  print $cgi->header; # function call: $obj->function  print $cgi->start\_html("Hello World");  Which style you use is up to you. The examples in this book use the function-oriented style, but feel free to use whichever style you're comfortable with. |

Let's try using CGI.pm in an actual program now. Start a new file and enter these lines:

**Program 1-4: fourth.cgi - Hello World Program, using CGI.pm**

#!/usr/bin/perl -wT

use CGI qw(:standard);

print header;

print start\_html("Hello World");

print "<h2>Hello, world!</h2>\n";

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch1/fourth-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch1/fourth.cgi>

Be sure to change the file permissions (**chmod 755 fourth.cgi**), then test it out in your browser.

CGI.pm also has a number of functions that serve as HTML shortcuts. For instance:

print h2("Hello, world!");

Will print an H2-sized header tag. You can find a list of all the CGI.pm functions by typing **perldoc CGI** in the shell, or visiting <http://perldoc.perl.org/> and entering "CGI.pm" in the search box.

## Documenting Your Programs

Documentation can be embedded in a program using comments. A comment in Perl is preceded by the # sign; anything appearing after the # is a comment:

**Program 1-5: fifth.cgi - Hello World Program, with Comments**

#!/usr/bin/perl -wT

use CGI qw(:standard);

# This is a comment

# So is this

#

# Comments are useful for telling the reader

# what's happening. This is important if you

# write code that someone else will have to

# maintain later.

print header; # here's a comment. print the header

print start\_html("Hello World");

print "<h2>Hello, world!</h2>\n";

print end\_html; # print the footer

# the end.

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch1/fifth-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch1/fifth.cgi>

You'll notice the first line (#!/usr/bin/perl) is a comment, but it's a special kind of comment. On Unix, it indicates what program to use to run the rest of the script.

There are several situations in Perl where an #-sign is not treated as a comment. These depend on specific syntax, and we'll look at them later in the book.

Any line that starts with an #-sign is a comment, and you can also put comments at the end of a line of Perl code (as we did in the above example on the header and end\_html lines). Even though comments will only be seen by someone reading the source code of your program, it's a good idea to add comments to your code explaining what's going on. Well-documented programs are much easier to understand and maintain than programs with no documentation.

## Debugging Your Programs

A number of problems can happen with your CGI programs, and unfortunately the default response of the webserver when it encounters an error (the "Internal Server Error") is not very useful for figuring out what happened.

If you see the code for the actual Perl program instead of the desired output page from your program, this probably means that your web server isn't properly configured to run CGI programs. You'll need to ask your webmaster how to run CGI programs on your server. And if you ARE the webmaster, check your server's documentation to see how to enable CGI programs.

If you get an Internal Server Error, there's either a permissions problem with the file (did you remember to chmod 755 the file?) or a bug in your program. A good first step in debugging is to use the CGI::Carp module in your program:

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

This causes all warnings and fatal error messages to be echoed in your browser window. You'll want to remove this line after you're finished developing and debugging your programs, because Carp errors can give away important security info to potential hackers.

If you're using the Carp module and are still seeing the "Internal Server Error", you can further test your program from the command line in the Unix shell. This will check the syntax of your program without actually running it:

**perl -cwT fourth.cgi**

If there are errors, it will report any syntax errors in your program:

**% perl -cwT fourth.cgi  
syntax error at fourth.cgi line 5, near "print"  
fourth.cgi had compilation errors.**

This tells you there's a problem on or around line 5; make sure you didn't forget a closing semicolon on the previous line, and check for any other typos. Also be sure you saved and uploaded the file as text; hidden control characters or smartquotes can cause syntax errors, too.

Another way to get more info about the error is to look at the webserver log files. Usually this will show you the same information that the CGI::Carp module does, but it's good to know where the server logs are located, and how to look at them. Some usual locations are /usr/local/etc/httpd/logs/error\_log, or /var/log/httpd/error\_log. Ask your ISP if you aren't sure of the location. In the Unix shell, you can use the tail command to view the end of the log file:

**tail /var/log/apache/error\_log**

The last line of the file should be your error message (although if you're using a shared webserver like an ISP, there will be other users' errors in the file as well). Here are some example errors from the error log:

**[Fri Jan 16 02:06:10 2004] access to /home/book/ch1/test.cgi failed for 205.188.198.46, reason: malformed header from script.  
In string, @yahoo now must be written as \@yahoo at /home/book/ch1/test.cgi line 331, near "@yahoo"  
Execution of /home/book/ch1/test.cgi aborted due to compilation errors.  
[Fri Jan 16 10:04:31 2004] access to /home/book/ch1/test.cgi failed for 204.87.75.235, reason: Premature end of script headers**

A "malformed header" or "premature end of script headers" can either mean that you printed something before printing the "Content-type: text/html" line, or your program died. An error usually appears in the log indicating where the program died, as well.

# Chapter 2: Perl Variables

Before you can proceed much further with CGI programming, you'll need some understanding of Perl variables and data types. A variable is a place to store a value, so you can refer to it or manipulate it throughout your program. Perl has three types of variables: scalars, arrays, and hashes.

## Scalars

A scalar variable stores a single (scalar) value. Perl scalar names are prefixed with a dollar sign ($), so for example, $x, $y, $z, $username, and $url are all examples of scalar variable names. Here's how variables are set:

$foo = 1;

$name = "Fred";

$pi = 3.141592;

In this example $foo, $name, and $pi are scalars. You do not have to declare a variable before using it, but its considered good programming style to do so. There are several different ways to declare variables, but the most common way is with the my function:

my $foo = 1;

my ($name) = "Fred";

my ($pi) = 3.141592;

my simultaneously declares the variables and limits their *scope* (the area of code that can see these variables) to the enclosing code block. (We'll talk more about scope later.) You can declare a variable without giving it a value:

my $foo;

You can also declare several variables with the same my statement:

my ($foo, $bar, $blee);

You can omit the parentheses if you are declaring a single variable, however a list of variables must be enclosed in parentheses.

A scalar can hold data of any type, be it a string, a number, or whatnot. You can also use scalars in double-quoted strings:

my $fnord = 23;

my $blee = "The magic number is $fnord.";

Now if you print $blee, you will get "The magic number is 23." Perl interpolates the variables in the string, replacing the variable name with the value of that variable.

Let's try it out in a CGI program. Start a new program called scalar.cgi:

**Program 2-1: scalar.cgi Print Scalar Variables Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

my $email = "fnord\@cgi101.com";

my $url = "http://www.cgi101.com";

print header;

print start\_html("Scalars");

print <<EndHTML;

<h2>Hello</h2>

<p>

My e-mail address is $email, and my web url is

<a href="$url">$url</a>.

</p>

EndHTML

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch2/scalar-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch2/scalar.cgi>

You may change the $email and $url variables to show your own e-mail address\* and website URL. Save the program, chmod 755 scalar.cgi, and test it in your browser.

You'll notice a few new things in this program. First, there's use strict. This is a standard Perl module that requires you to declare all variables. You don't have to use the strict module, but it's considered good programming style, so it's good to get in the habit of using it.

You'll also notice the variable declarations:

my $email = "fnord\@cgi101.com";

my $url = "http://www.cgi101.com";

Notice that the @-sign in the e-mail address is escaped with (preceded by) a backslash. This is because the @-sign means something special to Perl — just as the dollar sign indicates a scalar variable, the @-sign indicates an array, so if you want to actually use special characters like @, $, and % inside a double-quoted string, you have to precede them with a backslash (\).

A better way to do this would be to use a single-quoted string for the e-mail address:

my $email = 'fnord@cgi101.com';

Single-quoted strings are not interpolated the way double-quoted strings are, so you can freely use the special characters $, @ and % in them. However this also means you can't use a single-quoted string to print out a variable, because

print '$fnord';

will print the actual string "$fnord" . . . not the value stored in the variable named $fnord.

## Arrays

An array stores an ordered list of values. While a scalar variable can only store one value, an array can store many. Perl array names are prefixed with an @-sign. Here is an example:

my @colors = ("red","green","blue");

Each individual item (or element) of an array may be referred to by its index number. Array indices start with 0, so to access the first element of the array @colors, you use $colors[0]. Notice that when you're referring to a single element of an array, you prefix the name with $ instead of @. The $-sign again indicates that it's a single (scalar) value; the @-sign means you're talking about the entire array.

If you want to loop through an array, printing out all of the values, you could print each element one at a time:

my @colors = ("red","green","blue");

print "$colors[0]\n"; # prints "red"

print "$colors[1]\n"; # prints "green"

print "$colors[2]\n"; # prints "blue"

A much easier way to do this is to use a *foreach* loop:

my @colors = ("red","green","blue");

foreach my $i (@colors) {

print "$i\n";

}

For each iteration of the foreach loop, $i is set to an element of the @colors array. In this example, $i is "red" the first time through the loop. The braces {} define where the loop begins and ends, so for any code appearing between the braces, $i is set to the current loop iterator.

Notice we've used my again here to declare the variables. In the foreach loop, my $i declares the loop iterator ($i) and also limits its scope to the foreach loop itself. After the loop completes, $i no longer exists.

We'll cover loops more in Chapter 5.

## Getting Data Into And Out Of Arrays

An array is an ordered list of elements. You can think of it like a group of people standing in line waiting to buy tickets. Before the line forms, the array is empty:

my @people = ();

Then Howard walks up. He's the first person in line. To add him to the @people array, use the push function:

push(@people, "Howard");

Now Sara, Ken, and Josh get in line. Again they are added to the array using the push function. You can push a list of values onto the array:

push(@people, ("Sara", "Ken", "Josh"));

This pushes the list containing "Sara", "Ken" and "Josh" onto the end of the @people array, so that @people now looks like this: ("Howard", "Sara", "Ken", "Josh")

Now the ticket office opens, and Howard buys his ticket and leaves the line. To remove the first item from the array, use the shift function:

my $who = shift(@people);

This sets $who to "Howard", and also removes "Howard" from the @people array, so @people now looks like this: ("Sara", "Ken", "Josh")

Suppose Josh gets paged, and has to leave. To remove the last item from the array, use the pop function:

my $who = pop(@people);

This sets $who to "Josh", and @people is now ("Sara", "Ken")

Both shift and pop change the array itself, by removing an element from the array.

## Finding the Length of Arrays

If you want to find out how many elements are in a given array, you can use the scalar function:

my @people = ("Howard", "Sara", "Ken", "Josh");

my $linelen = scalar(@people);

print "There are $linelen people in line.\n";

This prints "There are 4 people in line." Of course, there's always more than one way to do things in Perl, and that's true here — the scalar function is not actually needed. All you have to do is evaluate the array in a scalar context. You can do this by assigning it to a scalar variable:

my $linelen = @people;

This sets $linelen to 4.

