Hello World

An introduction to computer programming

Scared yet? Well don't be. Computer programming scares beginners away for two reasons: the massive "introduction" to computer programming tomes, and the beginners themselves. With your help, I would like to confront both issues.

If I said that you could read a 2092 page "Gray's Anatomy in 21 days" to become a proficient surgeon, would you believe me? Would you blame yourself if you tried and failed? I hope you answered an emphatic "no" to both questions. Similar statements plague introductions to computer programming. Unfortunately, many early programmers sit down with huge books to learn programming all at once. With your help, I would like to *introduce* you to computer programming

If I told you that most people most people fail to learn computer programming because it is too simple, would you believe me? This time you should. Most people fail to learn computer programming because they try to do too much at once. A computer does anything you tell it to. Perfectly. But that's it. Computers never "get it" and run what you meant to type. If you want to make your computer do something, you tell it with a language that uses only small, precise, and un-ambiguous commands. With your help, I would like to help you begin to understand one of the small, precise, and un-ambiguous languages. In fact, I would like to help you begin to understand all of them.

Why "Hello World?"

In 1973, Brian Kernighan wrote the paper, "A Tutorial Introduction to the Language B." In it, Kernighan wrote several small B programs to illustrate important concepts. Many of these included obscure ways to print "Hello World" to the screen.

Tradition took over, and now almost every programming book includes a variation of these programs. Unlike the originals, the current "Hello World" is the simplest useful computer program – one that writes the sentence "Hello World." to the screen. In this introduction, I plan to give you the simplest possible introduction to computer programming that allows you to create something useful.

If history interests you, the "B" language quickly transformed into the very popular "C" language. Several years later, "C" evolved into one of today's most popular languages: C++.

Javascript: your first programming language

To introduce you to computer programming, we'll use what is perhaps the most simple and useful language on the internet: *Javascript*. We'll use Javascript for two reasons: Javascript is an *interpreted language* with a very common *interpreter*, and it is also very *high-level*.

Interpreted languages and your Latin friends

Programming languages come in two varieties: *interpreted* languages, and *compiled* languages. To illustrate the difference, imagine that you have a group of friends that speak only Latin. Imagine, also, that you speak only English. To talk with your friends, then, you pay an English-Latin translator to help you.

This translator helps you by writing everything you say on paper – but in Latin. You then give this transcription to any of your Latin friends. Since some of your Latin speak different dialects, you ask your translator to write many copies of what you want to say. He or she must write each copy a little bit differently, but say the same thing.

In computer programming terms, this is similar to a "compiled" language. C and C++ are two examples of compiled languages. Computers understand only a very low-level language (called "machine language" – Latin in our example.) To give your computer program to other people you must use a second program, a "compiler," to translate your instructions into the native "machine language" for their type of computer. Like a compiler, our translator makes this conversion between English to Latin for us. This translated form is called an executable. Unfortunately, compiled programs are "platform specific" and run only on the specific type of computer you target them for: Windows PCs or Macintosh computers, for example. Each platform understands a different dialect of machine language so you must compile your program for each type of machine you want to run on.

Instead of asking your translator to write everything you say on paper, you always have the option to bring him or her with you. Your assistant translates everything you say as you say it. In this role, we commonly call them interpreters.

In computer programming terms, this maps almost directly to an "interpreted" language. Python and Perl are two examples of interpreted languages. Interpreted languages translate your program to machine language much later than compiled languages do. In fact, interpreted languages translate your program into machine code as your program runs!

Each type of language has both advantages and drawbacks.

Type of language	Pros	Cons
Compiled	- Run faster than the equivalent interpreted program. - Create programs that you can distribute easily to a *single platform*.	- Require that you compile your program for each platform you want to install it on. - Sometimes require that you write different versions of your program to run on different platforms.
Interpreted	- Create programs that run on any platform with the interpreter installed. - Are much easier to develop and test with: do not require that you use your computer's command-line interface.	- Require an interpreter - Might have errors because of programming mistakes ("bugs") in the interpreter, not your program.

Although the biggest disadvantage of an interpreted language is its interpreter, we will be using Javascript mainly because of the popularity if its interpreter: a web browser. Throughout this piece, you'll write programs that you can add to any web page – a personal homepage, for example. By the end of this piece, you should be able to sit down with a good Javascript reference and make your own programs, too.

As a side note, Javascript relates only vaguely to the popular programming language, Java. To its credit, the Java language is much more powerful than Javascript. As programming languages go, it is also easy to learn. Unfortunately, it carries with it much of the baggage of a compiled language. Given these features, it is too complex for an *introduction* to computer programming. However, compiled languages are the most popular (and used) languages because they do not require an interpreter.

Distancing yourself from the computer

The earliest programming languages barely made it easier to write programs in than machine language itself. In fact, "assembly language" is simply a set of mnemonics for the equivalent machine-language instructions. Low-level languages talk in terms of “registers,” “memory alignment,” and “interrupts.” Almost every term directly relates to a concept from the hardware that supports it. We call languages like assembly "low level" languages because they abstract very little: they mire you in the details of even the smallest task.

"High level" languages like Javascript make a programmer's life much easier. High-level languages talk in terms of “arrays,” “functions,” and “abstraction.” Programs that you write in Javascript can often be several hundred times shorter than those written in assembly or even C++.

The tools you'll need

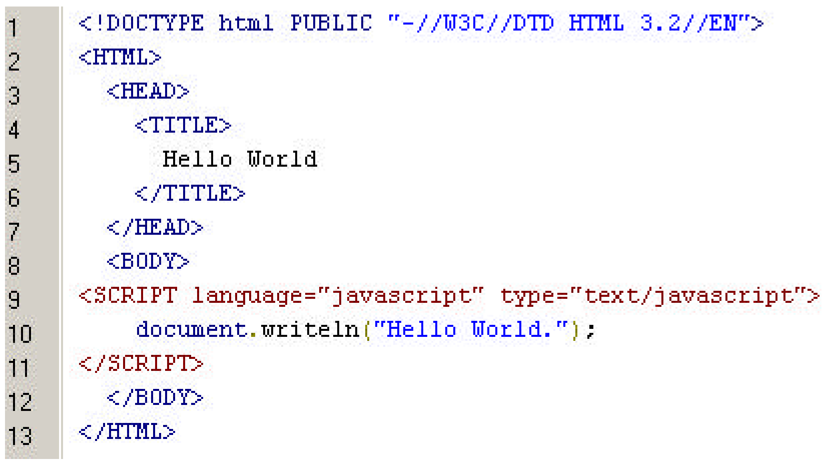
Now that we've chosen Javascript as your first programming language, what tools do you need to start programming?

1: A text editor

We'll be writing our programs / web-pages in a "text editor." Text editors let you create files without all of the special formatting information that word-processing programs add.

One text editor that comes with every Windows PC is called "Notepad." Although it lacks almost every feature that good programmer's text editors have, it is an option. To run Notepad, use Windows' start menu and navigate to **Programs | Accessories | Notepad.**

