Beginning Web Development with Perl

From Novice to Professional

Steve Suehring

Apress®

Beginning Web Development with Perl: From Novice to Professional

ISBN (pbk): 1-59059-531-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: James Lee
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Managers: Laura Cheu, Richard Dal Porto
Copy Editors: Marilyn Smith, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: Lori Bring
Indexer: Rebecca Plunkett
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

Contents at a Glance

PART 5 Creating Web Templates

Contents

About the Author

STEVE SUEHRING is a technology architect with a diverse set of skills. Steve works with a wide array of technologies, from mainframe OS/390 to Microsoft Windows to several distributions of Linux. Steve has written a book on MySQL, a book on Linux firewalls, and numerous magazine articles. During his tenure as an editor for *LinuxWorld Magazine*, Steve focused on advocacy of Linux and open-source software, as well as computer security. Among the articles Steve wrote for *LinuxWorld Magazine* is a cover story featuring the WilliamsF1 team's use of Linux to design their Formula 1 car. Steve is also a Cisco Certified Network Professional (CCNP).

About the Technical Reviewer

JAMES LEE is a hacker and open-source advocate based in Illinois. He has a master's degree from Northwestern University, where he can often be seen rooting for the Wildcats during football season. The founder of Onsight (`http://www.onsight.com`), he has worked as a programmer, trainer, manager, writer, and open-source advocate. He is the coauthor of the recently released *Hacking Linux Exposed, Second Edition* (Osbourne/McGraw-Hill, 2002). He has also written a number of articles on Perl for *Linux Journal*. Lee enjoys hacking Perl, developing software for the Web, reading, traveling, and most of all playing with his kids, who are too young to know why Dad's favorite animals are penguins and camels.

Acknowledgments

Thanks to James Lee for the excellent technical review assistance, and thanks to the entire Apress staff. Thanks to Laura Lewin and Studio B. Many thanks to everyone who helped with this or other projects that I've worked on, not only in 2005 but throughout the years. This list is always difficult but should include Jim, John, Jer, Chris, Ron, Andy, Jay, Brian, Michael, Dan, Justin, Sarah, AJ, Denise, Rob, Tim, Greg, Jeff, Aaron, Scott, Karla, Chad, Jess, Roman, Caele, Makenna, Tony, Joe, Rebecca, Tom, Kevin, Keith, Brooke, Nick, Colin, Bob, Mary, Sue, Sandi, Amy, Chris, Emily, Mike, Sandy, Ernie, Vera, Matthew, Kyle, Eric, Erich, Erin, Heather, Nate, Brandon, Maya, Deb, Steve, Jaci, JJ, Dave, Sam, Ed, Mike, Al, Neil, Geddy, Alex, Bill, Pearl, Moff, Duff, Jason, Mark, Michelle, Tara, Al, Spencer, Pat, Nicole, Jill, and everyone else that I'm forgetting.

My hope is that by using mostly first names everyone will think that I'm thanking them personally, and I am, of course.

Introduction

This is a book about Perl, the programming language. More specifically, this is a book about some aspects of Perl interacting with the Internet. The book includes coverage of CGI through the Perl `CGI.pm` module, as well as much additional information. You'll explore such topics as how to send e-mail from a Perl program through `Net::SMTP`, how to use `Net::DNS`, and even how to use `Net::Ping`. You'll learn how to build a templated web site two different ways, one with the help of `HTML::Mason` and the other with the Template Toolkit. You'll learn how to consume SOAP web services, how to read and write RSS, and how to work with XML, all through Perl.

What This Book Covers

This book attempts to give readers who have learned some Perl a grasp of the concepts and considerations for programming for the Web or creating a Perl program that interacts with the network. This book is fairly Linux-centric. That's not to say that these programs won't run on a Windows platform—they will, with very little tweaking.

The book is divided into five parts. The first part covers what comes to mind for many people when they think of Perl and web development—namely, CGI development and database interaction. The CGI module is examined, and I present tips for programming CGIs, such as the Carp module for debugging. The second part of the book widens the focus into Internet interaction with Perl. The `Net::` family of modules is covered, along with the LWP for retrieving web pages through Perl. Coverage of XML and RSS make up the third part of the book, where you'll learn about XML parsing through Perl, consumption of SOAP web services, and RSS feeds. The fourth part of the book looks at `mod_perl` for development. The fifth and final part of the book examines templating through the Template Toolkit and `HTML::Mason`.