What if you want to print the name of the last person in line? Remember that Perl array indices start with 0, so the index of the last element in the array is actually length-1:

print "The last person in line is $people[$linelen-1].\n";

Perl also has a handy shortcut for finding the index of the last element of an array, the $# shortcut:

print "The last person in line is $people[$#people].\n";

$#arrayname is equivalent to scalar(@arrayname)-1. This is often used in foreach loops where you loop through an array by its index number:

my @colors = ("cyan", "magenta", "yellow", "black");

foreach my $i (0..$#colors) {

print "color $i is $colors[$i]\n";

}

This will print out "color 0 is cyan, color 1 is magenta", etc.

The $#arrayname syntax is one example where an #-sign does not indicate a comment.

## Array Slices

You can retrieve part of an array by specifying the range of indices to retrieve:

my @colors = ("cyan", "magenta", "yellow", "black");

my @slice = @colors[1..2];

This example sets @slice to ("magenta", "yellow").

## Finding An Item In An Array

If you want to find out if a particular element exists in an array, you can use the grep function:

my @results = grep(/pattern/,@listname);

/pattern/ is a regular expression for the pattern you're looking for. It can be a plain string, such as /Box kite/, or a complex regular expression pattern.

/pattern/ will match partial strings inside each array element. To match the entire array element, use /^pattern$/, which anchors the pattern match to the beginning (^) and end ($) of the string. We'll look more at regular expressions in Chapter 13.

grep returns a list of the elements that matched the pattern.

## Sorting Arrays

You can do an alphabetical (ASCII) sort on an array of strings using the sort function:

my @colors = ("cyan", "magenta", "yellow", "black");

my @colors2 = sort(@colors);

@colors2 becomes the @colors array in alphabetically sorted order ("black", "cyan", "magenta", "yellow" ). Note that the sort function, unlike push and pop, does not change the original array. If you want to save the sorted array, you have to assign it to a variable. If you want to save it back to the original array variable, you'd do:

@colors = sort @colors;

You can invert the order of the array with the reverse function:

my @colors = ("cyan", "magenta", "yellow", "black");

@colors = reverse(@colors);

@colors is now ("black", "yellow", "magenta", "cyan").

To do a reverse sort, use both functions:

my @colors = ("cyan", "magenta", "yellow", "black");

@colors = reverse(sort(@colors));

@colors is now ("yellow", "magenta", "cyan", "black").

The sort function, by default, compares the ASCII values of the array elements (see <http://www.cgi101.com/book/ch2/ascii.html> for the chart of ASCII values). This means if you try to sort a list of numbers, you get "12" before "2". You can do a true numeric sort like so:

my @numberlist = (8, 4, 3, 12, 7, 15, 5);

my @sortednumberlist = sort( {$a <=> $b;} @numberlist);

{ $a <=> $b; } is actually a small subroutine, embedded right in your code, that gets called for each pair of items in the array. It compares the first number ($a) to the second number ($b) and returns a number indicating whether $a is greater than, equal to, or less than $b. This is done repeatedly with all the numbers in the array until the array is completely sorted.

We'll talk more about custom sorting subroutines in Chapter 12.

## Joining Array Elements Into A String

You can merge an array into a single string using the join function:

my @colors = ("cyan", "magenta", "yellow", "black");

my $colorstring = join(", ",@colors);

This joins @colors into a single string variable ($colorstring), with each element of the @colors array combined and separated by a comma and a space. In this example $colorstring becomes "cyan, magenta, yellow, black".

You can use any string (including the empty string) as the separator. The separator is the first argument to the join function:

join(separator, list);

The opposite of join is split, which splits a string into a list of values. See Chapter 7 for more on split.

## Array or List?

In general, any function or syntax that works for arrays will also work for a list of values:

my $color = ("red", "green", "blue")[1];

# $color is "green"

my $colorstring = join(", ", ("red", "green", "blue"));

# $colorstring is now "red, green, blue"

my ($first, $second, $third) = sort("red", "green", "blue");

# $first is "blue", $second is "green", $third is "red"

## Hashes

A hash is a special kind of array — an associative array, or paired list of elements. Each pair consists of a string key and a data value.

Perl hash names are prefixed with a percent sign (%). Here's how they're defined:

Hash Name key value

my %colors = ( "red", "#ff0000",

"green", "#00ff00",

"blue", "#0000ff",

"black", "#000000",

"white", "#ffffff" );

This particular example creates a hash named %colors which stores the RGB HEX values for the named colors. The color names are the hash keys; the hex codes are the hash values.

Remember that there's more than one way to do things in Perl, and here's the other way to define the same hash:

Hash Name key value

my %colors = ( red => "#ff0000",

green => "#00ff00",

blue => "#0000ff",

black => "#000000",

white => "#ffffff" );

The => operator automatically quotes the left side of the argument, so enclosing quotes around the key names are not needed.

To refer to the individual elements of the hash, you'll do:

$colors{'red'}

Here, "red" is the key, and $colors{'red'} is the value associated with that key. In this case, the value is "#ff0000".

You don't usually need the enclosing quotes around the value, either; $colors{red} also works if the key name doesn't contain characters that are also Perl operators (things like +, -, =, \* and /).

To print out all the values in a hash, you can use a foreach loop:

foreach my $color (keys %colors) {

print "$colors{$color}=$color\n";

}

This example uses the keys function, which returns a list of the keys of the named hash. One drawback is that keys %hashname will return the keys in unpredictable order — in this example, keys %colors could return ("red", "blue", "green", "black", "white") or ("red", "white", "green", "black", "blue") or any combination thereof. If you want to print out the hash in exact order, you have to specify the keys in the foreach loop:

foreach my $color ("red","green","blue","black","white") {

print "$colors{$color}=$color\n";

}

Let's write a CGI program using the colors hash. Start a new file called colors.cgi:

**Program 2-2: colors.cgi - Print Hash Variables Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

# declare the colors hash:

my %colors = ( red => "#ff0000", green=> "#00ff00",

blue => "#0000ff", black => "#000000",

white => "#ffffff" );

# print the html headers

print header;

print start\_html("Colors");

foreach my $color (keys %colors) {

print "<font color=\"$colors{$color}\">$color</font>\n";

}

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch2/colors-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch2/colors.cgi>

Save it and chmod 755 colors.cgi, then test it in your web browser.

Notice we've had to add backslashes to escape the quotes in this double-quoted string:

print "<font color=\"$colors{$color}\">$color</font>\n";

A better way to do this is to use Perl's qq operator:

print qq(<font color="$colors{$colors}">$color</font>\n);

qq creates a double-quoted string for you. And it's much easier to read without all those backslashes in there.

## Adding Items to a Hash

To add a new value to a hash, you simply do:

$hashname{newkey} = newvalue;

Using our colors example again, here's how to add a new value with the key "purple":

$colors{purple} = "#ff00ff";

If the named key already exists in the hash, then an assignment like this overwrites the previous value associated with that key.

## Determining Whether an Item Exists in a Hash

You can use the exists function to see if a particular key/value pair exists in the hash:

exists $hashname{key}

This returns a true or false value. Here's an example of it in use:

if (exists $colors{purple}) {

print "Sorry, the color purple is already in the hash.<br>\n";

} else {

$colors{purple} = "#ff00ff";

}

This checks to see if the key "purple" is already in the hash; if not, it adds it.

## Deleting Items From a Hash

You can delete an individual key/value pair from a hash with the delete function:

delete $hashname{key};

If you want to empty out the entire hash, do:

%hashname = ();

## Values

We've already seen that the keys function returns a list of the keys of a given hash. Similarly, the values function returns a list of the hash values:

my %colors = (red => "#ff0000", green=> "#00ff00",

blue => "#0000ff", black => "#000000",

white => "#ffffff" );

my @keyslice = keys %colors;

# @keyslice now equals a randomly ordered list of

# the hash keys:

# ("red", "green", "blue", "black", "white")

my @valueslice = values %colors;

# @valueslice now equals a randomly ordered list of

# the hash values:

# ("ff0000", "#00ff00", "#0000ff", "#000000", "#ffffff")

As with keys, values returns the values in unpredictable order.

## Determining Whether a Hash is Empty

You can use the scalar function on hashes as well:

scalar($hashname);

This returns true or false value — true if the hash contains any key/value pairs. The value returned does not indicate how many pairs are in the hash, however. If you want to find that number, use:

scalar keys(%hashname);

Here's an example:

my %colors = (red => "#ff0000", green=> "#00ff00",

blue => "#0000ff", black => "#000000",

white => "#ffffff" );

my $numcolors = scalar(keys(%colors));

print "There are $numcolors in this hash.\n";

This will print out "There are 5 colors in this hash."

# Chapter 3: CGI Environment Variables

Environment variables are a series of hidden values that the web server sends to every CGI program you run. Your program can parse them and use the data they send. Environment variables are stored in a hash named %ENV:

|  |  |
| --- | --- |
| Key | Value |
| DOCUMENT\_ROOT | The root directory of your server |
| HTTP\_COOKIE | The visitor's cookie, if one is set |
| HTTP\_HOST | The hostname of the page being attempted |
| HTTP\_REFERER | The URL of the page that called your program |
| HTTP\_USER\_AGENT | The browser type of the visitor |
| HTTPS | "on" if the program is being called through a secure server |
| PATH | The system path your server is running under |
| QUERY\_STRING | The query string (see GET, below) |
| REMOTE\_ADDR | The IP address of the visitor |
| REMOTE\_HOST | The hostname of the visitor (if your server has reverse-name-lookups on; otherwise this is the IP address again) |
| REMOTE\_PORT | The port the visitor is connected to on the web server |
| REMOTE\_USER | The visitor's username (for .htaccess-protected pages) |
| REQUEST\_METHOD | GET or POST |
| REQUEST\_URI | The interpreted pathname of the requested document or CGI (relative to the document root) |
| SCRIPT\_FILENAME | The full pathname of the current CGI |
| SCRIPT\_NAME | The interpreted pathname of the current CGI (relative to the document root) |
| SERVER\_ADMIN | The email address for your server's webmaster |
| SERVER\_NAME | Your server's fully qualified domain name (e.g. www.cgi101.com) |
| SERVER\_PORT | The port number your server is listening on |
| SERVER\_SOFTWARE | The server software you're using (e.g. Apache 1.3) |

Some servers set other environment variables as well; check your server documentation for more information. Notice that some environment variables give information about your server, and will never change (such as SERVER\_NAME and SERVER\_ADMIN), while others give information about the visitor, and will be different every time someone accesses the program.