As a much better alternative, check out one of the many free programmer's editors from the internet. A search on Google (http://www.google.com) for "free programmer's editor" should start you off.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
  <HEAD>
    <TITLE>
      Hello World
    </TITLE>
  </HEAD>
  <BODY>
<SCRIPT language="javascript" type="text/javascript">
    document.writeln("Hello World.");
</SCRIPT>
  </BODY>
</HTML>
```

In your search, look for the following features:

- **syntax highlighting**: good editors colour the words in your program differently depending on their syntax. For example, the editor might colour functions black, but output strings in blue.
- **auto-indentation**: programmers always indent the statements in their programs to make the code both visually appealing and easy to read. Audto-indenting does much of this for you.

- **macro recording**: macros help you automate repetitive programming tasks. Many programmer's editors even let you write programs to extend or customize their functionality to your needs!
- **bookmarks**: bookmarks help you efficiently jump around your code when your programs get long.

An excellent free editor that has these features (and more) is called ConTEXT from http://www.fixedsys.com/context/.

2: A web browser

As I mentioned earlier, the most important requirement of an interpreted language is its interpreter. The Python language, for example, requires that you find and install the Python interpreter for your platform.

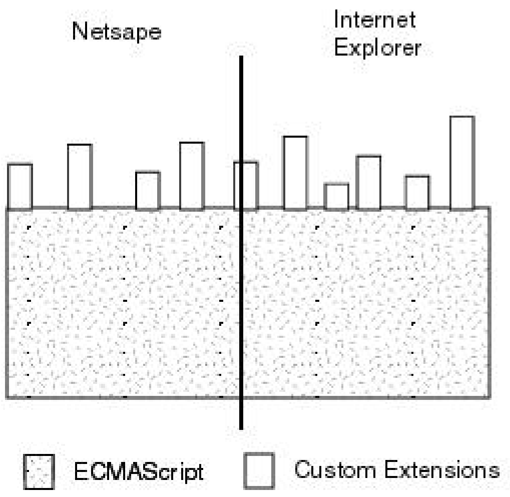
Fortunately, almost every web browser includes a Javascript interpreter. Internet Explorer, Netscape Navigator, and Opera are the three most popular. Since you'll write programs that run in most web browsers, you'll be writing programs that run on most web pages too.

3: A Javascript reference

This tutorial aims to introduce you to computer programming, not teach you every detail of the Javascript language. Even if you do read the entire ECMAScript specification, you can hardly be expected to memorize it. This is where a good reference helps.

Perhaps the weakest point of Javascript is its lack of standards. Netscape introduced Javascript in 1995. They designed the language to help web-developers make their pages look better in a Netscape browser. In 1996, Microsoft responded with a Javascript-like language called JScript. As each company released newer versions of their web browsers, they added their own bits of functionality to the language.

The ECMA standardization association partly sealed this rift with the 1999 release of standard number 262, also known as ECMAScript. The ECMAScript specification defines a *minimum* language very similar to Netscape's Javascript version 1.1. Microsoft claims that Internet Explorer 4+ conforms to the ECMAScript specification, and Netscape claims that Navigator 6 does as well. However, both browsers continue to add custom functionality on top of their base ECMAScript implementation..



Given the differences in their implementation, we must remain aware that the same Javascript may not execute the same (or at all) in different browsers. With the help of the following references, you should be able to quickly find information on any of Javascript's specifics. The cross-reference tells you if your Javascript is standard or browser-specific Javascript.

Javascript 1.1 specification:
http://home.netscape.com/eng/javascript/index.html

ECMAScript specification:
ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf

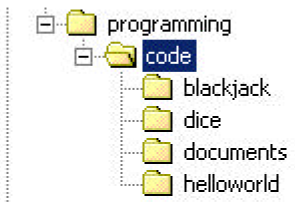
Current Netscape specification:
http://developer.netscape.com/docs/manuals/communicator/jsref/index.htm

Current Internet Explorer specification:
http://msdn.microsoft.com/scripting/default.htm?/scripting/JScript/doc/jsoprNot.htm

Nescape / Internet Explorer cross-reference:
http://www.coolnerds.com/xrefs/xrefjsom.htm

4: A working directory

Although you can store your programs anywhere on your hard-drive, I would strongly recommend that you store them in separate directories under a "programming" directory. Since anything but the smallest projects span multiple files, this structure helps separate your programming projects from each other.



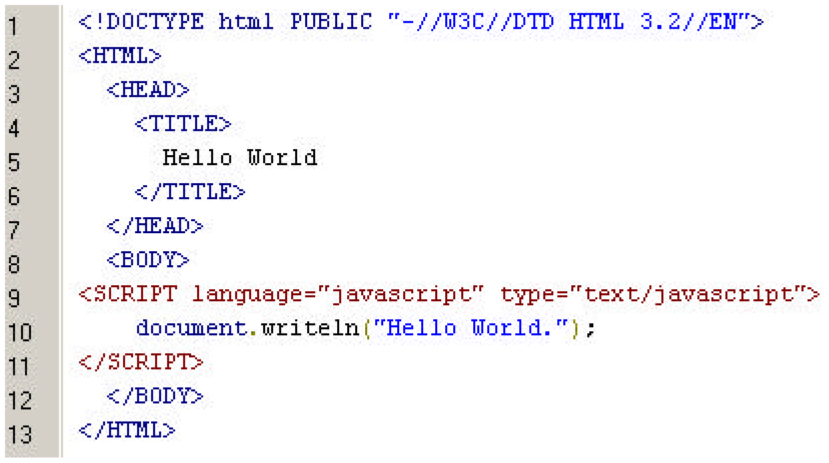
If you want to publish your projects to the web, I would also recommend that you make your directory names “web-friendly.” Web-friendly names use only the lowercase alphabet, dashes, and numbers. You should avoid spaces and special punctuation. If you don’t, Web browsers translate folders named like “Hello World” to “www.yoursite.com/Hello%20World”. You should avoid capitalization because users find it harder to memorize a strangely capitalized web addresses than a strictly lowercase one. It matters on most web servers.

5: An HTML editor (optional)

If you decide to incorporate some of your Javascript into larger web pages, you might find it difficult to hand-code the web page (not the Javascript.) If you’re in this position, there are many great free HTML editors that let you visually edit your pages. This type of editor is known as a “what you see is what you get” (WYSIWYG) editor. A search on Google (http://www.google.com) for "free html editor wysiwyg" should start you off. This tutorial does not assume that you have an HTML editor – only a text editor.

Your first program

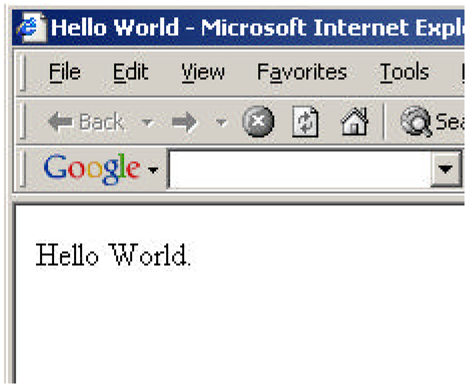
Let’s go over the famous “Hello World” program that you just saw in the “text editors” discussion.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
  <HEAD>
    <TITLE>
      Hello World
    </TITLE>
  </HEAD>
  <BODY>
<SCRIPT language="javascript" type="text/javascript">
    document.writeln("Hello World.");
</SCRIPT>
  </BODY>
</HTML>
```

To run your first program, type the above code (without the line numbers) into your text editor. It is a *very* good habit to save your work every few minutes. That way, when your computer crashes (and it will,) you only lose a few minutes of work. When you save your program, save it into your working directory, under a "helloworld" subdirectory. Call it "helloworld.html"

Next, browse to (and open) your file. Once you've found your program, double-click it. You should see a web page appear with the title "Hello World" and the page content "Hello World."


Hello World - Microsoft Internet Expl
File Edit View Favorites Tools
Back
Google

Hello World.

There are two types of programming statements in this (and any other) computer program: template code, and custom code.

Template code

Template code is the minimum amount of programming (coding) to create a valid program. This is typically a "Hello World" program without the "Hello World." In our example, every line except line 10 is template code. In addition, only lines 9 and 11 are template code for our Javascript: the rest is for the web page that contains our Javascript. Let's go through what each line of template code means:

1: Although not strictly necessary, this line is good programming style. It tells our browser that the web page conforms to the HTML version 3.2 specification.

2 … 4: Words inside of angled brackets (<>) are called HTML *tags*. They optionally contain a forward-slash (/). *Closing tags* start with a forward-slash, while *opening tags* do not. An opening tag "starts" something on a web page (italics, bold, or a paragraph for example,) while a closing tag ends something on a web page. So lines 2 through 4 start the web (HTML) page, the header of the web page, and the title of the web page.

5: Since it sits between the opening and closing TITLE tags, this line sets the title of the web page.

6 … 7: These lines end the title of the web page, then the header.

8: This line starts the body of the web page.

9: This line starts a section of script. The section, *language="javascript",* tells the browser that we're about to write some Javascript.

11: This line ends our section of script.

12 … 13: These lines end the body of the web page, then finally the web (HTML) page itself.

The great thing about template code is that you can simply copy and paste it for your next program. Since it will be the same for each program, you need not even remember what it does!

The remainder

After template code, we don't have much of our program left to discuss!

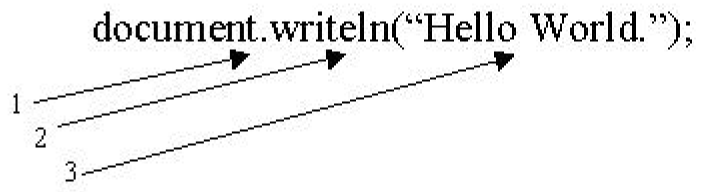
Line 10 tells Javascript to write "Hello World." to our web page. How does "document.writeln(…)" mean "Write … to the web page"?

Objects

Javascript break browsing sessions into several entities called "objects." Just as one "object" in a kitchen might be a fridge, one "object" in a browsing session is a "document." In line 10, we're using the "document" object. Some of the other objects that Javascript understands are Browser, Window, and Form. Other, more powerful languages (including Java,) use objects much more extensively. It's called "Object-Oriented Programming." Although it is extremely beneficial, it is much too complex to deal with in any more detail.

Methods

Objects wouldn't be very useful if they did nothing. Because of this, almost every object has a set of words ("methods") for you to command it with. If your computer portrayed it , a "fridge" object would have an "open" command (method.) Similarly, a "document" object has a "writeln" ("write line") method.



From a code perspective, methods have three main characteristics:

1) A dot (.) to separate them from the object that owns them.
2) A name followed by brackets. Ie: writeln()

3) A list of *parameters* to tell the method *what to do*. You place the parameters inside of the method's brackets. Ie: the string, "Hello World."

For more information on functions and methods, see the section in "Programming Concepts."

Writing valid programs

Any computer program consists of two major elements: its *syntax*, and its *semantics*.

Semantics

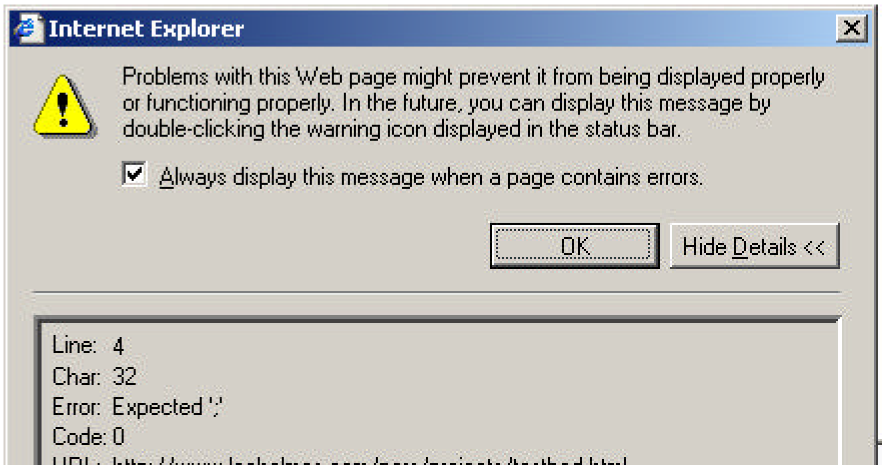
In spoken phrases, the semantics of a sentence tell us what it *means*. In programming, the semantics of a program tell us what it *does*. The programming term "bug" (or "defect") describes a piece of code that doesn't do what it should. This piece of code, then, has improper semantics. Although the topic of *semantics* is important, programmers rarely use the term.