What You Should Know

To use this book successfully, you'll need to know about Perl basics such as scalar variables, arrays, and hashes; the use of control structures such as if/else, for, and while; and other topics such as those you'd find in a book like *Beginning Perl, Second Edition* (Apress, 2004). Essentially, you should be comfortable with Perl before reading this book. You should also know how to install Perl modules, whether through your distribution's mechanism, such as apt for Debian, or through the use of CPAN. The book's appendix is an excerpt from the aforementioned *Beginning Perl* title. If you haven't used Perl in a while, the appendix can provide a good refresher into the language, but you'll find that reading the entire *Beginning Perl* book will be helpful to pick up where the appendix leaves off.

This book will use Perl on a Linux system. The exact flavor of Linux isn't important, though I likely have some references to Debian. Perl being Perl means that the programs will run successfully with little modification on any system that can run Apache, including Microsoft Windows.

Why Apache?

Simply put, Apache works. Apache is overwhelmingly the most popular web server in use on the Internet, according to the Netcraft survey (`http://news.netcraft.com/archives/web_server_survey.html`). Apache is open source and configurable for just about any need. Apache is stable and predictable, even under heavy load. Apache is robust, serving web without using too many resources, and it's able to use resources effectively to serve heavy traffic loads.

This is not to say that Microsoft's IIS hasn't made good strides toward meeting its competition in Apache. IIS has indeed improved its stability with version 6, but with the benchmark being IIS 5 and its predecessors, it didn't take much to improve on the stability. However, IIS is still not as good as Apache at serving web and has a long way to go before getting there.

PART 1

CGI Development with Perl

CHAPTER 1

The CGI Module

The CGI module is central to web development with Perl. In fact, the CGI module is as far as some people go with Perl web development. However, there's more to Perl for the Web than just CGI. You have endless ways to interact with the Internet from a Perl program. This book will show you many of those methods. Even so, the CGI module is a good place to start.

In this chapter, you'll learn the basics of Perl-based CGI application development, including how to use both the function-oriented and object-oriented styles, some of the functions offered by the CGI module, how to use environment variables, tips for debugging and troubleshooting CGI applications, and the all-important security considerations. Let's begin with an introduction to CGI.

An Overview of CGI

CGI, or Common Gateway Interface, is a standard of communication between a web server and a client, such as a user with a web browser. Over time, CGI has come to refer to a class of programs that work with web servers. The programs themselves are considered to be external or separate from the web servers, but they provide a gateway between the client and server—thus the Gateway portion of the name Common Gateway Interface.

CGI is a standard unto itself. The current standard provides a common interface, which means that CGI programs can be written in a number of languages. Of the languages with which a CGI program can be written, Perl is arguably the most common.

Perl provides an easy-to-use syntax with a gently sloped learning curve. Yet Perl, along with its myriad modules, is incredibly powerful and robust. The Apache web server, which is usually run on Unix/Linux, continues to hold the dominant market share over all web servers on the Internet.[1] Perl is also very common on these same Unix/Linux systems. Being available and powerful leads to the ubiquity of Perl for CGI development.

Note If you're unfamiliar with Perl, please flip to this book's appendix for an overview of the language. In that appendix, you'll find excerpts from *Beginning Perl, Second Edition* by James Lee (Apress, 2004). I recommend picking up a copy of *Beginning Perl* for a full introduction to the language.

---

1. The surveys conducted by Netcraft (`http://news.netcraft.com/archives/web_server_survey.html`) confirm this fact.

For quite a long time (in Internet time), CGIs were frequently developed with help from a library called `cgi-lib.pl`. The CGI library, or `cgi-lib`, was a function-oriented way to simplify development of CGI programs in Perl. A developer could use the common functions of the library, rather than needing to rewrite those functions from scratch. The `cgi-lib` also enabled developers with less experience to write very powerful applications to interact with the Web.

As the Internet grew, so did the needs of Perl-based CGI developers. The CGI module, referred to as `CGI.pm` after the name of the module file itself, gives developers a very powerful interface to common CGI writing tasks. `CGI.pm` became available with Perl 5.

`CGI.pm` can be used both in the function-oriented approach of `cgi-lib` and also in an object-oriented fashion, allowing developers to use more than one CGI object within their program. The main difference between the two approaches is the way in which you call methods. With the function-oriented approach, you must call each method during the declaration of the CGI module. The object-oriented approach makes these methods available to you, regardless of whether you explicitly specify that they should be included in the program beforehand.