Not all environment variables get set. REMOTE\_USER is only set for pages in a directory or subdirectory that's password-protected via a .htaccess file. (See Chapter 20 to learn how to password protect a directory.) And even then, REMOTE\_USER will be the username as it appears in the .htaccess file; it's not the person's email address. There is no reliable way to get a person's email address, short of asking them for it with a web form.

You can print the environment variables the same way you would any hash value:

print "Caller = $ENV{HTTP\_REFERER}\n";

Let's try printing some environment variables. Start a new file named env.cgi:

**Program 3-1: env.cgi - Print Environment Variables Program**

#!/usr/bin/perl -wT

use strict;

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

print header;

print start\_html("Environment");

foreach my $key (sort(keys(%ENV))) {

print "$key = $ENV{$key}<br>\n";

}

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch3/env-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: [http://www.cgi101.com/book/ch3/env.cgi](http://neo.cgi101.com/env.cgi)

Save the file, **chmod 755 env.cgi**, then try it in your web browser. Compare the environment variables displayed with the list on the previous page. Notice which values show information about your server and CGI program, and which ones give away information about you (such as your browser type, computer operating system, and IP address).

Let's look at several ways to use some of this data.

## Referring Page

When you click on a hyperlink on a web page, you're being referred to another page. The web server for the receiving page keeps track of the referring page, and you can access the URL for that page via the HTTP\_REFERER environment variable. Here's an example:

**Program 3-2: refer.cgi - HTTP Referer Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;

print start\_html("Referring Page");

print "Welcome, I see you've just come from

$ENV{HTTP\_REFERER}!<p>\n";

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch3/refer-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch3/> (click on refer.cgi)

Remember, HTTP\_REFERER only gets set when a visitor actually clicks on a link to your page. If they type the URL directly (or use a bookmarked URL), then HTTP\_REFERER is blank. To properly test your program, create an HTML page with a link to refer.cgi, then click on the link:

[Referring Page](http://www.cgi101.com/book/ch3/refer.cgi)

HTTP\_REFERER is not a foolproof method of determining what page is accessing your program. It can easily be forged.

## Remote Host Name, and Hostname Lookups

You've probably seen web pages that greet you with a message like "Hello, visitor from (yourhost)!", where (yourhost) is the hostname or IP address you're currently logged in with. This is a pretty easy thing to do because your IP address is stored in the %ENV hash.

If your web server is configured to do hostname lookups, then you can access the visitor's actual hostname from the $ENV{REMOTE\_HOST} value. Servers often don't do hostname lookups automatically, though, because it slows down the server. Since $ENV{REMOTE\_ADDR} contains the visitor's IP address, you can reverse-lookup the hostname from the IP address using the Socket module in Perl. As with CGI.pm, you have to use the Socket module:

use Socket;

(There is no need to add qw(:standard) for the Socket module.)

The Socket module offers numerous functions for socket programming (most of which are beyond the scope of this book). We're only interested in the reverse-IP lookup for now, though. Here's how to do the reverse lookup:

my $ip = "209.189.198.102";

my $hostname = gethostbyaddr(inet\_aton($ip), AF\_INET);

There are actually two functions being called here: gethostbyaddr and inet\_aton. gethostbyaddr is a built-in Perl function that returns the hostname for a particular IP address. However, it requires the IP address be passed to it in a packed 4-byte format. The Socket module's inet\_aton function does this for you.

Let's try it in a CGI program. Start a new file called rhost.cgi, and enter the following code:

**Program 3-3: rhost.cgi - Remote Host Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

use Socket;

print header;

print start\_html("Remote Host");

my $hostname = gethostbyaddr(inet\_aton($ENV{REMOTE\_ADDR}), AF\_INET);

print "Welcome, visitor from $hostname!<p>\n";

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch3/rhost-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch3/rhost.cgi>

## Detecting Browser Type

The HTTP\_USER\_AGENT environment variable contains a string identifying the browser (or "user agent") accessing the page. Unfortunately there is no standard (yet) for user agent strings, so you will see a vast assortment of different strings. Here's a sampling of some:

DoCoMo/1.0/P502i/c10 (Google CHTML Proxy/1.0)  
Firefly/1.0 (compatible; Mozilla 4.0; MSIE 5.5)  
Googlebot/2.1 (+http://www.googlebot.com/bot.html)  
Mozilla/3.0 (compatible)  
Mozilla/4.0 (compatible; MSIE 4.01; MSIECrawler; Windows 95)  
Mozilla/4.0 (compatible; MSIE 5.0; MSN 2.5; AOL 8.0; Windows 98; DigExt)  
Mozilla/4.0 (compatible; MSIE 5.0; Mac\_PowerPC)  
Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt; Hotbar 4.1.7.0)  
Mozilla/4.0 (compatible; MSIE 6.0; AOL 9.0; Windows NT 5.1)  
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; DigExt)  
Mozilla/4.0 WebTV/2.6 (compatible; MSIE 4.0)  
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US; rv:1.0.2) Gecko/20020924 AOL/7.0  
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US; rv:1.0.2) Gecko/20021120 Netscape/7.01  
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us) AppleWebKit/85 (KHTML, like Gecko) Safari/85  
Mozilla/5.0 (Windows; U; Win98; en-US; m18) Gecko/20010131 Netscape6/6.01  
Mozilla/5.0 (Slurp/cat; slurp@inktomi.com; http://www.inktomi.com/slurp.html)  
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.5a) Gecko/20030718  
Mozilla/5.0 (compatible; Konqueror/3.0-rc3; i686 Linux; 20020913)  
NetNewsWire/1.0 (Mac OS X; Pro; http://ranchero.com/netnewswire/)  
Opera/6.0 (Windows 98; U) [en]  
Opera/7.10 (Linux 2.4.19 i686; U) [en]  
Scooter/3.3

As you can see, sometimes the user agent string reveals what type of browser and computer the visitor is using, and sometimes it doesn't. Some of these aren't even browsers at all, like the search engine robots (Googlebot, Inktomi and Scooter) and RSS reader (NetNewsWire). You should be careful about writing programs (and websites) that do browser detection. It's one thing to collect browser info for logging purposes; it's quite another to design your entire site exclusively for a certain browser. Visitors will be annoyed if they can't access your site because you think they have the "wrong" browser.

That said, here's an example of how to detect the browser type. This program uses Perl's index function to see if a particular substring (such as "MSIE") exists in the HTTP\_USER\_AGENT string. index is used like so:

index(string, substring);

It returns a numeric value indicating where in the string the substring appears, or -1 if the substring does not appear in the string. We use an if/else block in this program to see if the index is greater than -1.

**Program 3-4: browser.cgi - Browser Detection Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;

print start\_html("Browser Detect");

my($ua) = $ENV{HTTP\_USER\_AGENT};

print "User-agent: $ua<p>\n";

if (index($ua, "MSIE") > -1) {

print "Your browser is Internet Explorer.<p>\n";

} elsif (index($ua, "Netscape") > -1) {

print "Your browser is Netscape.<p>\n";

} elsif (index($ua, "Safari") > -1) {

print "Your browser is Safari.<p>\n";

} elsif (index($ua, "Opera") > -1) {

print "Your browser is Opera.<p>\n";

} elsif (index($ua, "Mozilla") > -1) {

print "Your browser is probably Mozilla.<p>\n";

} else {

print "I give up, I can't tell what browser you're using!<p>\n";

}

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch3/browser-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch3/browser.cgi>

If you have several different browsers installed on your computer, try testing the program with each of them.

We'll look more at if/else blocks in Chapter 5.

## A Simple Form Using GET

There are two ways to send data from a web form to a CGI program: GET and POST. These *methods* determine how the form data is sent to the server.

With the GET method, the input values from the form are sent as part of the URL and saved in the QUERY\_STRING environment variable. With the POST method, data is sent as an input stream to the program. We'll cover POST in the next chapter, but for now, let's look at GET.

You can set the QUERY\_STRING value in a number of ways. For example, here are a number of direct links to the env.cgi program:

[http://www.cgi101.com/book/ch3/env.cgi?test1](http://neo.cgi101.com/env.cgi?test1)

[http://www.cgi101.com/book/ch3/env.cgi?test2](http://neo.cgi101.com/env.cgi?test2)

[http://www.cgi101.com/book/ch3/env.cgi?test3](http://neo.cgi101.com/env.cgi?test3)

Try opening each of these in your web browser. Notice that the value for QUERY\_STRING is set to whatever appears after the question mark in the URL itself. In the above examples, it's set to "test1", "test2", and "test3" respectively.

You can also process simple forms using the GET method. Start a new HTML document called envform.html, and enter this form:

**Program 3-5: envform.html - Simple HTML Form Using GET**

<html><head><title>Test Form</title></head>

<body>

<form action="env.cgi" method="GET">

Enter some text here:

<input type="text" name="sample\_text" size=30>

<input type="submit"><p>

</form>

</body></html>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch3/envform.html>

Save the form and upload it to your website. Remember you may need to change the path to env.cgi depending on your server; if your CGI programs live in a "cgi-bin" directory then you should use action="cgi-bin/env.cgi".

Bring up the form in your browser, then type something into the input field and hit return. You'll notice that the value for QUERY\_STRING now looks like this:

sample\_text=whatever+you+typed

The string to the left of the equals sign is the name of the form field. The string to the right is whatever you typed into the input box. Notice that any spaces in the string you typed have been replaced with a +. Similarly, various punctuation and other special non-alphanumeric characters have been replaced with a %-code. This is called *URL-encoding*, and it happens with data submitted through either GET or POST methods.

You can send multiple input data values with GET:

<form action="env.cgi" method="GET">

First Name: <input type="text" name="fname" size=30><p>

Last Name: <input type="text" name="lname" size=30><p>

<input type="submit">

</form>

This will be passed to the env.cgi program as follows:

$ENV{QUERY\_STRING} = "fname=joe&lname=smith"

The two form values are separated by an ampersand (&). You can divide the query string with Perl's split function:

my @values = split(/&/,$ENV{QUERY\_STRING});

split lets you break up a string into a list of strings, splitting on a specific character. In this case, we've split on the "&" character. This gives us an array named @values containing two elements: ("fname=joe", "lname=smith"). We can further split each string on the "=" character using a foreach loop:

foreach my $i (@values) {

my($fieldname, $data) = split(/=/, $i);

print "$fieldname = $data<br>\n";

}

This prints out the field names and the data entered into each field in the form. It does not do URL-decoding, however. A better way to parse QUERY\_STRING variables is with CGI.pm.