Unfortunately, only programming experience teaches you how to write better programs. Even worse, the quest for perfect programming never ends. You only get better: never perfect. That is why we call it "the art of computer programming."

Unlike semantic correctness, we can easily learn and test for *syntactic* correctness.

Syntax

To make sense in any language (spoken languages included,) you must follow the language's grammar. "Sentence not this English is" is not a valid English phrase. It fails to follow several important rules of English. Like English, you must follow a certain set of rules to create valid Javascript phrases (programs.)



Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.

[✓] Always display this message when a page contains errors.

```
Line:  4
Char:  32
Error:  Expected ';'
Code: 0
```

Because Javascript's syntax is so well defined (and easily tested,) your web browser gives you an error whenever you load Javascript with incorrect syntax. Most often, your browser's error message is good enough to tell you what you've done wrong. The error message above tells us that line 4 of our program needs a semicolon.

Let's go over Javascript's *syntax rules*.

Programs must begin with "<script language="javascript">"

This is officially an HTML requirement, not a Javascript requirement. However, no Javascript will run without this phrase. This is part of our template code.

Programs must end with "</script>"

Like the statement above, this is an HTML requirement, not a Javascript one. This, too, is part of our template code.

Javascript is case sensitive

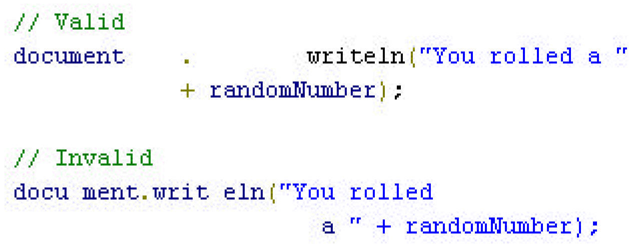
Do you remember when we wrote to our web page with the "writeln" method from the document object? Since Javascript is case sensitive, "WriteLn" does not work. Note that this does not apply to HTML phrases – the previous two rules included.

Javascript allows comments

Comments are sections of code that you've added to your program to make it easier to read and understand. For more information, see the "comments" section of "programming concepts."

Javascript ignores whitespace

Javascript ignores all spaces, tabs, and new lines in your program unless they interrupt a statement. Here is an example of both a valid and invalid use of whitespace.



```
// Valid
document    .        writeln("You rolled a "
            + randomNumber);

// Invalid
docu ment.writ eln("You rolled
                      a " + randomNumber);
```

Javascript programs are lists of statements separated by semicolons

We will go over these shortly, but we can break statements into one of several categories:

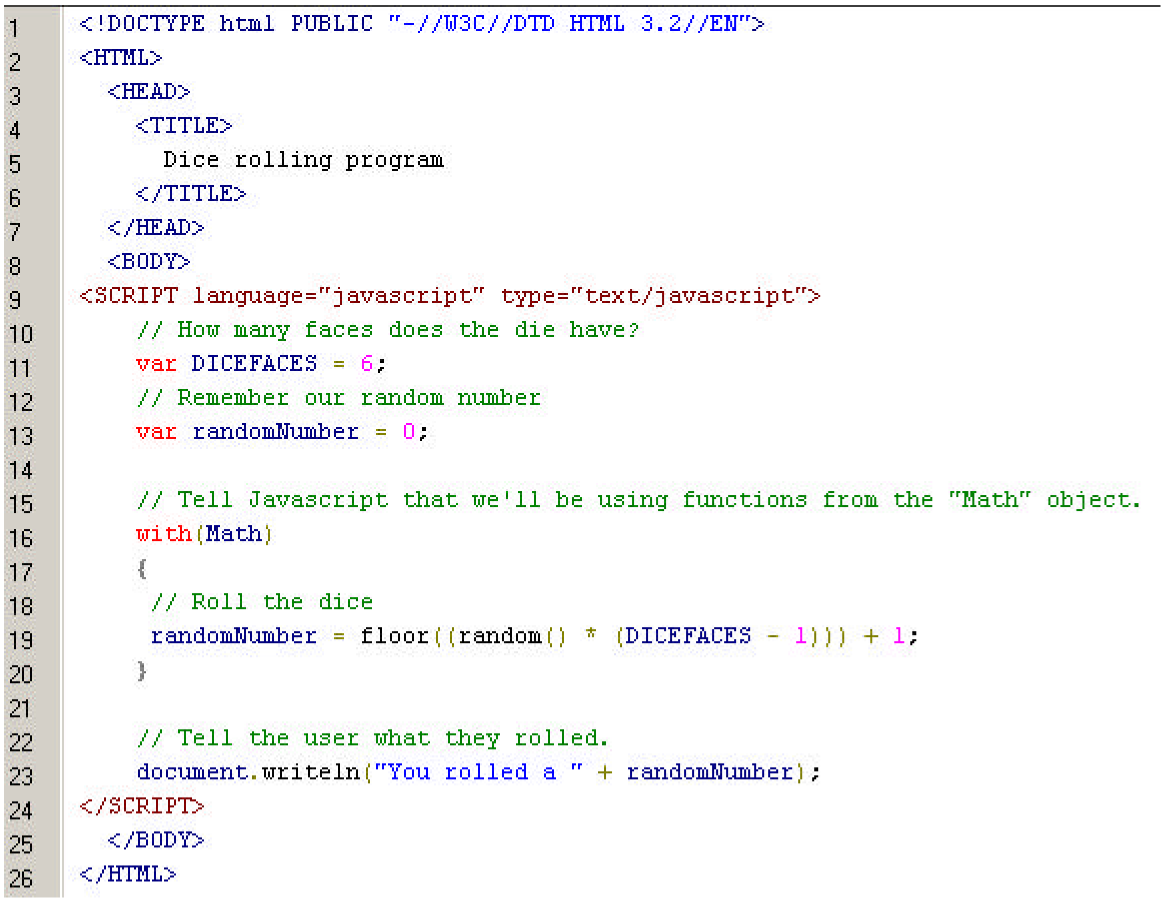
- code blocks (a group of statements between '{' and '}' symbols that Javascript treats as one statement)

- variable definitions (tell Javascript to put aside enough storage to remember some information for you)
- variable assignment (tell Javascript to actually put information in storage)
- method / function calls (tell Javascript to "do something" – ie: write 'Hello World' to the web page.)
- program logic / flow control (execute some statements *if* a condition is true)
- loops (execute the same set of statements more than once)

Our next program, a Javascript dice roller, shows some of these statements in action.

Description of "Dice"

Let's go over a more complex Javascript program to illustrate some of Javascript's rules of syntax.

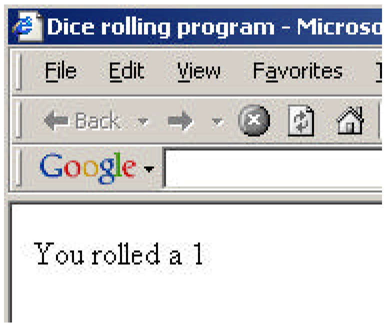


```
1   <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
2   <HTML>
3     <HEAD>
4       <TITLE>
5         Dice rolling program
6       </TITLE>
7     </HEAD>
8     <BODY>
9   <SCRIPT language="javascript" type="text/javascript">
10      // How many faces does the die have?
11      var DICEFACES = 6;
12      // Remember our random number
13      var randomNumber = 0;
14
15      // Tell Javascript that we'll be using functions from the "Math" object.
16      with(Math)
17      {
18       // Roll the dice
19       randomNumber = floor((random() * (DICEFACES - 1))) + 1;
20      }
21
22      // Tell the user what they rolled.
23      document.writeln("You rolled a " + randomNumber);
24  </SCRIPT>
25    </BODY>
26  </HTML>
```

Do you notice any similarities to "Hello World"? Of course! Template code saves us quite a bit of typing. Aside from the page's title (line 5,) our *custom* code runs only between lines 10 and 23. Even at that, most of the code is whitespace and comments!

To run this dice program, first copy the *template code* from "helloworld.html" and paste it into a new file. Next, type the custom code (without the line numbers) into your text editor. Again, don't forget to save your work every few minutes. When you save your program, put it into your working directory, under a "dice" subdirectory. Call it "dice.html"

Next, use Windows' Explorer to browse to your file. Once you've found your program, double-click it. You should see a web page appear with the title "Dice rolling program" and the page content "You rolled a" followed by a number between 1 and 6.



```
You rolled a 1
```

Lines 10, 12, 15, 18, and 22: The first thing you might notice about the program is the green sections that I've written in proper English. These are called *comments*. Comments describe sections of code to other programmers (or ourselves) that aren't particularly intuitive. Whenever the Javascript interpreter reads two forward slashes, it ignores them and anything to the right of them on that line. Alternatively, you can place your comments between the slash-star combinations, "/*" and "*/". This second type of comment can span multiple lines if you want. Comments should not restate the obvious. After all, the comment "define some variables" does little to enhance your program's readability.