When using the CGI module in a function-oriented fashion, you can choose to import only the individual methods that you'll be using, or you can import just those functions within predefined logical groupings of methods. `CGI.pm` currently has several such groupings, as shown in Table 1-1. You'll see how to use these methods later in the chapter.

Table 1-1. *CGI.pm Method Groupings*

Group	Description
`:all`	Contains all of the methods available in the `CGI.pm` module
`:cgi`	Contains methods specific to the CGI protocol
`:form`	Contains methods used to generate forms
`:html`	A superset of the `:html2`, `:html3`, `:html4`, and `:netscape` groups
`:html2`	Contains methods that enable the developer to quickly use elements from the HTML version 2 specification
`:html3`	Like the `:html2` group, contains methods that enable the developer to quickly use elements from the HTML version 3 specification
`:html4`	Contains methods that enable the developer to quickly use elements from the HTML version 4 specification
`:netscape`	Contains the Netscape extensions for the HTML 3 specification, including a shortcut to my favorite HTML tag `<BLINK>`
`:multipart`	Contains methods used to help work with MIME types
`:standard`	Probably the most frequently used group of methods, a superset of the `:html2`, `:html3`, `:html4`, `:form`, and `:cgi` groups

What You Need for This Chapter

The examples in this chapter use version 3.04 of `CGI.pm` on a Linux-based Apache (version 1.3.N) server. Even though Apache 2 has been released, the 1.3 branch is stable and quite common as of this writing. The 1.3 branch is also easier to work with for beginners, making it a good choice

for this book. However, the routines in this chapter should work on many different versions of Apache or another CGI-compliant web server, and with different versions of the `CGI.pm` module as well. You can always obtain the latest version of `CGI.pm` from `http://search.cpan.org/dist/CGI.pm/`.

You'll also need to be able to execute CGI programs on your web server of choice. For example, on an Apache server, you would need a directive such as this:

```
Options ExecCGI
```

Any CGI testing performed must be done from a directory that has the `ExecCGI` option enabled in Apache.[2]

Although it's obvious (I hope), I'll point out that you'll need Perl on your system. You'll also need to know where Perl is located on the system. Perl is usually located in `/usr/bin`, but it's sometimes found in `/usr/local/bin` or elsewhere. When in doubt, use the `which` command to find the Perl interpreter:

```
% which perl
```

The system will respond with the first instance of Perl in its search path. In this case, my system happened to respond with this:

```
/usr/local/bin/perl
```

This was somewhat confusing to me, since I know that Perl is installed on this system in `/usr/bin/perl` as well. Looking at `/usr/local/bin/perl` revealed that it's merely a symbolic link[3] to the real Perl in `/usr/bin`:

```
% ls -la /usr/local/bin/perl
lrwxrwxrwx 1 root staff 13 Feb 24 2002 /usr/local/bin/perl -> /usr/bin/perl
```

Hello World, CGI Style

Before going too far, and to assist in testing your CGI environment, the requisite "Hello World" example is now provided. I'll show you how to create this example using both the function-oriented approach and the object-oriented approach, to highlight the basic differences between the two methods.

Function-Oriented Hello World

As I stated previously, the function-oriented method requires that the developer explicitly call or import the desired functions into their program. This is sometimes done by importing the method groups rather than the individual methods themselves. Recall the method groups listed in Table 1-1.

---

2. For more information about setting up Apache for CGI, see the Apache documentation at `http://httpd.apache.org/docs/howto/cgi.html`.
3. Now, if I could only remember why I created that symbolic link back in 2002. The joys of system administration on a long-lived system never cease to amaze.

The most common method grouping that I've seen used in practice is the `:standard` group. This method group contains the most frequently used methods, including those that make it easy to create and use of web forms, as well as the HTML that surrounds those forms. You'll use the `:standard` group in this first Hello World example.

■ **Caution** New developers may be tempted to use the `:all` group of methods in `CGI.pm`. However, doing so can create a security risk, since unnecessary methods are imported into your namespace. It's a good idea to use only those methods that are necessary for the functionality of your program. The `:standard` group provides a good compromise in the never-ending battle between security and complexity.