## Using CGI.pm to Parse the Query String

If you're sending more than one value in the query string, it's best to use CGI.pm to parse it. This requires that your query string be of the form:

fieldname1=value1

For multiple values, it should look like this:

fieldname1=value1&fieldname2=value2&fieldname3=value3

This will be the case if you are using a form, but if you're typing the URL directly then you need to be sure to use a fieldname, an equals sign, then the field value.

CGI.pm provides these values to you automatically with the param function:

param('fieldname');

This returns the value entered in the fieldname field. It also does the URL-decoding for you, so you get the exact string that was typed in the form field.

You can get a list of all the fieldnames used in the form by calling param with no arguments:

my @fieldnames = param();

## param is NOT a Variable

param is a function call. You can't do this:

print "$p = param($p)<br>\n";

If you want to print the value of param($p), you can print it by itself:

print param($p);

Or call param outside of the double-quoted strings:

print "$p = ", param($p), "<br>\n";

You won't be able to use param('fieldname') inside a here-document. You may find it easier to assign the form values to individual variables:

my $firstname = param('firstname');

my $lastname = param('lastname');

Another way would be to assign every form value to a hash:

my(%form);

foreach my $p (param()) {

$form{$p} = param($p);

}

You can achieve the same result by using CGI.pm's Vars function:

use CGI qw(:standard Vars);

my %form = Vars();

The Vars function is not part of the "standard" set of CGI.pm functions, so it must be included specifically in the use statement.

Either way, after storing the field values in the %form hash, you can refer to the individual field names by using $form{'fieldname'}. (This will not work if you have a form with multiple fields having the same field name.)

Let's try it now. Create a new form called getform.html:

**Program 3-6: getform.html - Another HTML Form Using GET**

<html><head><title>Test Form</title></head>

<body>

<form action="get.cgi" method="GET">

First Name: <input type="text" name="firstname" size=30><br>

Last Name: <input type="text" name="lastname" size=30><br>

<input type="submit"><p>

</form>

</body></html>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch3/getform.html>

Save and upload it to your webserver, then bring up the form in your web browser.

Now create the CGI program called get.cgi:

**Program 3-7: get.cgi Form Processing Program Using GET**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;

print start\_html("Get Form");

my %form;

foreach my $p (param()) {

$form{$p} = param($p);

print "$p = $form{$p}<br>\n";

}

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch3/get-cgi.html>

Save and **chmod 755 get.cgi**. Now fill out the form in your browser and press submit. If you encounter errors, refer back to [Chapter 1](http://www.cgi101.com/book/ch1/text.html) for debugging.

Take a look at the full URL of get.cgi after you press submit. You should see all of your form field names and the data you typed in as part of the URL. This is one reason why GET is not the best method for handling forms; it isn't secure.

## GET is NOT Secure

GET is not a secure method of sending data. Don't use it for forms that send password info, credit card data or other sensitive information. Since the data is passed through as part of the URL, it'll show up in the web server's logfile (complete with all the data). Server logfiles are often readable by other users on the system. URL history is also saved in the browser and can be viewed by anyone with access to the computer. Private information should always be sent with the POST method, which we'll cover in the next chapter. (And if you're asking visitors to send sensitive information like credit card numbers, you should also be using a secure server in addition to the POST method.)

There may also be limits to how much data can be sent with GET. While the HTTP protocol doesn't specify a limit to the length of a URL, certain web browsers and/or servers may.

Despite this, the GET method is often the best choice for certain types of applications. For example, if you have a database of articles, each with a unique article ID, you would probably want a single article.cgi program to serve up the articles. With the article ID passed in by the GET method, the program would simply look at the query string to figure out which article to display:

<a href="article.cgi?id=22">Article Name</a>

We'll be revisiting that idea later in the book. For now, let's move on to Chapter 4 where we'll see how to process forms using the POST method.

# Chapter 4: Processing Forms and Sending Mail

Most forms you create will send their data using the POST method. POST is more secure than GET, since the data isn't sent as part of the URL, and you can send more data with POST. Also, your browser, web server, or proxy server may cache GET queries, but posted data is resent each time.

Your web browser, when sending form data, encodes the data being sent. Alphanumeric characters are sent as themselves; spaces are converted to plus signs (+); other characters — like tabs, quotes, etc. — are converted to "%HH" — a percent sign and two hexadecimal digits representing the ASCII code of the character. This is called URL encoding.

In order to do anything useful with the data, your program must decode these. Fortunately the CGI.pm module does this work for you. You access the decoded form values the same way you did with GET:

$value = param('fieldname');

So you already know how to process forms! You can try it now by changing your getform.html form to method="POST" (rather than method="GET"). You'll see that it works identically whether you use GET or POST. Even though the data is sent differently, CGI.pm handles it for you automatically.

## The Old Way of Decoding Form Data

Before CGI.pm was bundled with Perl, CGI programmers had to write their own form-parsing code. If you read some older CGI books (including the first edition of this book), or if you're debugging old code, you'll probably encounter the old way of decoding form data. Here's what it looks like:

read(STDIN, $buffer, $ENV{'CONTENT\_LENGTH'});

@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {

($name, $value) = split(/=/, $pair);

$value =~ tr/+/ /;

$value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;

$FORM{$name} = $value;

}

This code block reads the posted form data from standard input, loops through the fieldname=value fields in the form, and uses the pack function to do URL-decoding. Then it stores each fieldname/value pair in a hash called %FORM.

This code is deprecated and should be avoided; use CGI.pm instead. If you want to upgrade an old program that uses the above code block, you can replace it with this:

my %FORM;

foreach my $field (param()) {

$FORM{$field} = param($field);

}

Or you could use the Vars function:

use CGI qw(:standard Vars);

my %FORM = Vars();

Either method will replace the old form-parsing code, although keep in mind that this will not work if your form has multiple fields with the same name. We'll look at how to handle those in the next chapter.

## Guestbook Form

One of the first CGI programs you're likely to want to add to your website is a guestbook program, so let's start writing one. First create your HTML form. The actual fields can be up to you, but a bare minimum might look like this:

<form action="post.cgi" method="POST">

Your Name: <input type="text" name="name"><br>

Email Address: <input type="text" name="email"><br>

Comments:<br>

<textarea name="comments" rows="5"

cols="60"></textarea><br>

<input type="submit" value="Send">

</form>

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch4/guestbook1.html>

(Stylistically it's better NOT to include a "reset" button on forms like this. It's unlikely the visitor will want to erase what they've typed, and more likely they'll accidentally hit "reset" instead of "send", which can be an aggravating experience. They may not bother to re-fill the form in such cases.)

Now you need to create post.cgi. This is nearly identical to the get.cgi from last chapter, so you may just want to copy that program and make changes:

**Program 4-1: post.cgi - Form Processing Program Using POST**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;

print start\_html("Thank You");

print h2("Thank You");

my %form;

foreach my $p (param()) {

$form{$p} = param($p);

print "$p = $form{$p}<br>\n";

}

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch4/post-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch4/form.html>

Test your program by entering some data into the fields, and pressing "send" when finished. Notice that the data is not sent in the URL this time, as it was with the GET example.

Of course, this form doesn't actually DO anything with the data, which doesn't make it much of a guestbook. Let's see how to send the data in e-mail.

## Sending Mail

There are several ways to send mail. We'll be using the sendmail program for these examples. If you're using a non-Unix system (or a Unix without sendmail installed), there are a number of third-party Perl modules that you can use to achieve the same effect. See <http://search.cpan.org/> (search for "sendmail") for a list of platform-independent mailers, and Chapter 14 for examples of how to install third-party modules. If you're using ActivePerl on Windows, visit <http://www.cgi101.com/book/ch4/> for a link to more information about sending mail from Windows.

Before you can write your form-to-mail CGI program, you'll need to figure out where the sendmail program is installed on your webserver. (For cgi101.com, it's in /usr/sbin/sendmail. If you're not sure where it is, try doing **which sendmail** or **whereis sendmail**; usually one of these two commands will yield the correct location.)

Since we're using the -T flag for taint checking, the first thing you need to do before connecting to sendmail is set the PATH environment variable:

$ENV{PATH} = "/usr/sbin";

The path should be the directory where sendmail is located; if sendmail is in /usr/sbin/sendmail, then $ENV{PATH} should be "/usr/sbin". If it's in /var/lib/sendmail, then $ENV{PATH} should be "/var/lib".

Next you open a *pipe* to the sendmail program:

open (MAIL, "|/usr/sbin/sendmail -t -oi") or

die "Can't fork for sendmail: $!\n";

The pipe (which is indicated by the | character) causes all of the output printed to that filehandle (MAIL) to be fed directly to the /usr/sbin/sendmail program as if it were standard input to that program. Several flags are also passed to sendmail:

|  |  |  |
| --- | --- | --- |
| -t |  | Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for recipient addresses |
| -oi |  | Ignore dots alone on lines by themselves in incoming messages. |

The -t flag tells sendmail to look at the message headers to determine who the mail is being sent to. You'll have to print all of the message headers yourself:

my $recipient = 'recipient@cgi101.com';

print MAIL "From: sender\@cgi101.com\n";

print MAIL "To: $recipient\n";

print MAIL "Subject: Guestbook Form\n\n";

Remember that you can safely put an @-sign inside a single-quoted string, like 'recipient@cgi101.com', or you can escape the @-sign in double-quoted strings by using a backslash ("sender\@cgi101.com").

The message headers are complete when you print a single blank line following the header lines. We've accomplished this by printing two newlines at the end of the subject header:

print MAIL "Subject: Guestbook Form\n\n";

After that, you can print the body of your message.

Let's try it. Start a new file named guestbook.cgi, and edit it as follows. You don't need to include the comments in the following code; they are just there to show you what's happening.