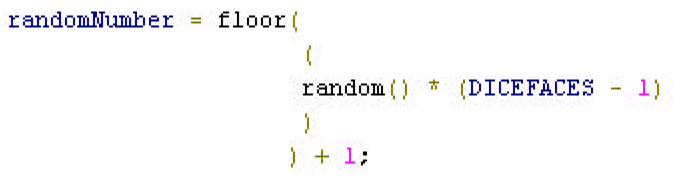
Lines 11, 13: The word *var* tells Javascript that we want it to remember a bit of information. We call this bit of information a *variable*. Notice that we didn't tell Javascript what *type* of variable we want (a number, or some money, for example.) This means that Javascript is a *weakly typed language*. Strongly typed languages require that you tell the computer exactly what kind of information you want to store.

Also notice that these statements end with a semicolon. Except for comments, conditional, and looping statements, you should end every Javascript *statement* with a semicolon. See the section, "Programming Concepts" for clarifications on this.

As a programming convention, variables in ALL UPPERCASE are called *constants*. Constants help make our code both easy to read and easy to maintain. Code without constants is said to use "magic numbers." Although they mean the same thing to Javascript, would you rather read the equation "2 x PI x R" or "2 x 3.14159265359 x R"? Also, if we use this number throughout our program, we can change a constant's definition much easier than we can change every place that we use its value.

Line 16: The statement *with(…)* tells Javascript that we're lazy. We're saying that there are certain methods (random(), and floor() in this example) that belong to a certain object (Math). However, we don't want to type "Math.random()" so just let us type random(). Note that this laziness applies only to statements inside the curly braces ({ and }).

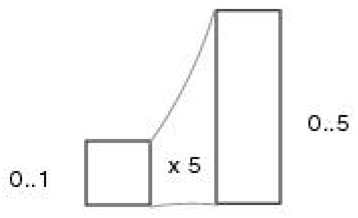
Line 19: This line packs quite a punch! Let's break it down even further.



```
randomNumber = floor(
                      (
                     random() * (DICEFACES - 1)
                      )
                     ) + 1;
```

Although it looks complex, this line really breaks into a few method calls and a little bit of math.

1) *randomNumber = …*: This is called a variable *assignment*. Do you remember when we used the *var* word to ask Javascript to store our bit of information? Now we're telling Javascript that we'd like the randomNumber variable to store the results of …
2) *floor(…)*: The outer wrapper of most of this line is the *floor()* method. Remember, we're inside a "with(Math)" block, so we're actually calling the Math.floor() method. Math's floor() method takes a number as a *parameter* and returns the *floor* of that number as a *result*. If you are unfamiliar with the term, *flooring* a positive number essentially rounds it down. The parameter that we're passing to floor is …
3) *random() * (DICEFACES – 1)*: Again, we're in the "with(Math)" block, so random() is a method from the Math object. According to Chaper 4 of Netscape's Javascript Reference, Math.random() returns a decimal number between 0 and 1. So how do we convert this to a random number between 1 and 6? Easy! First, find out a number between 0 and 5. We can do this by multiplying the return of random() by 5. The DICEFACES constant, as defined earlier, does quite a bit to make our code more readable. Note that the Javascript term for multiply is "*"
4) *+ 1*: Finally, we add 1 to the result of the floor. This converts a number between 0 and 5 to a number between 1 and 6.



A common mistake in this last bit of code would have been to multiply the random number by the number of faces on your die. Unfortunately, this would give you a dice roll between 0 and 6!

Line 23: We've already seen the document.writeln() method in action. Unlike the previous usage, this line seems to mix addition with sentences (called strings in Javascript.) Fortunately, Javascript offers a very useful feature called *string concatenation.* That is, Javascript makes a new string out of two smaller strings if you separate them with a "+". In this case, the writeln() method thinks we're passing it the single string "You rolled a 5" if in fact we did.

Questions

At this point you're probably asking the question, "How did you know what methods to use?" or more likely, "How did you know how to make that program?" That's a completely valid question. It all comes down to how well you know your toolbox.

Take, for example, an apprentice carpenter who has just been introduced to his or her first tool: a hammer. They can nail boards together with abandon but lack the skills to make a house. Luckily, apprentice carpenters have master carpenters to introduce them to the proper tools of their profession: the band saw, nail gun (wow! There's a more efficient way?), and many more.

Early programmers find themselves in a similar situation. You know how to print to the screen, but soon tire of writing "Hello World" programs. That's where your trusty reference material saves you. I mentioned earlier that you're wasting your time if you memorize every method of every class. However, you should *glance* over your toolbox at least once so you know what's available to you. I recommend that you skim chapters 2 through 7 of Netscape's Javascript reference or the description column of the Coolnerds cross-reference to see what Javascript offers.

Programming Concepts

Although you might be aware of the tools that your language provides, I still haven't introduced you to the statements that control those tools. After all, we still need ways to express common concepts like repetition ("hammer that wall until lunch") and conditionals ("if you're tired then hammer slower.")

Although they differ slightly in their implementation, almost every recent programming language supports the following concepts: *variables*, *if / else, switch*, *for*, *comments*, *functions*, *return*, *while*, *break*, *continue*, *do … while*, and *operators*. Again, quickly glance over the Netscape Javascript reference (chapter 3) to familiarize yourself with additional features.

Variables

Computers use variables to store information for us. In general, you create one of five types of variables: *integers*, *floats*, *strings*, *booleans*, and *arrays*. As I mentioned earlier, Javascript is a weakly typed language. Because of this, you do not have to tell Javascript what type of variable you want to store – simply use the 'var' statement. Try to name your variables descriptively. For example, use the variable name "menuSelection" instead of the cryptic "m." All languages have rules about what letters you can use in variable names, and how long your variable names can be. For the specifics, see one of your handy references.

Integers / Floats

The difference between an *integer* and a *float* ("floating point number") is that *floats* also store decimal places. It sounds simple, but the computer makes a very clear distinction. Most programming languages break the distinction even further by letting you say how big of a number you want to store. With these additional distinctions, you'll often see the terms "byte," "short," "int," and "long" describe integers. Likewise, you'll often see the term "real," "float," and "double" describe floating point numbers.

<u>Javascript examples:</u>

```
var myAge = 23;
var PI = 3.14159;
```

Strings

Strings store sequences of letters, numbers, and possibly other characters. You normally surround strings with single (') or double (") quote characters. Examples include "Hello world", "¦ ¦ ¦ ¦ ¦ ¦ ¦ ", and words that the multi-language "Unicode" convention supports. Some languages define strings as "char[]" or "char*", stemming from the way that C and C++ store strings as *arrays* (lists) of characters.

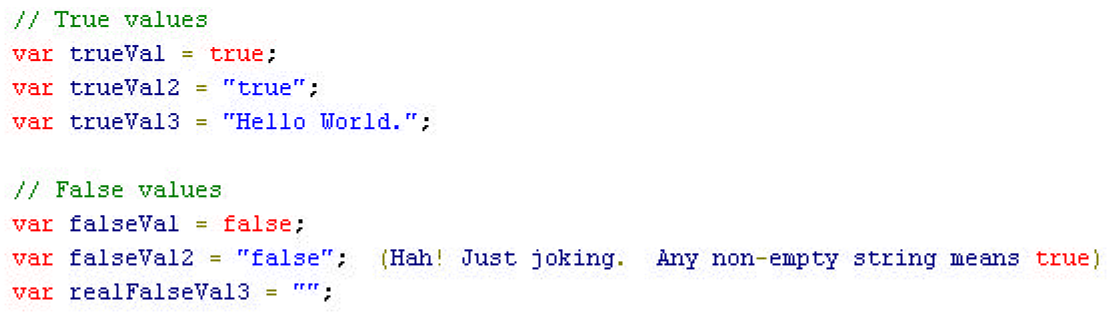
<u>Javascript examples:</u>

```
var myFirstString = "Hello World.";
var statusLine = "…-----=== Press a Key ===----…";
```

Booleans

Boolean variables are those that represent "true" or "false" values. Many languages do not directly support boolean variables, but instead use some sort of mapping. For example, "1" might represent "true" and "0" might represent "false." Javascript uses the empty string ("") and the value "false" to represent false values. Everything else works out as "true."

Javascript examples:



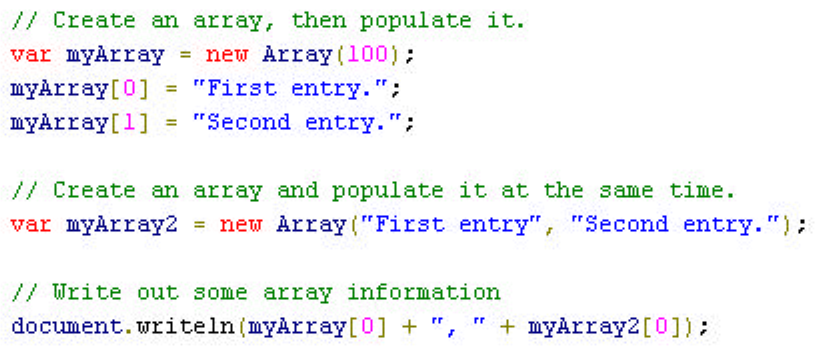
```
// True values
var trueVal = true;
var trueVal2 = "true";
var trueVal3 = "Hello World.";

// False values
var falseVal = false;
var falseVal2 = "false";  (Hah! Just joking.  Any non-empty string means true)
var realFalseVal3 = "";
```

Arrays

Arrays hold *lists* of variables to give you a convenient way to access them. Rather than create 100 "phone number" variables, you can create a single "phone number" array to hold 100 numbers. Usually, all of the variables must be of the same type (number, string, etc.) Most languages use the '[' and ']' symbols to define an array, and nearly every language uses those symbols to access the elements of an array. Javascript lets you mix the types of variables in an array.