Creating the Script

The function-oriented method of CGI development allows you to rapidly develop small CGI scripts. Using the function-oriented method, the individual methods within the script can be called directly, rather than creating an object through which to call the methods, which can add overhead during development. This makes creation of the first script rather trivial, but it should give you a taste for a few of the functions involved when developing a CGI. The code in Listing 1-1 shows a Hello World script written using the function-oriented approach.

Listing 1-1. *Hello World in Function-Oriented Fashion*

```
#!/usr/bin/perl -T

use strict;
use CGI ':standard';

print header;
print start_html('Hello World');
print h1('Hello World');
print end_html();

exit;
```

You can place this code in your favorite text editor and save it to a location defined to run CGI scripts within your web server. For example, I saved this script as `hello.cgi` in the directory `/usr/lib/cgi-bin` on a server running Debian 3.0.

Don't forget that the CGI will need the correct permissions in order to run. This can usually be accomplished with the `chmod 755 <scriptname.cgi>` command:

```
chmod 755 /usr/lib/cgi-bin/hello.cgi
```

To view the code in a web browser, point the browser to the URL of the CGI script. For example, my script is on a server at the IP address 192.168.1.10. Combining the server address plus the aliased script location results in the following URL:

```
http://192.168.1.10/cgi-bin/hello.cgi
```

Obviously, the URL for your server will likely vary from this example. If all goes well, you should see a page similar to that in Figure 1-1. If all doesn't go well, skip ahead to the "Debugging and Troubleshooting" section of this chapter.

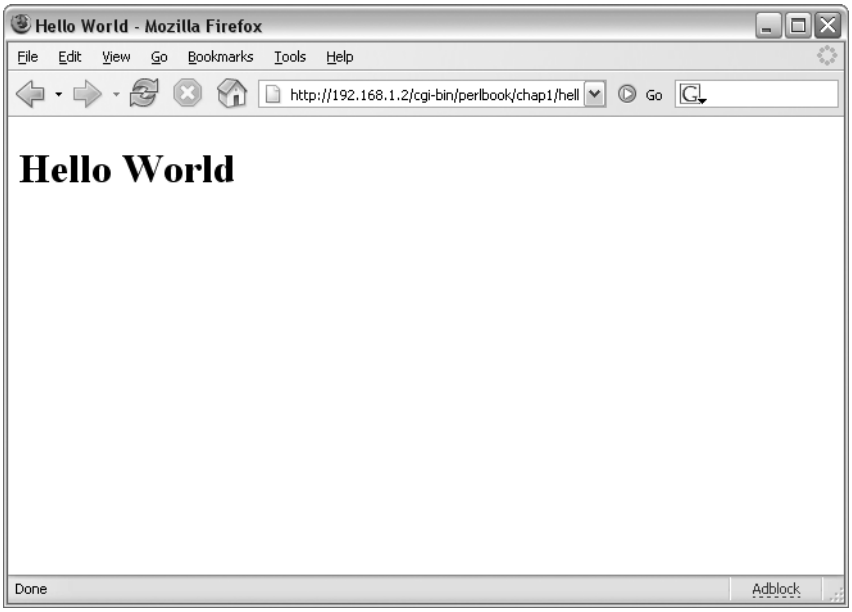

Figure 1-1. *The Hello World example viewed in a web browser*

That's all there is to programming your first CGI script. Of course, if that's all there truly were to the Perl CGI module, this book wouldn't be as thick. In the upcoming chapters, I'll expand on the CGI module to show you how to use it to interact with web forms and build applications.

Reviewing the Code

In Listing 1-1, you see the standard invocation of the `perl` interpreter `#!/usr/bin/perl`. However, there is also a `-T` option added to the invocation:

```
#!/usr/bin/perl -T
```

Refer to the "Security Considerations with CGI Programs" section later in this chapter for more information about the `-T` option. For now, just be happy that it's there.

The next line of code enables `strict` checking for the script:

```
use strict;
```

Like the `-T` option, the `strict` directive has to do with security, and it's explained in "Security Considerations with CGI Programs" section later in this chapter. This line will show up in every script in this chapter.

Following `use strict;` is the code that actually calls the `CGI.pm` module, and more specifically, calls the `:standard` method group of the `CGI.pm` module.

```
use CGI ':standard';
```

You may see this method group call made using the `qw()` operator, like this:

```
use CGI qw(:standard);
```

or

```
use CGI qw/:standard/;
```

The `qw()` operator is a *quote* operator that separates the given string into individual elements based on white space. All of these calls (and others using `qw`) are valid and essentially result in the same thing, which is the `:standard` method group being called. Perl has a number of quote and quote-like characters, aside from `qw()`. For more information about these, and all of the other Perl operators, see the `perlop` documentation (`http://perldoc.perl.org/perlop.html`).