**Program 4-2: guestbook.cgi - Guestbook Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;

print start\_html("Results");

# Set the PATH environment variable to the same path

# where sendmail is located:

$ENV{PATH} = "/usr/sbin";

# open the pipe to sendmail

open (MAIL, "|/usr/sbin/sendmail -oi -t") or

&dienice("Can't fork for sendmail: $!\n");

# change this to your own e-mail address

my $recipient = 'nullbox@cgi101.com';

# Start printing the mail headers

# You must specify who it's to, or it won't be delivered:

print MAIL "To: $recipient\n";

# From should probably be the webserver.

print MAIL "From: nobody\@cgi101.com\n";

# print a subject line so you know it's from your form cgi.

print MAIL "Subject: Form Data\n\n";

# Now print the body of your mail message.

foreach my $p (param()) {

print MAIL "$p = ", param($p), "\n";

}

# Be sure to close the MAIL input stream so that the

# message actually gets mailed.

close(MAIL);

# Now print a thank-you page

print <<EndHTML;

<h2>Thank You</h2>

<p>Thank you for writing!</p>

<p>Return to our <a href="index.html">home page</a>.</p>

EndHTML

print end\_html;

# The dienice subroutine handles errors.

sub dienice {

my($errmsg) = @\_;

print "<h2>Error</h2>\n";

print "<p>$errmsg</p>\n";

print end\_html;

exit;

}

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch4/guestbook-cgi.html>

Save and chmod the file, then modify your guestbook.html form so that the action points to guestbook.cgi:

<form action="guestbook.cgi" method="POST">

Try testing the form. If the program runs successfully, you'll get e-mail in a few moments with the results of your post. (Remember to change $recipient to your email address!)

## Subroutines

In the guestbook program we used a new structure: a subroutine called "dienice." A subroutine is a user-defined function. You've already used functions like param and start\_html from the CGI.pm module, and built-in functions like shift and pop. You can also define your own custom functions.

In the mail program, the dienice subroutine is only called if the program can't open the pipe to sendmail. Rather than aborting and giving you a server error (or worse, NO error), you want your program to give you some useful data about what went wrong; dienice does that, by printing the error message and closing HTML tags, and exiting the program. We'll be using the dienice subroutine throughout the rest of the book, as a generic catch-all error-handler.

Subroutines are useful for isolating blocks of code that are reused frequently in your program. The structure of a subroutine is as follows:

sub subname {

# your code here

}

The subroutine block starts with the word sub, followed by the name of the subroutine. The code for the subroutine is then enclosed in curly braces { }.

Subroutines can be placed anywhere in your program, though for readability it's usually best to put them at the end, after the main program code.

To invoke a subroutine, enter the subroutine name and an optional list of arguments:

subname;

subname(arguments);

You may prefix the subroutine name with an &-sign:

&subname;

&subname(arguments);

The &-sign is optional. However, we'll be using this syntax throughout the book to differentiate calls to subroutines we've written ourselves. Calls to built-in functions or functions provided by external modules will not have this sign.

Here is an example of a call to a subroutine named "mysub" with three arguments:

&mysub($arg1, "whatever", 23);

The arguments are passed to the subroutine in the special Perl array @\_. You can then assign the elements of that array to special temporary variables, like so:

sub mysub {

my($arg1, $arg2, $arg3) = @\_;

# your code here

}

In this example, the my function limits the scope of $arg1, $arg2 and $arg3 to the mysub subroutine. This keeps your temporary variables visible only to the subroutine itself (where they're actually needed and used), rather than to the entire program (where they're not needed). This also means if you change one of the variables inside your subroutine, the value of the original variable won't change (unless it's a reference, which we'll look at next).

## Passing Arrays and Hashes to Subroutines

When passing an array (or a hash) to a subroutine, the array is expanded into a list of its values. This might be okay if the array is the only argument:

&subname(@array1);

However if you have multiple arguments, you're going to run into problems:

&subname(@array1, $item2, $item3);

sub subname {

my(@ary, $arg2, $arg3) = @\_;

}

In this example, all of the arguments (including $item2 and $item3) are stored in @ary, and $arg2 and $arg3 are undefined. In order to pass the array or hash properly to the subroutine, you need to pass it as a reference, by prefixing the @ (or %) by a backslash:

&subname(\@array1, $item2, \%hash1);

sub subname {

my($arrayref, $arg2, $hashref) = @\_;

}

Now $arrayref is a reference to @array1, $arg2 is whatever the value of $item2 is, and $hashref is a reference to %hash1. To access individual elements of an array reference, instead of using $arrayref[1], you use $arrayref->[1]. Similarly with a hash reference you use $hashref->{key} instead of $hashref{key}.

A *reference* is a pointer to the original variable. If you change the value of an element of an array reference, you're changing the original array's values.

Optionally you could *dereference* the array inside your subroutine by doing:

my @localary = @{$arrayref};

A hash is dereferenced like so:

my %localhash = %{$hashref};

A dereferenced array (or hash) is localized to your subroutine, so you can change the values of @newarray or %newhash without altering the original variables.

You can find out a lot more about references by reading **perldoc perlref** and **perldoc perlreftut** (the Perl reference tutorial).

## Subroutine Return Values

Subroutines can return a value:

sub subname {

# your code here

return $somevalue;

}

If you omit the return statement, then the value returned by the subroutine is the value of the last expression executed in that routine.

If you want to save the return value, be sure to assign it to a variable:

my $result = &subname(arguments);

Subroutines can also return a list:

sub subname {

# your code here

return $value1, $value2, 'foo';

}

Which can then be assigned to a list of variables:

my ($x, $y, $z) = &subname(arguments);

Or an array:

my @x = &subname(arguments);

## Return vs. Exit

You'll notice that our dienice subroutine does not return a value at all, but rather calls the exit function. exit causes the entire program to terminate immediately.

## Sendmail Subroutine

Here is an example of the mail-sending code in a compact subroutine:

sub sendmail {

my ($from, $to, $subject, $message) = @\_;

$ENV{PATH} = "/usr/sbin";

open (MAIL, "|/usr/sbin/sendmail -oi -t") or

&dienice("Can't fork for sendmail: $!\n");

print MAIL "To: $to\n";

print MAIL "From: $from\n";

print MAIL "Subject: $subject\n\n";

print MAIL "$message\n";

close(MAIL);

}

## Sending Mail to More Than One Recipient

If you want to send mail to more than one email address, just add the desired addresses to the $recipient line, separated by commas:

my $recipient = 'recipient1@cgi101.com, recipient2@cgi101.com, recipient3@cgi101.com';

## Defending Against Spammers

When building form-to-mail programs, you need to take precautions to prevent spammers from hijacking your programs to send unwanted e-mail to other recipients. They can do this by writing their own form (or program) to send data to your CGI. If your program prints any of the form fields as mail headers without checking them first, the spammer can insert their own mail headers (and even their own message). The end result: your program becomes a relay for spammers.

The primary defense against this is to not allow the form to specify ANY of the mail headers (such as the From, To, or Subject headers). Note that in our guestbook program, the From, To and Subject headers were all hardcoded in the program.

Of course, it would be nice to have the "From" header show the poster's e-mail address. You could allow this if you validate it first, verifying that it's really an e-mail address and doesn't contain any extra headers. You can validate e-mail addresses by using a regular expression pattern match, which we'll cover in Chapter 13, or by using the [Email::Valid](http://search.cpan.org/~maurice/Email-Valid-0.15/) module, which we'll look at in Chapter 14.

# Chapter 5: Advanced Forms and Perl Control Structures

In the last chapter you learned how to decode form data, and mail it to yourself. However, one problem with the guestbook program is that it didn't do any error-checking or specialized processing. You might not want to get blank forms, or you may want to require certain fields to be filled out. You might also want to write a quiz or questionnaire, and have your program take different actions depending on the answers. All of these things require some more advanced processing of the form data, and that will usually involve using control structures in your Perl code.

Control structures include conditional statements, such as if/elsif/else blocks, as well as loops like foreach, for and while.

## If Conditions

You've already seen if/elsif in action. The structure is always started by the word if, followed by a condition to be evaluated, then a pair of braces indicating the beginning and end of the code to be executed if the condition is true. The condition is enclosed in parentheses:

if (condition) {

code to be executed

}

The condition statement can be anything that evaluates to true or false. In Perl, any string is true except the empty string and 0. Any number is true except 0. An undefined value (or undef) is false.You can also test whether a certain value equals something, or doesn't equal something, or is greater than or less than something. There are different conditional test operators, depending on whether the variable you want to test is a string or a number:

**Relational and Equality Operators**

|  |  |  |
| --- | --- | --- |
| Test | Numbers | Strings |
| $x is equal to $y | $x == $y | $x eq $y |
| $x is not equal to $y | $x != $y | $x ne $y |
| $x is greater than $y | $x > $y | $x gt $y |
| $x is greater than or equal to $y | $x >= $y | $x ge $y |
| $x is less than $y | $x < $y | $x lt $y |
| $x is less than or equal to $y | $x <= $y | $x le $y |

If it's a string test, you use the letter operators (eq, ne, lt, etc.), and if it's a numeric test, you use the symbols (==, !=, etc.). Also, if you are doing numeric tests, keep in mind that $x >= $y is not the same as $x => $y. Be sure to use the correct operator!

Here is an example of a numeric test. If $varname is greater than 23, the code inside the curly braces is executed:

if ($varname > 23) {

# do stuff here if the condition is true

}

If you need to have more than one condition, you can add elsif and else blocks:

if ($varname eq "somestring") {

# do stuff here if the condition is true

}

elsif ($varname eq "someotherstring") {

# do other stuff

}

else {

# do this if none of the other conditions are met

}

The line breaks are not required; this example is just as valid:

if ($varname > 23) {

print "$varname is greater than 23";

} elsif ($varname == 23) {

print "$varname is 23";

} else { print "$varname is less than 23"; }

You can join conditions together by using logical operators:

**Logical Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Example | Explanation |
| && | condition1 && condition2 | True if condition1 and condition2 are both true |
| || | condition1 || condition2 | True if either condition1 or condition2 is true |
| and | condition1 and condition2 | Same as && but lower precedence |
| or | condition1 or condition2 | Same as || but lower precedence |

Logical operators are evaluated from left to right. *Precedence* indicates which operator is evaluated first, in the event that more than one operator appears on one line. In a case like this:

condition1 || condition2 && condition3

condition2 && condition3 is evaluated first, then the result of that evaluation is used in the || evaluation.

and and or work the same way as && and ||, although they have lower precedence than their symbolic counterparts.

## Unless

unless is similar to if. Let's say you wanted to execute code only if a certain condition were false. You could do something like this:

if ($varname != 23) {

# code to execute if $varname is not 23

}

The same test can be done using unless:

unless ($varname == 23) {

# code to execute if $varname is not 23

}

There is no "elseunless", but you can use an else clause:

unless ($varname == 23) {

# code to execute if $varname is not 23

} else {

# code to execute if $varname IS 23

}

## Validating Form Data

You should always *validate* data submitted on a form; that is, check to see that the form fields aren't blank, and that the data submitted is in the format you expected. This is typically done with if/elsif blocks.