One tricky point about arrays in most languages is that they start from zero, not one. They are "zero-based." This means that you access the *first* element of *myArray* with "myArray[0]".

Javascript examples:



```
// Create an array, then populate it.
var myArray = new Array(100);
myArray[0] = "First entry.";
myArray[1] = "Second entry.";

// Create an array and populate it at the same time.
var myArray2 = new Array("First entry", "Second entry.");

// Write out some array information
document.writeln(myArray[0] + ", " + myArray2[0]);
```

If / Else

"If-Else statements" give us the power to make our program "branch." They let our program behave differently depending on the results of a test. This "test" is simply a boolean value: true or false.

"If-Else statements" are called *conditional statements.* They operate differently in different conditions. Like all other conditional statements, you can follow them with either a single statement or block of code. Conditional statements should not have a semicolon at the end of the line, although the code directly underneath should.

If-else ladder

We can string together "if (…) else if (…) … else if (…) else (…)" statements to create what is known as an "if-else ladder." With this structure, our program executes only the first statement (or block of code) that passes its boolean test. We usually use the final "else" statement as a "catch-all" to execute when none of the above tests pass. Of course, anything past the first "if" statement is optional. The code below shows an example if-else ladder.

Code blocks

We surround statements with the symbols, '{' and '}' to tell our language that those statements are a block of code. Blocks of code cause our programming language to treat them as *one statement*. Rather than have a conditional statement execute only one line of your program, code blocks let you execute many.

Although the statements *inside* the block of code need semicolons at the end of the line, the block itself does not.

<u>Javascript example:</u>

```
var iAmAGreatProgrammer = true;
var allProgrammersAreGreat = false;

if(allProgrammersAreGreat)
   document.writeln("All programmers are great.");
else if(! iAmAGreatProgrammer)
   document.writeln("I am not a great programmer");
else
{
   document.writeln("I'm a great programmer!");

   // Great programmer's aren't conceited
   iAmAGreatProgrammer = false;
}
```

Case

Case statements work very much like an if-else ladder. They simplify the very common task of performing different operations based on one of many possible values of a variable. For example, you might use a case statement to process a user's menu selection. Most languages restrict this comparison to numeric variables, but Javascript does not.

We create a case statement in three parts: the *switch* (the variable being tested), *case statements* (operations based on a possible values of that variable), and perhaps a *default* statement (the catch-all clause.)

The switch

The *switch* statement tests the variable that you write against each of the following *case statements*.

The case statement

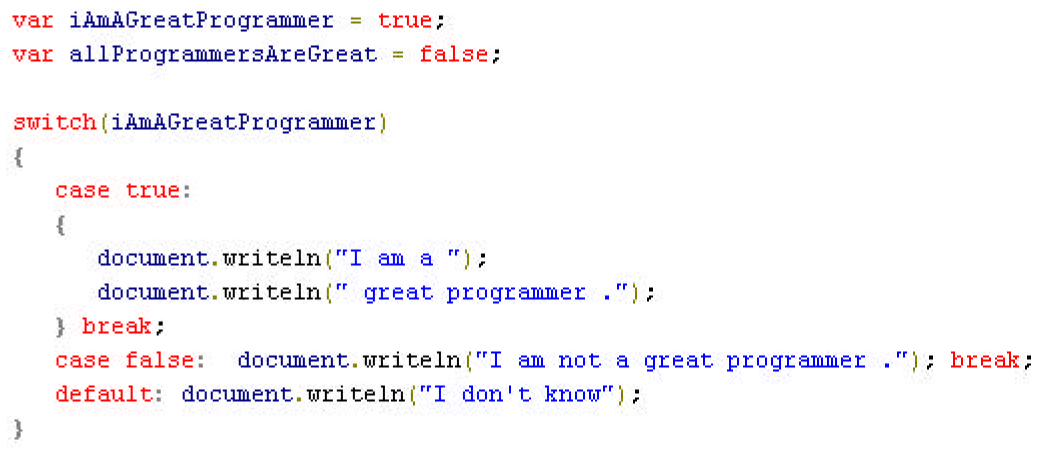
"Case statements" tell the computer "In the case that my variable is (…), then execute this statement." Like other conditional statements, you can always replace "this statement" with a block of code containing many statements.

Tricky point: You must finish your statement (or block of code) with a "break;" statement if you want to exit from the entire case statement. Otherwise, Javascript continues to process (without testing) the rest of the commands. This is an obscure feature known as a "fall-through."

The default

As you might have guessed, Javascript calls your *default* case when none of the other case statements match. You can leave out this case if you want Javascript to take no action at all.

Javascript example:



```
var iAmAGreatProgrammer = true;
var allProgrammersAreGreat = false;

switch(iAmAGreatProgrammer)
{
   case true:
   {
      document.writeln("I am a ");
      document.writeln(" great programmer .");
   } break;
   case false:  document.writeln("I am not a great programmer ."); break;
   default: document.writeln("I don't know");
}
```

For / break / continue

Like switch statements, “for-loops” use a test to help control the flow of your program. Unlike switch statements, though, they execute the same statement (or block of code) until the test is *false*.

We can break a for-loop into three parts: the *initializer*, *conditional*, and *iterator*. We separate each part with a semicolon.

The initializer

Programming languages call the initializer of your for-loop just as they starts to process the for-loop If you write a for-loop to run a specific number of times (known as a *counted for-loop*), you normally use the intializer to declare a variable and set its value to 0. For example, “for (**var counter = 0;** counter < 10; counter++).”

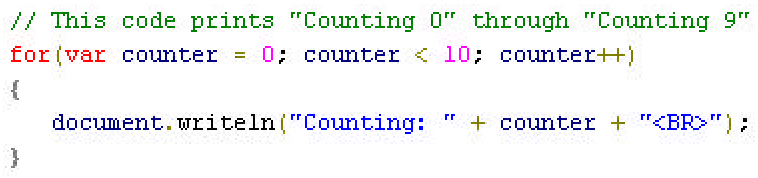
The conditional

Programming languages continue to execute the statement (or block of code) beneath your for-loop as long as the conditional evaluates to *true*. For example, “for (var counter = 0; **counter < 10;** counter++).”

The iterator

Programming languages execute the iterator of a for-loop *after* they execute the statement or block of code. If you write a counted for-loop, you’ll usually use this space to increment the variable you’re counting with. For example, “for (var counter = 0; counter < 10; **counter++**).”

Javascript example:



```
// This code prints "Counting 0" through "Counting 9"
for(var counter = 0; counter < 10; counter++)
{
   document.writeln("Counting: " + counter + "<BR>");
}
```

Aside from what looping statements already offer, we have two more statements to help control a looping statement. These are *continue* and *break*.

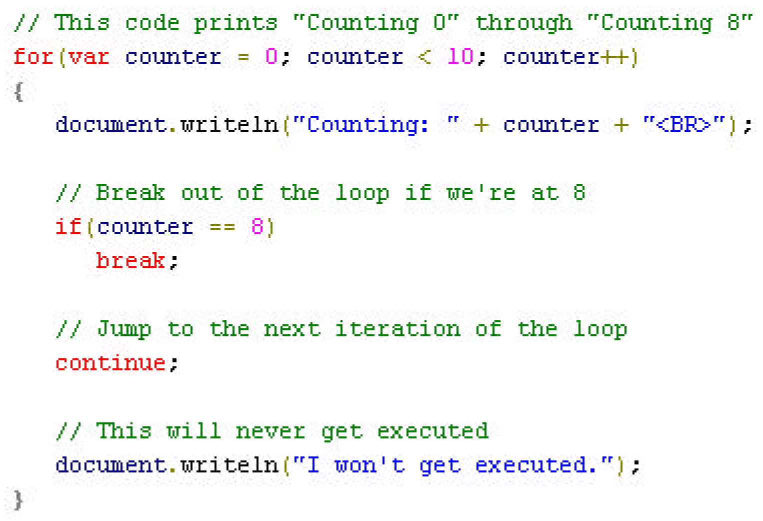
Continue

The *continue* statement tells our programming language to stop processing our block of code. Instead, the programming language starts the next iteration of the loop. When it starts the next loop, it executes the iterator then re-evaluates the conditional.

Break

Break statements break out of a loop or block of code. When your programming language encounters a break statement, it simply jumps to the end of your block of code and continues processing.

Javascript example:



```
// This code prints "Counting 0" through "Counting 8"
for(var counter = 0; counter < 10; counter++)
{
   document.writeln("Counting: " + counter + "<BR>");

   // Break out of the loop if we're at 8
   if(counter == 8)
      break;

   // Jump to the next iteration of the loop
   continue;

   // This will never get executed
   document.writeln("I won't get executed.");
}
```

Comments

As I mentioned earlier, comments are sections of code that you add to your program to make it easier to read and understand. Most languages support two types of comments: line comments and block comments. As with other languages, Javascript ignores the comment characters and all text until the end of the comment.