Four functions of the CGI module are used in this script, as shown here:

```
print header;
print start_html('Hello World');
print h1('Hello World');
print end_html();
```

The first function, `header()`, sends the `Content-Type` to the browser. In this instance, the `header()` function is equivalent to using this bit of code in the script (see Chapter 5 for more information about the `Content-Type` and why the extra newline is present):

```
print "Content-Type: text/html\n\n";
```

The `header()` function can also be used for other HTTP headers, such as cookies.

The next CGI function used is `start_html()`. This function begins the HTML portion of the page with elements like `<title>`, `<html>`, `<head>`, and so on. In this instance, the script calls the `start_html()` function with a string parameter `'Hello World'`. As you saw in Figure 1-1, this parameter was placed into the title bar of the web browser.

Another CGI function called in this script is `h1()`. This function places an `<h1>` element around its parameter. In this case, the parameter passed is `'Hello World'`, and as you saw from the results in Figure 1-1, the phrase "Hello World" was indeed given `<h1>` size. Finally, the `end_html()` function is called to provide the `</body>` and `</html>` closing tags.

The code in the example uses a semicolon (`;`) to end each line and then another `print` statement to begin the next line. This was done to make the code easier to read. However, it's quite common to use a comma in place of the semicolon when programming a CGI application, so the code would look like this:

```
#!/usr/bin/perl -T

use strict;
use CGI ':standard';

print header,
   start_html('Hello World'),
```

```
   h1('Hello World'),
   end_html();

exit;
```

This code is functionally equivalent to the form in Listing 1-1, but it does save some keystrokes.

Object-Oriented Hello World

When programming CGI applications in object-oriented fashion, you must instantiate a new CGI object. This effectively means that you can use multiple CGI objects within the program.

Creating the Script

The object-oriented Hello World script, presented in Listing 1-2, uses only one CGI object. In fact, it's not all that common to use more than one CGI object within most CGI programs.

Listing 1-2. *Hello World in Object-Oriented Fashion*

```
#!/usr/bin/perl -T

use strict;
use CGI;

my $cgi = new CGI;

print $cgi->header;
print $cgi->start_html('Hello World');
print $cgi->h1('Hello World');
print $cgi->end_html();

exit;
```

Like the code in the function-oriented example, you can place Listing 1-2 in your favorite text editor and save it to a location defined to run CGI scripts within your web server. For example, I saved this script as `hello-oo.cgi` (the `oo` stands for object-oriented—how clever) into the directory `/usr/lib/cgi-bin` on a server running Debian 3.0. Again, remember that the CGI will need the correct permissions in order to run, which is usually accomplished with the `chmod 755 <scriptname.cgi>` command, as in this example:

```
chmod 755 /usr/lib/cgi-bin/hello-oo.cgi
```

To view the code in a web browser, point your browser to the URL of the CGI script, as in this example, for my server with the IP address 192.168.1.10:

```
http://192.168.1.10/cgi-bin/hello-oo.cgi
```

You should see a page similar to that in Figure 1-2. Again, see the "Debugging and Troubleshooting" section later in this chapter if you have problems with the script.