Here are some examples. This condition checks to see if the "name" field isn't blank:

if (param('name') eq "") {

&dienice("Please fill out the field for your name.");

}

You can also test multiple fields at the same time:

if (param('name') eq "" or param('email') eq "") {

&dienice("Please fill out the fields for your name

and email address.");

}

The above code will return an error if either the name or email fields are left blank.

param('fieldname') always returns one of the following:

|  |  |
| --- | --- |
| undef — or undefined | fieldname is not defined in the form itself, or it's a checkbox/radio button field that wasn't checked. |
| the empty string | fieldname exists in the form but the user didn't type anything into that field (for text fields) |
| one or more values | whatever the user typed into the field(s) |

If your form has more than one field containing the same fieldname, then the values are stored sequentially in an array, accessed by param('fieldname').

You should always validate all form data — even fields that are submitted as hidden fields in your form. Don't assume that your form is always the one calling your program. Any external site can send data to your CGI. Never trust form input data.

## Looping

Loops allow you to repeat code for as long as a condition is met. Perl has several loop control structures: foreach, for, while and until.

### Foreach Loops

foreach iterates through a list of values:

foreach my $i (@arrayname) {

# code here

}

This loops through each element of @arrayname, setting $i to the current array element for each pass through the loop. You may omit the loop variable $i:

foreach (@arrayname) {

# $\_ is the current array element

}

This sets the special Perl variable $\_ to each array element. $\_ does not need to be declared (it's part of the Perl language) and its scope localized to the loop itself.

### For Loops

Perl also supports C-style for loops:

for ($i = 1; $i < 23; $i++) {

# code here

}

The for statement uses a 3-part conditional: the loop initializer; the loop condition (how long to run the loop); and the loop re-initializer (what to do at the end of each iteration of the loop). In the above example, the loop initializes with $i being set to 1. The loop will run for as long as $i is less than 23, and at the end of each iteration $i is incremented by 1 using the auto-increment operator (++).

The conditional expressions are optional. You can do infinite loops by omitting all three conditions:

for (;;) {

# code here

}

You can also write infinite loops with while.

### While Loops

A while loop executes as long as particular condition is true:

while (condition) {

# code to run as long as condition is true

}

### Until Loops

until is the reverse of while. It executes as long as a particular condition is NOT true:

until (condition) {

# code to run as long as condition is not true

}

### Infinite Loops

An infinite loop is usually written like so:

while (1) {

# code here

}

Obviously unless you want your program to run forever, you'll need some way to break out of these infinite loops. We'll look at breaking next.

## Breaking from Loops

There are several ways to break from a loop. To stop the current loop iteration (and move on to the next one), use the next command:

foreach my $i (1..20) {

if ($i == 13) {

next;

}

print "$i\n";

}

This example prints the numbers from 1 to 20, except for the number 13. When it reaches 13, it skips to the next iteration of the loop.

To break out of a loop entirely, use the last command:

foreach my $i (1..20) {

if ($i == 13) {

last;

}

print "$i\n";

}

This example prints the numbers from 1 to 12, then terminates the loop when it reaches 13.

next and last only effect the innermost loop structure, so if you have something like this:

foreach my $i (@list1) {

foreach my $j (@list2) {

if ($i == 5 && $j == 23) {

last;

}

}

# this is where that last sends you

}

The last command only terminates the innermost loop. If you want to break out of the outer loop, you need to use loop labels:

OUTER: foreach my $i (@list1) {

INNER: foreach my $j (@list2) {

if ($i == 5 && $j == 23) {

last OUTER;

}

}

}

# this is where that last sends you

The loop label is a string that appears before the loop command (foreach, for, or while). In this example we used OUTER as the label for the outer foreach loop and INNER for the inner loop label.

Now that you've seen the various types of Perl control structures, let's look at how to apply them to handling advanced form data.

## Handling Checkboxes

Checkboxes allow the viewer to select one or more options on a form. If you assign each checkbox field a different name, you can print them the same way you'd print any form field using param('fieldname').

Here is the HTML code for a set of checkboxes:

<b>Pick a Color:</b><br>

<form action="colors.cgi" method="POST">

<input type="checkbox" name="red" value=1> Red<br>

<input type="checkbox" name="green" value=1> Green<br>

<input type="checkbox" name="blue" value=1> Blue<br>

<input type="checkbox" name="gold" value=1> Gold<br>

<input type="submit">

</form>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch5/colors.html>

This example lets the visitor pick as many options as they want — or none, if they prefer. Since this example uses a different field name for each checkbox, you can test it using param:

my @colors = ("red","green","blue","gold");

foreach my $color (@colors) {

if (param($color)) {

print "You picked $color.\n";

}

}

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch5/colors-cgi.html>

Since we set the value of each checkbox to 1 (a true value), we didn't need to actually see if param($color) was equal to anything — if the box is checked, its true. If it's not checked, then param($color) is undefined and therefore not true.

The other way you could code this form is to set each checkbox name to the same name, and use a different value for each checkbox:

<b>Pick a Color:</b><br>

<form action="colors.cgi" method="POST">

<input type="checkbox" name="color" value="red"> Red<br>

<input type="checkbox" name="color" value="green"> Green<br>

<input type="checkbox" name="color" value="blue"> Blue<br>

<input type="checkbox" name="color" value="gold"> Gold<br>

<input type="submit">

</form>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch5/colors2.html>

param('color') returns a list of the selected checkboxes, which you can then store in an array. Here is how you'd use it in your CGI program:

my @colors = param('color');

foreach my $color (@colors) {

print "You picked $color.<br>\n";

}

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch5/colors2-cgi.html>

## Handling Radio Buttons

Radio buttons are similar to checkboxes in that you can have several buttons, but the difference is that the viewer can only pick one choice. As with our last checkbox example, the group of related radio buttons must all have the same name, and different values:

<b>Pick a Color:</b><br>

<form action="colors.cgi" method="POST">

<input type="radio" name="color" value="red"> Red<br>

<input type="radio" name="color" value="green"> Green<br>

<input type="radio" name="color" value="blue"> Blue<br>

<input type="radio" name="color" value="gold"> Gold<br>

<input type="submit">

</form>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch5/colors3.html>

Since the viewer can only choose one item from a set of radio buttons, param('color') will be the color that was picked:

my $color = param('color');

print "You picked $color.<br>\n";

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch5/colors3-cgi.html>

It's usually best to set the values of radio buttons to something meaningful; this allows you to print out the button name and its value, without having to store another list inside your CGI program. But if your buttons have lengthy values, or values unsuitable for storing in the value field, you can set each value to an abbreviation, then define a hash in your CGI program where the hash keys correspond to the abbreviations. The hash values can then contain longer data.

Let's try it. Create a new HTML form called colors4.html:

**Program 5-1: colors4.html - Favorite Colors HTML Form**

<html><head><title>Pick a Color</title></head>

<body>

<b>Pick a Color:</b><br>

<form action="colors4.cgi" method="POST">

<input type="radio" name="color" value="red"> Red<br>

<input type="radio" name="color" value="green"> Green<br>

<input type="radio" name="color" value="blue"> Blue<br>

<input type="radio" name="color" value="gold"> Gold<br>

<input type="submit">

</form>

</body></html>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch5/colors4.html>

Next create colors4.cgi. This example not only prints out the color you picked, but also sets the page background to that color. The %colors hash stores the various RGB hex values for each color. The hex value for the selected color is then passed to CGI.pm's start\_html function as the bgcolor (background color) parameter.

**Program 5-2: colors4.cgi - Favorite Colors Program**

#!/usr/bin/perl -wT

use strict;

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

my %colors = ( red => "#ff0000",

green => "#00ff00",

blue => "#0000ff",

gold => "#cccc00");

print header;

my $color = param('color');

# do some validation - be sure they picked a valid color

if (exists $colors{$color}) {

print start\_html(-title=>"Results", -bgcolor=>$color);

print "You picked $color.<br>\n";

} else {

print start\_html(-title=>"Results");

print "You didn't pick a color! (You picked '$color')";

}

print end\_html;

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch5/colors4-cgi.html>

## Handling SELECT Fields

SELECT fields are handled almost the same way as radio buttons. A SELECT field is a pull-down menu with one or more choices. Unless you specify a multiple select (see below), the viewer can only choose one option. Here is the HTML for creating a SELECT field:

<select name="color">

<option value="red"> Red

<option value="green"> Green

<option value="blue"> Blue

<option value="gold"> Gold

</select>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch5/colors5.html>

As with radio buttons, you access the selection in your CGI program using param('color'):

my $color = param('color');

print "You picked $color.<br>\n";

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch5/colors5-cgi.html>

## Multiple-choice SELECTs

Multiple SELECTs allow the viewer to choose more than one option from the list, usually by option-clicking or control-clicking on the options they want. Here is the HTML for a multiple SELECT:

<select name="color" multiple size=3>

<option value="red"> Red

<option value="green"> Green

<option value="blue"> Blue

<option value="gold"> Gold

</select>

In your CGI program, param('color') returns a list of the selected values, just as it did when we had multiple checkboxes of the same name:

my @colors = param('color');

foreach my $color (@colors) {

print "You picked $color.<br>\n";

}

So now you've seen every type of form element (except for file-uploads, which we'll look at in Chapter 14), and in every case you've seen that CGI.pm's param function returns the value (or values) from each form field. The value returned by param is always a list, but for text, textarea, password, radio, and single select fields you can use it in a scalar context. For checkboxes and multiple select fields, you use it in an array context.

In the next chapter we'll learn how to read and write data files, so you'll be able to save and analyze the data collected by your forms.

# Chapter 6: Reading and Writing Data Files

As you start to program more advanced CGI applications, you'll want to store data so you can use it later. Maybe you have a guestbook program and want to keep a log of the names and email addresses of visitors, or a page counter that must update a counter file, or a program that scans a flat-file database and draws info from it to generate a page. You can do this by reading and writing data files (often called file I/O).

## File Permissions

Most web servers run with very limited permissions; this protects the server (and the system it's running on) from malicious attacks by users or web visitors. On Unix systems, the web process runs under its own userid, typically the "web" or "nobody" user. Unfortunately this means the server doesn't have permission to create files in your directory. In order to write to a data file, you must usually make the file (or the directory where the file will be created) world-writable — or at least writable by the web process userid. In Unix a file can be made world-writable using the **chmod** command:

**chmod 666 myfile.dat**

To set a directory world-writable, you'd do:

**chmod 777 directoryname**

See Appendix A for a chart of the various chmod permissions.