In Javascript, *line comments* start with a "//" and end at the right-hand edge of the page. Because they end at the end of the page, they work only for one line. Some other languages use the hash (#) symbol to start a line comment.

In Javascript, *block comments* start with a "/*" and end with a "*/". Unlike line comments, they may span several lines.

Aside from that simple summary, the most important thing to say about comments it to *use them*. They may seem to be a nuisance at first, but they help focus your mind and describe the "high-level" flow of your program.

Functions / return

We've seen and used functions already, but not formally. When we discussed them earlier, we called them "methods." Functions let us create (and name) a block of code to perform some logical task. Rather than write the code to generate a random number each time we want to, we can simply call Javascript's Math.random() function. This function contains all the code we need to get a random number.

A function is made of four parts: *name*, *parameters*, *body*, and *return value*.

Name

A function's name is the short-form that we use to refer to the function's block of code. Always try to name your function to properly describe its purpose. Function names like "doIt()" or "process()" do little to improve your program's readability. Like variable names, all languages have rules about what letters you can use in function names and how long your function names can be. For the specifics, see one of your references.

Parameters

Most functions work with one or more *input* parameters. For example, the Javascript *document.writeln(…)* function writes its *string parameter* to the web page. We place parameters between brackets immediately after the function name. Of course, your function may take no parameters. In this case, you put nothing in between the brackets. Most languages require that you tell them the types of the parameters, but Javascript does not.

Body

The body of your function defines the code that you want this function to execute. You may use any of the *input parameters* during your code. An important point to note, however, is that you cannot generally modify these input parameters. If you do, the change remains local to your function only – and will not remain once your function exits.

Return value

Most functions of any usefulness return some sort of result. For example, the Javascript Math.ceil(…) method essentially returns a rounded-up version of its input. The Javascript Math.random() method returns a random number between 0 and 1. Strongly-typed languages require that you tell them what type of variable your function returns, but Javascript does not.

Javascript example:

```
// Calculate the area of a rectangle
function calculateArea(length, width)
{
   var area = length * width;
   return area;
}

// The length and width of our rectangle
var LENGTH = 10;
var WIDTH = 11;

// Write out the area of the rectangle
document.writeln("A rectangle with length " + LENGTH +
   " and width " + WIDTH + " has area: " + calculateArea(LENGTH, WIDTH));
```

While / do … while

The *while* and *do … while* loops are minor variations of *for* loops.

While-loops execute their block of code as long as their test holds true. Use a while loop when you have no need for the *initializer* or *iterator* portions of a for-loop. While-loops *do not* execute their block of code (even the first time) if the test holds false.

Javascript example:

```
document.writeln("We are preparing your report.  Please wait.");
while(preparingReport())
{
   document.writeln(".");
   updateNumberOfWaits();
}
```

Do-while loops slightly twist the while-loop concept in that they guarantee that the code block will run at least once. Use this type of loop when you want to continuously test the results of an operation that happens inside your block of code. Note that unlike the other while-statements, this one must end with a semicolon!

```
// Prepare to store a random number
var myRandom;

// Generate random numbers until we get one greater than
// (or equal to) 0.5
// Note that we need this block to execute at least once!
do
{
   myRandom = Math.random();
   document.writeln("Random number: " + myRandom + "<BR>");
}
while(myRandom < 0.5);
```

Operators

Operations are the simplest operations in a programming language. They are also the most essential.

Typical operations include comparisons (ie: <, >, ==,) arithmetic (ie: +, -, /, *, %,) and logic (ie: !, &&, ||.) Any good programming reference for a language lists and explains all of its operators. It should also list their relative *precedence*. In programming terms, *rules of precedence* tell a programming language the order to evaluate a list of operators in the same statement. Like the old mathematical precedence rule (BEDMAS – brackets, exponents, division, multiplication, addition, and subtraction,) programming languages follow similar conventions.

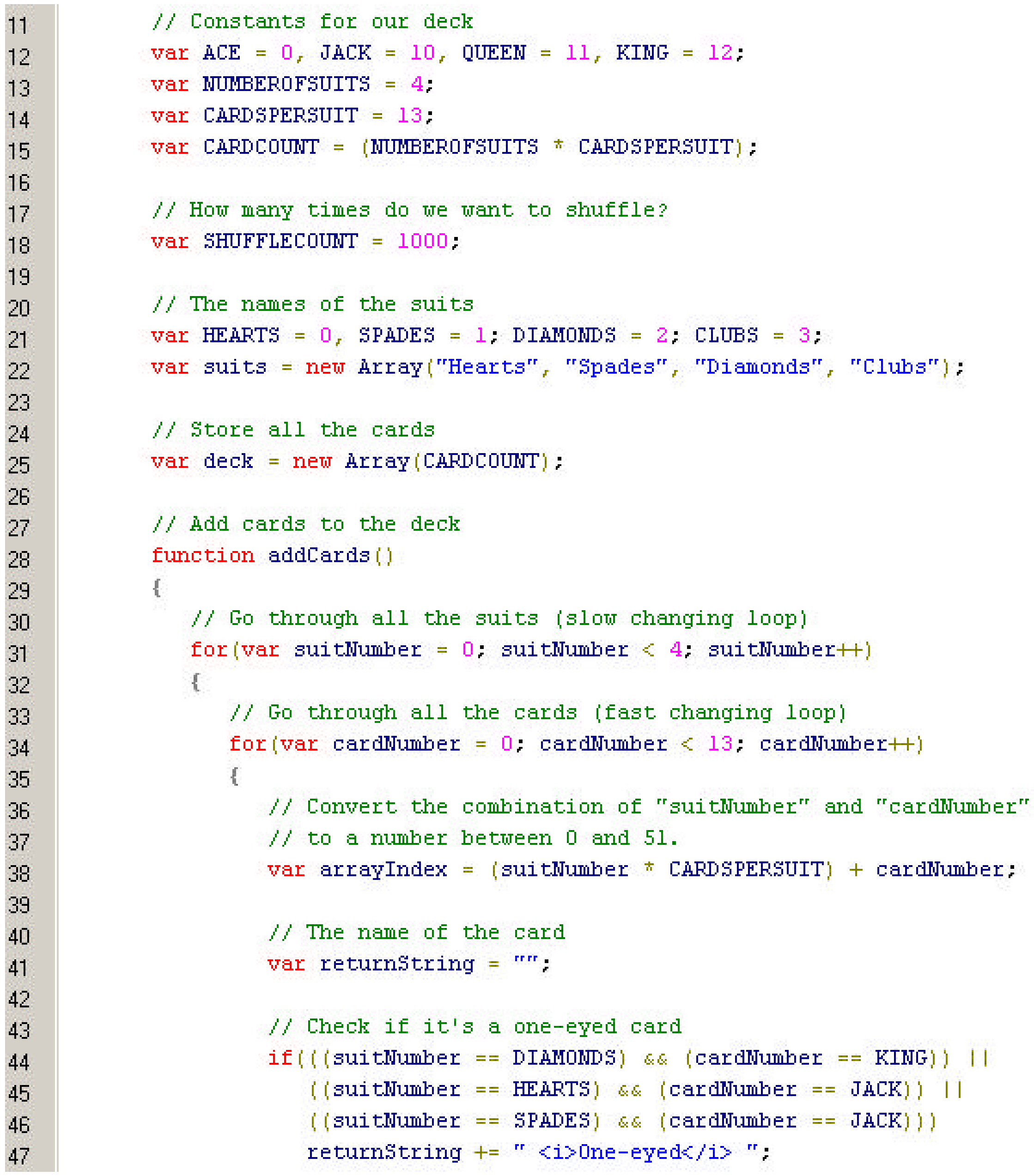
As an important note, language designers define operator precedence rules simply because a language cannot survive without them. In practical terms, good programmers *always* use brackets to identify their desired order of operations. They do this even when their desired order of operations match the language's precedence rules. We have better things to do in life than memorize (or expect other to memorize) long lists of precedence rules.

```
var myMath = (3 * 4) + 5;
```