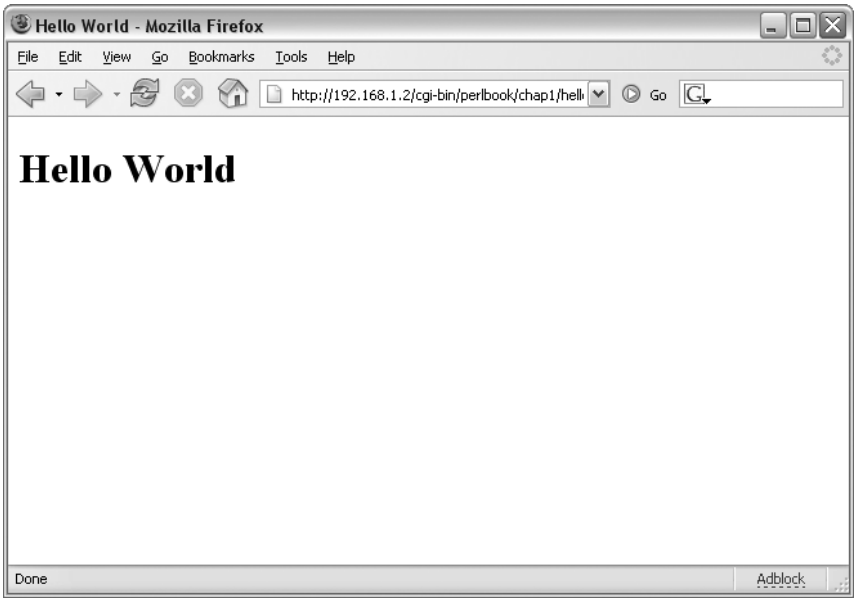

Figure 1-2. *The object-oriented Hello World example viewed in a web browser*

Reviewing the Code

If you think that the code for `hello-oo.cgi` looks eerily similar to the `hello.cgi` code, you're right. The key differences are in how the functions within the CGI module are called in this version.

Recall that in the function-oriented example, the program calls a method grouping, `:standard`, on the `use CGI` line. With the object-oriented model, that call is not necessary. Thus, the line is simply written like this:

```
use CGI;
```

The next line is new to the object-oriented approach and is responsible for instantiating the CGI object so that you can call methods within it later. This line of code creates a CGI object called `$cgi`:

```
my $cgi = new CGI;
```

Note that the script could have used any scalar variable name instead of `$cgi`. Many examples use `$q` as a variable name for this CGI object.

The four CGI functions are called next. Notice that these calls look different from the ones in the function-oriented version.

```
print $cgi->header;
print $cgi->start_html('Hello World');
print $cgi->h1('Hello World');
print $cgi->end_html();
```

Each instance of a function call must be called through the CGI object (`$cgi`) created earlier. Attempting to call one of these functions without qualifying it through the `$cgi` object will result in an Internal Server Error message.

As with the example in Listing 1-1, the code in the example uses the semicolon format, but the comma format is acceptable and common. In that format, the code looks like this:

```
#!/usr/bin/perl -T

use strict;
use CGI;

my $cgi = new CGI;

print $cgi->header,
   $cgi->start_html('Hello World'),
   $cgi->h1('Hello World'),
   $cgi->end_html();

exit;
```

A Closer Look at the CGI.pm Functions

The CGI module has a large number of functions to make the life of the programmer easier. These functions range from those that create HTML tags, HTTP headers, and cookies to those for working with web forms. This section looks at some of the most commonly used functions.

HTML Shortcuts

The HTML functions within the CGI module save time and make the code look cleaner. Without the HTML functions, a CGI program would be forced to include bits of HTML interspersed with Perl, read in external files with HTML, or (more commonly) use a combination of these techniques. This makes the code much more difficult to troubleshoot and debug.

Most of the HTML version 3 and 4 tags are available through a function in the CGI module. Simply calling the HTML function with a `print` statement will cause it to be sent to the outputted page. Recall the Hello World examples in Listings 1-1 and 1-2. These examples use the `h1()` function to create an `<h1></h1>` HTML tag around the phrase "Hello World." Viewing the source from the resulting page shows this HTML:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html
          PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
          "http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
<head><title>Hello World</title>
</head><body><h1>Hello World</h1></body></html>
```

The CGI module functions will usually create both the starting and ending HTML tags. However, should you wish to manually create these tags, you can do so by calling the function with `start_` or `end_` prepended, as in this example:

```
print start_h1;
print "This is some text";
print end_h1;
```

This code produces an opening `<h1>` tag followed by some text and then a closing `</h1>` tag. See the CGI module documentation (type `perldoc CGI` from a terminal window) for more information about HTML element generation.

You can also define your own functions for HTML tags. This allows you to define and call HTML tags that aren't already provided for within the CGI module. For example, suppose that a new HTML tag called `<snow>` is available in a certain browser, and you want to use it within your CGI program. You only need to define the function, like so:

```
use CGI qw/:standard snow/
```

Then you can call it in the same way that you would call any other HTML function:

```
print snow('Fall');
```

The resulting HTML looks like this:

```
<snow>Fall</snow>
```

Dynamic Pages and Forms

Just as standard HTML elements are available through functions in the CGI module, so are form elements. For example, tags to start and end forms, code for text fields and other input fields, and code for buttons are available as CGI module functions. The syntax for these functions is the same as the syntax for calling HTML tag functions. Consider the code shown in Listing 1-3.

Listing 1-3. *Code to Accept Input with the CGI Module*

```
#!/usr/bin/perl -T

use strict;
use CGI qw/:standard/;

print header,
start_html('Hello'),
start_form,
"Enter your name: ",textfield('name'),
submit,
end_form,
hr,
end_html;

exit;
```

When the code is saved to a web server and viewed through a browser, it should look like Figure 1-3. It looks like a regular form, but don't bother filling in the name and clicking on submit yet. There's no code to handle it!

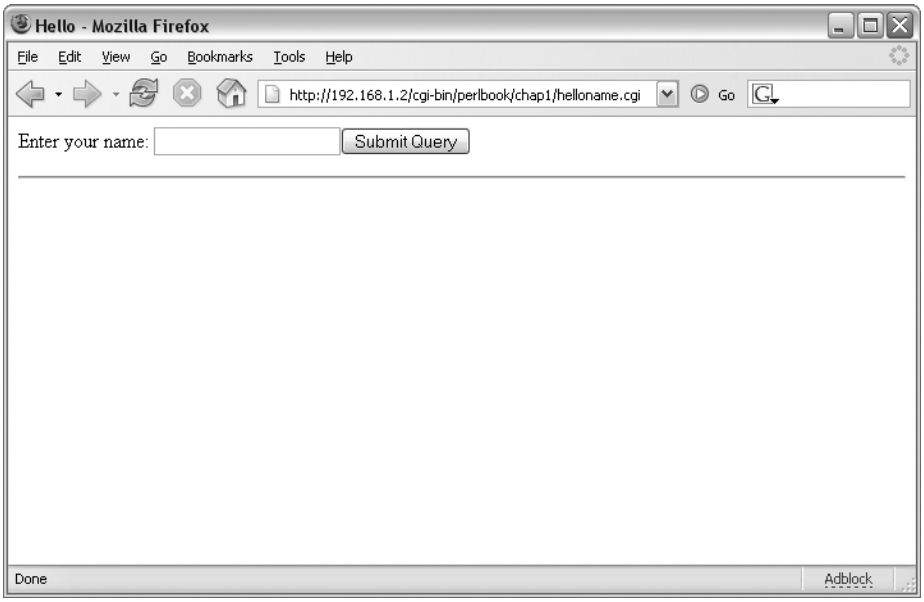

Figure 1-3. *A form created with CGI module functions*

Viewing the source code for the resulting HTML page reveals what this bit of Perl code did in the background:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"><head><title>Hello</title>
</head><body><form method="post" action="/cgi-bin/name.cgi"
enctype="application/x-www-form-urlencoded">
Enter your name: <input type="text" name="name" />
<input type="submit" name=".submit" /></form></body></html>
```

As you can see, the `start_form()` function inserted a `<form>` tag into the page. It automatically made the method a `POST` and set the action to itself—`"/cgi-bin/name.cgi"`, in this case. You could change both of these parameters by providing them within the call to the `start_form()` function, as in this example:

```
start_form(-method=>'GET',-action=>"/cgi-bin/someothercgi.cgi")
```

The code would then produce a form tag like this:

```
<form method="get" action="/cgi-bin/someothercgi"
enctype="application/x-www-form-urlencoded">
```

However, for this example, leave the form method and action at their defaults (`POST` and `/cgi-bin/name.cgi`).

Making the form actually perform some action means that you need to handle parameters within the program. To do so, you can simply check whether the `param()` function returns `true`.

```
if (param()) {
#do something
}
```

Listing 1-4 shows a more complete function added to the code shown in Listing 1-3.

Listing 1-4. *Printing the Name Input Using the CGI Module*

```
#!/usr/bin/perl -T

use strict;
use CGI qw/:standard/;

print header,
start_html('Hello'),
start_form,
"Enter your name: ",textfield('name'),
submit,
end_form,
hr;

if (param()) {
  print "Hello ",
  param('name'),
  p;
}

print end_html;

exit;
```

■ **Tip** The basic `if (param())` type of check is commonly used to determine whether to serve the default page or whether to process the CGI parameters. For example, if there are parameters within `param()`, then the CGI will output "Hello," along with the value that was filled in for the text field that is contained in `param('name')`.