Unfortunately, if the file is world-writable, it can be written to (or even deleted) by other users on the system. You should be very cautious about creating world-writable files in your web space, and you should *never* create a world-writable directory there. (An attacker could use this to install their own CGI programs there.) If you must have a world-writable directory, either use /tmp (on Unix), or a directory outside of your web space. For example if your web pages are in /home/you/public\_html, set up your writable files and directories in /home/you.

A much better solution is to configure the server to run your programs with your userid. Some examples of this are CGIwrap (platform independent) and suEXEC (for Apache/Unix). Both of these force CGI programs on the web server to run under the program owner's userid and permissions. Obviously if your CGI program is running with your userid, it will be able to create, read and write files in your directory without needing the files to be world-writable.

The Apache web server also allows the webmaster to define what user and group the server runs under. If you have your own domain, ask your webmaster to set up your domain to run under your own userid and group permissions.

Permissions are less of a problem if you only want to read a file. If you set the file permissions so that it is group- and world-readable, your CGI programs can then safely read from that file. Use caution, though; if your program can read the file, so can the webserver, and if the file is in your webspace, someone can type the direct URL and view the contents of the file. Be sure not to put sensitive data in a publicly readable file.

## Opening Files

Reading and writing files is done by opening a file and associating it with a filehandle. This is done with the statement:

open(filehandle,filename);

The filename may be prefixed with a >, which means to overwrite anything that's in the file now, or with a >>, which means to append to the bottom of the existing file. If both > and >> are omitted, the file is opened for reading only. Here are some examples:

open(INF,"out.txt"); # opens mydata.txt for reading

open(OUTF,">out.txt"); # opens out.txt for overwriting

open(OUTF,">>out.txt"); # opens out.txt for appending

open(FH, "+<out.txt"); # opens existing file out.txt for reading AND writing

The filehandles in these cases are INF, OUTF and FH. You can use just about any name for the filehandle.

Also, a warning: your web server might do strange things with the path your programs run under, so it's possible you'll have to use the full path to the file (such as /home/you/public\_html/somedata.txt), rather than just the filename. This is generally not the case with the Apache web server, but some other servers behave differently. Try opening files with just the filename first (provided the file is in the same directory as your CGI program), and if it doesn't work, then use the full path.

One problem with the above code is that it doesn't check the return value of open to ensure the file was really opened. open returns nonzero upon success, or undef (which is a false value) otherwise. The safe way to open a file is as follows:

open(OUTF,">outdata.txt") or &dienice("Can't open outdata.txt for writing: $!");

This uses the "dienice" subroutine we wrote in Chapter 4 to display an error message and exit if the file can't be opened. You should do this for all file opens, because if you don't, your CGI program will continue running even if the file isn't open, and you could end up losing data. It can be quite frustrating to realize you've had a survey running for several weeks while no data was being saved to the output file.

The $! in the above example is a special Perl variable that stores the error code returned by the failed open statement. Printing it may help you figure out why the open failed.

## Guestbook Form with File Write

Let's try this by modifying the guestbook program you wrote in Chapter 4. The program already sends you e-mail with the information; we're going to have it write its data to a file as well.

First you'll need to create the output file and make it writable, because your CGI program probably can't create new files in your directory. If you're using Unix, log into the Unix shell, **cd** to the directory where your guestbook program is located, and type the following:

**touch guestbook.txt  
chmod 622 guestbook.txt**

The Unix **touch** command, in this case, creates a new, empty file called "guestbook.txt". (If the file already exists, touch simply updates the last-modified timestamp of the file.) The chmod 622 command makes the file read/write for you (the owner), and write-only for everyone else.

If you don't have Unix shell access (or you aren't using a Unix system), you should create or upload an empty file called guestbook.txt in the directory where your guestbook.cgi program is located, then adjust the file permissions on it using your FTP program.

Now you'll need to modify guestbook.cgi to write to the file:

**Program 6-1: guestbook.cgi - Guestbook Program With File Write**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;

print start\_html("Results");

# first print the mail message...

$ENV{PATH} = "/usr/sbin";

open (MAIL, "|/usr/sbin/sendmail -oi -t -odq") or

&dienice("Can't fork for sendmail: $!\n");

print MAIL "To: recipient\@cgi101.com\n";

print MAIL "From: nobody\@cgi101.com\n";

print MAIL "Subject: Form Data\n\n";

foreach my $p (param()) {

print MAIL "$p = ", param($p), "\n";

}

close(MAIL);

# now write (append) to the file

open(OUT, ">>guestbook.txt") or &dienice("Couldn't open output file: $!");

foreach my $p (param()) {

print OUT param($p), "|";

}

print OUT "\n";

close(OUT);

print <<EndHTML;

<h2>Thank You</h2>

<p>Thank you for writing!</p>

<p>Return to our <a href="index.html">home page</a>.</p>

EndHTML

print end\_html;

sub dienice {

my($errmsg) = @\_;

print "<h2>Error</h2>\n";

print "<p>$errmsg</p>\n";

print end\_html;

exit;

}

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch6/guestbook-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch6/guestbook.html>

Now go back to your browser and fill out the guestbook form again. If your CGI program runs without any errors, you should see data added to the guestbook.txt file. The resulting file will show the submitted form data in pipe-separated form:

Someone|someone@wherever.com|comments here

Ideally you'll have one line of data (or record) for each form that is filled out. This is what's called a flat-file database.

Unfortunately if the visitor enters multiple lines in the comments field, you'll end up with multiple lines in the data file. To remove the newlines, you should substitute newline characters (\n) as well as hard returns (\r). Perl has powerful pattern matching and replacement capabilities; it can match the most complex patterns in a string using regular expressions (see Chapter 13). The basic syntax for substitution is:

$mystring =~ s/pattern/replacement/;

This command substitutes "pattern" for "replacement" in the scalar variable $mystring. Notice the operator is a =~ (an equals sign followed by a tilde); this is Perl's binding operator and indicates a regular expression pattern match/substitution/replacement is about to follow.

Here is how to replace the end-of-line characters in your guestbook program:

foreach my $p (param()) {

my $value = param($p);

$value =~ s/\n/ /g; # replace newlines with spaces

$value =~ s/\r//g; # remove hard returns

print OUT "$p = $value,";

}

Go ahead and change your program, then test it again in your browser. View the guestbook.txt file in your browser or in a text editor and observe the results.

## File Locking

CGI processes on a Unix web server can run simultaneously, and if two programs try to open and write the same file at the same time, the file may be erased, and you'll lose all of your data. To prevent this, you need to lock the files you are writing to. There are two types of file locks:

* A shared lock allows more than one program (or other process) to access the file at the same time. A program should use a shared lock when reading from a file.
* An exclusive lock allows only one program or process to access the file while the lock is held. A program should use an exclusive lock when writing to a file.

File locking is accomplished in Perl using the Fcntl module (which is part of the standard library), and the flock function. The use statement is like CGI.pm's:

use Fcntl qw(:flock);

The Fcntl module provides symbolic values (like abbreviations) representing the correct lock numbers for the flock function, but you must specify :flock in the use statement in order for Fctnl to export those values. The values are as follows:

|  |  |
| --- | --- |
| LOCK\_SH | shared lock |
| LOCK\_EX | exclusive lock |
| LOCK\_NB | non-blocking lock |
| LOCK\_UN | unlock |

These abbreviations can then be passed to flock. The flock function takes two arguments: the filehandle and the lock type, which is typically a number. The number may vary depending on what operating system you are using, so it's best to use the symbolic values provided by Fcntl. A file is locked after you open it (because the filehandle doesn't exist before you open the file):

open(FH, "filename") or &dienice("Can"t open file: $!");

flock(FH, LOCK\_SH);

The lock will be released automatically when you close the file or when the program finishes.

Keep in mind that file locking is only effective if all of the programs that read and write to that file also use flock. Programs that don't will ignore the locks held by other processes.

Since flock may force your CGI program to wait for another process to finish writing to a file, you should also reset the file pointer, using the seek function:

seek(filehandle, offset, whence);

*offset* is the number of bytes to move the pointer, relative to *whence*, which is one of the following:

|  |  |
| --- | --- |
| 0 | beginning of file |
| 1 | current file position |
| 2 | end of file |

So seek(OUTF,0,2) repositions the pointer to the end of the file. If you were reading the file instead of writing to it, you'd want to do seek(OUTF,0,0) to reset the pointer to the beginning of the file.

The Fcntl module also provides symbolic values for the seek pointers:

|  |  |
| --- | --- |
| SEEK\_SET | beginning of file |
| SEEK\_CUR | current file position |
| SEEK\_END | end of file |

To use these, add :seek to the use Fcntl statement:

use Fcntl qw(:flock :seek);

Now you can use seek(OUTF,0,SEEK\_END) to reset the file pointer to the end of the file, or seek(OUTF,0,SEEK\_SET) to reset it to the beginning of the file.

## Closing Files

When you're finished writing to a file, it's best to close the file, like so:

close(filehandle);

Files are automatically closed when your program ends. File locks are released when the file is closed, so it is not necessary to actually unlock the file before closing it. (In fact, releasing the lock before the file is closed can be dangerous and cause you to lose data.)

## Reading Files

There are two ways you can handle reading data from a file: you can either read one line at a time, or read the entire file into an array. Here's an example:

open(FH,"guestbook.txt") or &dienice("Can't open guestbook.txt: $!");

my $a = <FH>; # reads one line from the file into

# the scalar $a

my @b = <FH>; # reads the ENTIRE FILE into array @b

close(FH); # closes the file

If you were to use this code in your program, you'd end up with the first line of guestbook.txt being stored in $a, and the remainder of the file in array @b (with each element of @b containing one line of data from the file). The actual read occurs with <filehandle>; the amount of data read depends on the type of variable you save it into.

The following section of code shows how to read the entire file into an array, then loop through each element of the array to print out each line:

open(FH,"guestbook.txt") or &dienice("Can"t open guestbook.txt: $!");

my @ary = <FH>;

close(FH);

foreach my $line (@ary) {

print $line;

}

This code minimizes the amount of time the file is actually open. The drawback is it causes your CGI program to consume as much memory as the size of the file. Obviously for very large files that's not a good idea; if your program consumes more memory than the machine has available, it could crash the whole machine (or at the very least make things extremely slow). To process data from a very large file, it's better to use a while loop to read one line at a time:

open(FH,"guestbook.txt") or &dienice("Can"t open guestbook.txt: $!");

while (my $line = <FH>) {

print $line;

}

close(FH);

## Poll Program

Let's try another example: a web poll. You've probably seen them on various news sites. A basic poll consists of one question and several potential answers (as radio buttons); you pick one of the answers, vote, then see the poll results on the next page.