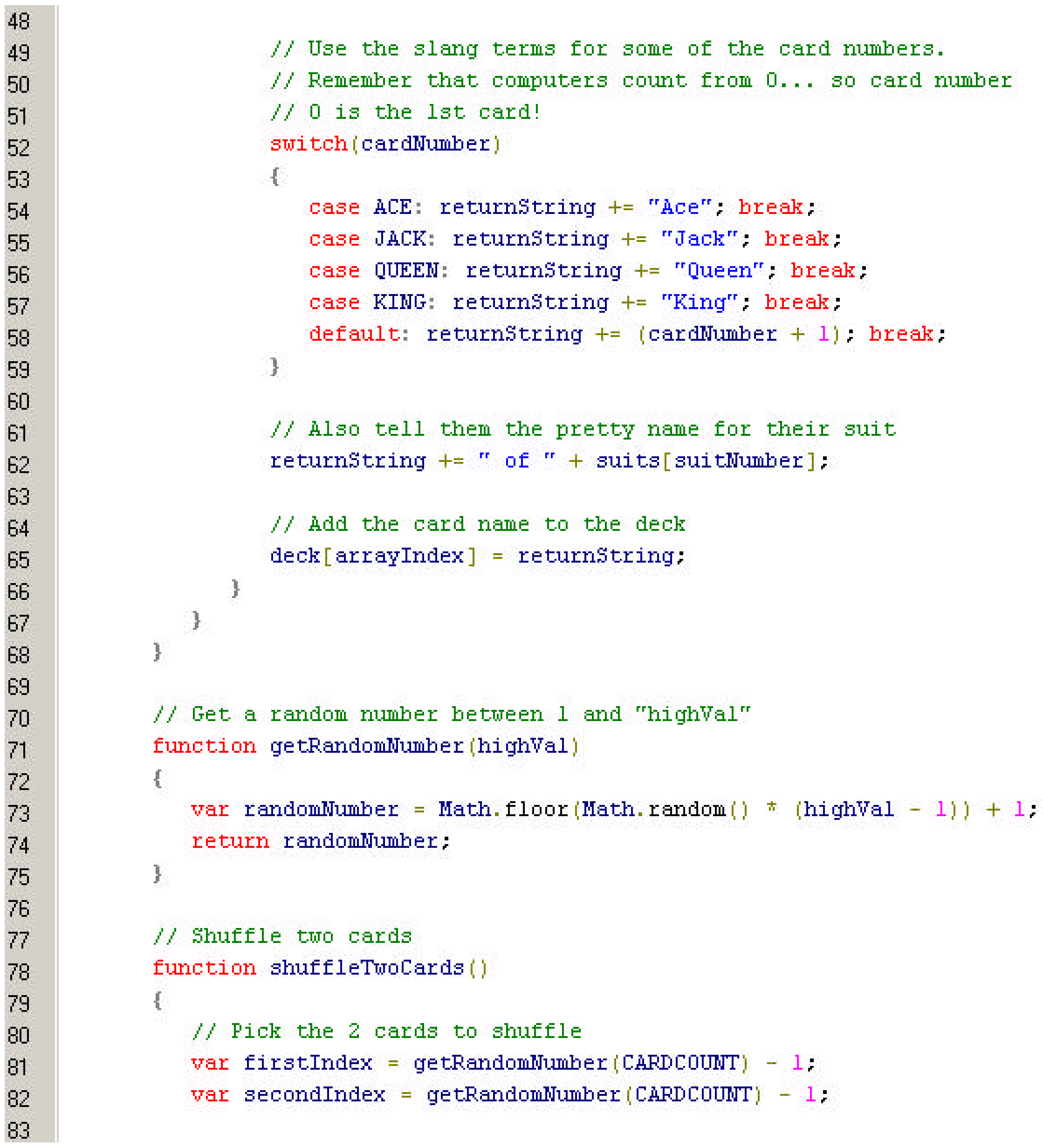
A Large Example: Shuffling Cards

To finish this introduction to computer programming, let's review a rather large piece of Javascript that shuffles and displays a deck of cards. I've written this program primarily to illustrate the concepts that we just covered. However, this "tutorial code" is by no-means the best code: I'll point out items that I would have written differently if it were not a tutorial.

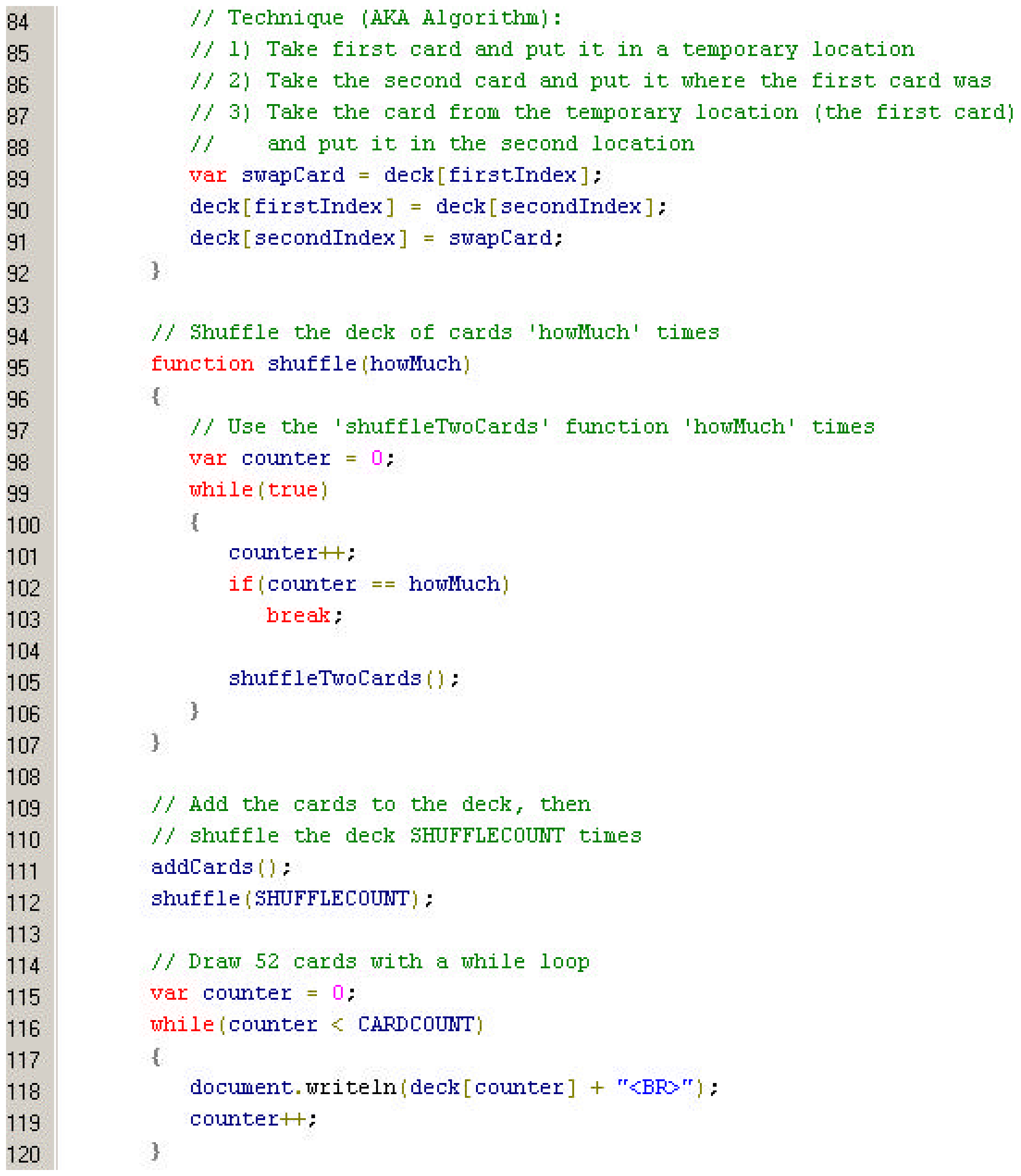
I'll also mention some programming points that have nothing to do with syntax. These will help you with programming in general, not simply programming in Javascript.



```
// Constants for our deck
var ACE = 0, JACK = 10, QUEEN = 11, KING = 12;
var NUMBEROFSUITS = 4;
var CARDSPERSUIT = 13;
var CARDCOUNT = (NUMBEROFSUITS * CARDSPERSUIT);

// How many times do we want to shuffle?
var SHUFFLECOUNT = 1000;

// The names of the suits
var HEARTS = 0, SPADES = 1; DIAMONDS = 2; CLUBS = 3;
var suits = new Array("Hearts", "Spades", "Diamonds", "Clubs");

// Store all the cards
var deck = new Array(CARDCOUNT);

// Add cards to the deck
function addCards()
{
   // Go through all the suits (slow changing loop)
   for(var suitNumber = 0; suitNumber < 4; suitNumber++)
   {
      // Go through all the cards (fast changing loop)
      for(var cardNumber = 0; cardNumber < 13; cardNumber++)
      {
         // Convert the combination of "suitNumber" and "cardNumber"
         // to a number between 0 and 51.
         var arrayIndex = (suitNumber * CARDSPERSUIT) + cardNumber;

         // The name of the card
         var returnString = "";

         // Check if it's a one-eyed card
         if(((suitNumber == DIAMONDS) && (cardNumber == KING)) ||
            ((suitNumber == HEARTS) && (cardNumber == JACK)) ||
            ((suitNumber == SPADES) && (cardNumber == JACK)))
            returnString += " <i>One-eyed</i> ";
```



```
         // Use the slang terms for some of the card numbers.
         // Remember that computers count from 0... so card number
         // 0 is the 1st card!
         switch(cardNumber)
         {
            case ACE: returnString += "Ace"; break;
            case JACK: returnString += "Jack"; break;
            case QUEEN: returnString += "Queen"; break;
            case KING: returnString += "King"; break;
            default: returnString += (cardNumber + 1); break;
         }

         // Also tell them the pretty name for their suit
         returnString += " of " + suits[suitNumber];

         // Add the card name to the deck
         deck[arrayIndex] = returnString;
      }
   }
}

// Get a random number between 1 and "highVal"
function getRandomNumber(highVal)
{
   var randomNumber = Math.floor(Math.random() * (highVal - 1)) + 1;
   return randomNumber;
}

// Shuffle two cards
function shuffleTwoCards()
{
   // Pick the 2 cards to shuffle
   var firstIndex = getRandomNumber(CARDCOUNT) - 1;
   var secondIndex = getRandomNumber(CARDCOUNT) - 1;

```



```
   // Technique (AKA Algorithm):
   // 1) Take first card and put it in a temporary location
   // 2) Take the second card and put it where the first card was
   // 3) Take the card from the temporary location (the first card)
   //    and put it in the second location
   var swapCard = deck[firstIndex];
   deck[firstIndex] = deck[secondIndex];
   deck[secondIndex] = swapCard;
}

// Shuffle the deck of cards 'howMuch' times
function shuffle(howMuch)
{
   // Use the 'shuffleTwoCards' function 'howMuch' times
   var counter = 0;
   while(true)
   {
      counter++;
      if(counter == howMuch)
         break;

      shuffleTwoCards();
   }
}

// Add the cards to the deck, then
// shuffle the deck SHUFFLECOUNT times
addCards();
shuffle(SHUFFLECOUNT);

// Draw 52 cards with a while loop
var counter = 0;
while(counter < CARDCOUNT)
{
   document.writeln(deck[counter] + "<BR>");
   counter++;
}
```

Code Summary

We can break this card-shuffling program into three parts: *adding cards to the deck*, *shuffling the cards*, then finally *displaying the cards*. As you'll soon see, it's no surprise that I've written functions to handle the first two operations. These functions parcel large amounts of code into easily understandable operations.