Start by creating the poll HTML form. Use whatever question and answer set you wish.

**Program 6-2: poll.html - Poll HTML Form**

<form action="poll.cgi" method="POST">

Which was your favorite <i>Lord of the Rings</i> film?<br>

<input type="radio" name="pick" value="fotr">The Fellowship of the Ring<br>

<input type="radio" name="pick" value="ttt">The Two Towers<br>

<input type="radio" name="pick" value="rotk">Return of the King<br>

<input type="radio" name="pick" value="none">I didn't watch them<br>

<input type="submit" value="Vote">

</form>

<a href="results.cgi">View Results</a><br>

http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch6/poll.html>

In this example we're using abbreviations for the radio button values. Our CGI program will translate the abbreviations appropriately.

Now the voting CGI program will write the result to a file. Rather than having this program analyze the results, we'll simply use a redirect to bounce the viewer to a third program (results.cgi). That way you won't need to write the results code twice.

Here is how the voting program (poll.cgi) should look:

**Program 6-3: poll.cgi - Poll Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

use Fcntl qw(:flock :seek);

my $outfile = "poll.out";

# only record the vote if they actually picked something

if (param('pick')) {

open(OUT, ">>$outfile") or &dienice("Couldn't open $outfile: $!");

flock(OUT, LOCK\_EX); # set an exclusive lock

seek(OUT, 0, SEEK\_END); # then seek the end of file

print OUT param('pick'),"\n";

close(OUT);

} else {

# this is optional, but if they didn't vote, you might

# want to tell them about it...

&dienice("You didn't pick anything!");

}

# redirect to the results.cgi.

# (Change to your own URL...)

print redirect("http://cgi101.com/book/ch6/results.cgi");

sub dienice {

my($msg) = @\_;

print header;

print start\_html("Error");

print h2("Error");

print $msg;

print end\_html;

exit;

}

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch6/poll-cgi.html>

Finally results.cgi reads the file where the votes are stored, totals the overall votes as well as the votes for each choice, and displays them in table format.

**Program 6-4: results.cgi - Poll Results Program**

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

use Fcntl qw(:flock :seek);

my $outfile = "poll.out";

print header;

print start\_html("Results");

# open the file for reading

open(IN, "$outfile") or &dienice("Couldn't open $outfile: $!");

# set a shared lock

flock(IN, LOCK\_SH);

# then seek the beginning of the file

seek(IN, 0, SEEK\_SET);

# declare the totals variables

my($total\_votes, %results);

# initialize all of the counts to zero:

foreach my $i ("fotr", "ttt", "rotk", "none") {

$results{$i} = 0;

}

# now read the file one line at a time:

while (my $rec = <IN>) {

chomp($rec);

$total\_votes = $total\_votes + 1;

$results{$rec} = $results{$rec} + 1;

}

close(IN);

# now display a summary:

print <<End;

<b>Which was your favorite <i>Lord of the Rings</i> film?

</b><br>

<table border=0 width=50%>

<tr>

<td>The Fellowship of the Ring</td>

<td>$results{fotr} votes</td>

</tr>

<tr>

<td>The Two Towers</td>

<td>$results{ttt} votes</td>

</tr>

<tr>

<td>Return of the King</td>

<td>$results{rotk} votes</td>

</tr>

<tr>

<td>didn't watch them</td>

<td>$results{none} votes</td>

</tr>

</table>

<p>

$total\_votes votes total

</p>

End

print end\_html;

sub dienice {

my($msg) = @\_;

print h2("Error");

print $msg;

print end\_html;

exit;

}

http://www.cgi101.com/book/img/doc.gifSource code: <http://www.cgi101.com/book/ch6/results-cgi.html>  
http://www.cgi101.com/book/img/arrow.gifWorking example: <http://www.cgi101.com/book/ch6/results.cgi>

The results program only shows the total number of votes. You may also want to calculate the percentages and display a bar-graph for each vote relative to the overall total. We'll look at how to calculate percentages in the next chapter.

# Windows DNA - Relevance

## Where would Windows DNA be relevant to us?

Customers want these services:

* *Web enabling applications:*

This is a very real possibility. We can market services for web enabling applications using the Microsoft Windows DNA framework.

* *Development of Large applications:*

When customers ask for a complete end-to-end solution that involves a database and users in the internet/intranet - we come down to Windows DNA. We use the DNA framework to develop applications.

* *Component development as a part of large applications:*

Customers want us to develop components for their Windows DNA applications: This might be a very typical development task a few years hence if Microsoft has their way with COM and Windows DNA.

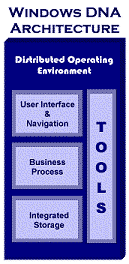
Read on to find out what DNA is all about.

[[Click here to go to Top](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)

## What is it and what is it not?

Windows DNA is not a solution: it is a methodology propounded by Microsoft for building web enabled applications.

Overall, the DNA methodology is an umbrella term that covers many existing technologies to help us design and implement robust, distributed applications. It visualizes this whole application as a series of tiers, with the client at the top and the data store at the bottom. The core of DNA is the use of business objects in a middle tier of the application, and this is supported by two solutions: Microsoft Transaction Server (MTS), which is the component manager offering full transaction support; and Microsoft Message Queue Server (MSMQ), which provides the fault-tolerance required in a distributed application.



Windows DNA is like a map to help developers, technology planners, and IS managers avoid getting lost in the details of Microsoft's many operating systems, tools, technologies, and applications.

Windows DNA, at its core, is about integrating Web and client-server models of computing.

Explore this a little and get a glimpse of what the model comes down to.

* Want to incorporate area-wise tax figures into the application? Buy a custom tax component.
* Want to read data into a custom view? Buy an ADO component.
* Want to send custom email messages to customers after a transaction? Plug in an email component.

[[Click here to go to Top](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)

## Windows DNA building blocks

Another way of phrasing the same question is "What are the technologies that make up the building blocks for Windows DNA?"

Here are the basic technologies:

* COM and DCOM: Well, we cannot ever over-emphasize this. COM, COM and COM again.
* ADO, OLE-DB
* Microsoft Transaction server.

Here is approximately what it takes to round off the basic technology skills:

* Win 32
* Microsoft Message Queue server.
* HTML, DHTML
* Site Server
* Java
* Security (Kerberos)

We in Calsoft Labs have considerable expertise in most of the supporting technologies.

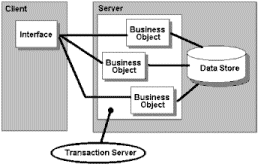
[[Click here to go to Top](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)

## Basic Facts on Windows DNA

* Built around COM: Business objects are COM objects
* Windows DNA applications can be built using existing tools such as VB, VC++, VJ++, Delphi, etc.
* Massively scalable. Windows DNA architecture can support thousands of simultaneous users and can do so with the same application logic, tool set, and engineering effort as required for an application that supports a single user.
* Applications built for Windows DNA will support all platforms that support HTML

In DNA, business objects are implemented as software components. These components can be accessed by the client interface application or by another component, and can themselves call on other components, data stores, etc. Componentization of business rules brings many benefits, such as easier maintenance, encapsulation of the rules, protection of intellectual copyright, etc.

DNA is an approach to design that can speed up overall development time, while creating more reliable and fault tolerant applications that are easily distributable over a whole variety of networks. In this book, we're concentrating on the Internet, using TCP/IP as our protocol. However, the components can just as easily be used with compiled applications specially written to follow the DNA principles.



[[Click here to go to Top](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)](http://www.calsoftlabs.com/whitepapers/windows-dna.html#top)

## More Details on DNA: Windows DNA services

DNA centers around a set of services that are provided by Windows NT Server. These include both application and infrastructure services. The infrastructure services are provided by Windows NT Server itself, and consist of things like the Network service, the NT5 Active Directory service, the Remote File and Print service, the Security service and various other Component services that don't easily fall into one of these categories

What DNA means is that each of these services provides a common and easily accessible interface through which other components and scripts can access them. There is no 'natural' language, and developers use whichever language best suits their particular requirements. And, because the interfaces are published and open, independent (non-Microsoft) suppliers can create components and services of their own which can 'plug into' NT Server.

Application Services

Example

Database

Microsoft SQL Server

Java virtual machine

Microsoft JVM

Mail and collaboration service

Microsoft Exchange Server

Message queuing service

Microsoft Message Queue Server (MSMQ)

Scripting

Active Server Pages, Dynamic HTML, JScript, VBScript

Transaction service

Microsoft Transaction Server (MTS)

Universal data access

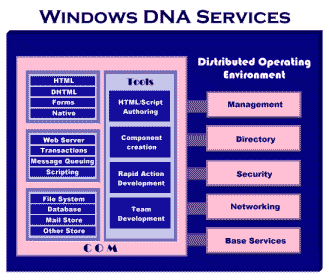
ADO, OLE-DB, ODBC, etc.

Web browser

Web browser Netscape Communicator, Internet Explorer .

Web server

Microsoft Internet Information Server (IIS)



#### An important benefit: Scalability

The most important feature of DNA is probably scalability - which has been one of the major factors blocking adoption of Windows NT in corporate environments.

Scalability is particularly important in a Web environment, where popular sites can attract thousands of visitors per day. It's a lot harder to plan capacity in this environment than on a local area network with a fixed number of users and a stable traffic pattern. To get round this, one of the new services supplied with Windows NT Server is the Microsoft Transaction Server (MTS). As we'll see in later chapters, this can bring benefits to DNA-compliant applications in a number of different, and not always obvious, ways.

Microsoft promises that, using Windows DNA, the very same design and architecture needs to be employed regardless of whether the application will have 10 or 10000 concurrent users.

#### Trends that suggest we look seriously at Windows DNA

Here are some things one can speculate on:

* Component development is increasing in importance. Technological uncertainties are being plugged(to a large extent, in any case) by Microsoft solutions. All these technologies/solutions fit into one big map called Windows DNA. Developers will just have to write business logic code for business objects - which will be COM components.
* Windows NT server is increasing in importance and performance. This might imply that large corporations would move their applications from other platforms to NT server. And application development on NT server would involve using many of the technologies used in Windows DNA.
* Once the Y2k rush has spent itself - Web technologies and Microsoft technologies will be a prime focus for spending by large corporations. And Windows DNA and COM will be at the very heart of all that.

One can even proceed a step further and suggest that Windows DNA might be the map to application developers (for Microsoft technologies, at the least) that will finally move software development from the coding-intensive framework to a paradigm oriented towards design and component-reuse.