Using functions properly

This is one tenet of good computer programming. Like any other endeavour, you need to learn how to delegate responsibility.

If you want to write a program to shuffle cards and display them, you might at first feel overwhelmed by the size of the task. Being one of keen management whit, you can delegate your responsibility. Maybe you'll get one friend to write the part that creates the deck of cards. You might also get another friend to write the part that shuffles the cards. While you're at it, you might as well get another friend to write the part that displays the cards.

If your friend asked you to write a bit of code to *create* a deck of cards, you could just run off and start coding. After all, it's about a single page of programming.

If your friend asked you to write a bit of code to *display* a shuffled a deck of cards, you could just run off and start coding. After all, it's much less than a single page of programming.

If your friend asked you to write a bit of code to *shuffle* a deck of cards, you might at first feel overwhelmed by the size of the task. After all, how do you shuffle an entire deck of cards? Being one of keen management whit, you could delegate your responsibility. Maybe you'd get a friend to write a part that switches two (random) cards and just use that part many times.

If your friend asked you to write a bit of code to *switch* two cards in a deck, you could just run off and start coding. After all, it's much less than a single page of programming.

Do you see any patterns in this logic? Of course! If a programming task contains more than one about one page of logic, you're probably trying to say too much at once. At that point, you should break your task into several independent sub-tasks. Of course, if your friends don't want to write the bits of code, you can always do it.

How does this help you in the long run? It helps in two respects: *maintenance*, and *readability.*

Maintenance
Let's say you've written your card-shuffle and display program, but want to tweak it. Perhaps you want to change the shuffling algorithm to comply with the "Las Vegas International Committee for Card Shuffling" guidelines. If you've written your program poorly, you might wade through pages of code before you find the part that shuffles your cards. In fact, since you wove it so intricately with the rest of your code, you also worry that you'll break the "add cards" bit, or the "display cards" bit.

However, if you've write your code properly, you can search for the "shuffle" function and restrict your changes to that function alone.

Readability
Reading (and understanding) a well-written computer program is much like reading a book. When you want a very high-level view of a book, you simply read the *first level* of the table of contents. For example, a quick scan of this document's table of contents tells you that we're learning about "The Tools You'll Need," "Your First Program," and so on. Likewise, we know that our Card Shuffling program executes the functions "addCards()," "shuffle()", then displays the cards.

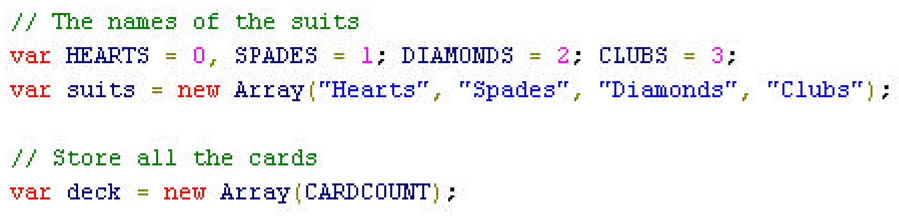
After this glance at our book, we might want to learn more about "The Tools You'll Need." By scanning the table of contents one level deeper, we can see that we'll need "A Text Editor," "A Web Browser," and a few more items. Likewise, we know that our "shuffle()" function in the Card Shuffling program executes the function "shuffleTwoCards()" many times. Without reading any code that actually *does* anything, we already have a good idea of what our program does.

We can continue to break the structure of a book or computer program into finer details until we finally arrive at sentences or lines of code.

Program variables

Lines 12-25 create several variables for our program to use.

Constants
Lines 12-21 define constants for our program to use. By glancing through the code, it is easy to see how our constants improve the program's readability. One sticky point is that the constants on line 12 define some of the special cards with a *zero-based index*. That is, the first card (ACE) would be card #0 in a list. Likewise, the 12th card (QUEEN) would be card #11 in a list.



```
// The names of the suits
var HEARTS = 0, SPADES = 1; DIAMONDS = 2; CLUBS = 3;
var suits = new Array("Hearts", "Spades", "Diamonds", "Clubs");

// Store all the cards
var deck = new Array(CARDCOUNT);
```

Working variables
Lines 22-25 create variables that we actively work with throughout our program. Notice that we create two arrays: one that we populate immediately, and one that we populate soon after.

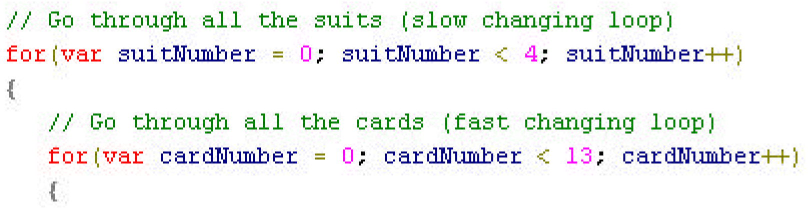
Lines 21 and 22 give us a convenient way to find out the name of a suit from its suit number. We arbitrarily define the suit numbers on line 21. Computer programmers call this a *look-up table* since you *look up* the name of a suit once you know its number. As you will soon see, the expression "suits[HEARTS]" is much easier to read than a large *case statement* to return a suit name from a suit number.

Function addCards()

As its comments describe, lines 27-68 define a function that adds cards to our program's deck. This is the deck that we defined in line 25.

Nested for-loops

At the highest level, this function loops through each of the 4 suits. Nested in this "outer loop" is an "inner loop" that goes through each of the 13 cards of that suit. As the comments mention, the outer loop runs much slower than the inner loop – because the inner loop happens 13 times for each time that the outer loop gets executed. Notice your added bonus of bad code: the comparison in each loop uses a **magic number**! We should replace the numbers '4' and '13' in the loops with the NUMBEROFSUITS and CARDSPERSUIT constants to make this code easier to understand.



```
// Go through all the suits (slow changing loop)
for(var suitNumber = 0; suitNumber < 4; suitNumber++)
{
    // Go through all the cards (fast changing loop)
    for(var cardNumber = 0; cardNumber < 13; cardNumber++)
    {
```

Flattened arrays

The next part of our function, line 38, uses a technique called *flattened array indexing*. Even the term is complicated, so let's explain the concept.

Pretend you have a friend who lives in a 5-story apartment building. Each story has 6 apartments. If the landlord wants to give an apartment number to each unit – with no gaps or doubles, he or she might use the following system:

1) Number the apartments on the bottom floor (floor number 0 in computer speak) from 0 to 5.
2) Number the apartments on the next floor (floor number 1 in computer speak) from 6 to 11
3) …

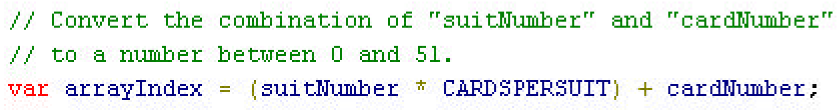
In fact, he or she might make their life a little easier with a little math. To find out the apartment number of any unit on any floor, simply:

1) Count the number of units in the floors below. To work this out, multiply the number of floors below the unit by the number of units per floor. Notice that the number of floors below the 4th floor (3) is the same as the 0-based number of the 4th floor (3).
2) Add the unit's hallway-position to that number. Ie: add 0 for the 0th unit, and 4 for the 5th unit.

So the third unit on the 4th floor would be:

= (3 * UNITSPERFLOOR) + 2
= (3 * 6) + 2
= #20.

How does this apply to cards? Well, rather than use their proper names, we can number the suits of our deck between 0 and 3. Next, we can number the cards of each suit between 0 and 12. Now that we know the 0-based index of the suit (floor) and 0-based index of the card (hallway-position,) we can give each card a unique number between 0 and 51. This fits very nicely into a single 52-element array with the formula: ("suit number" * "cards per suit") + "card number".



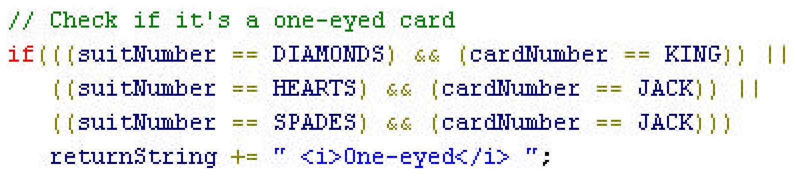
```
// Convert the combination of "suitNumber" and "cardNumber"
// to a number between 0 and 51.
var arrayIndex = (suitNumber * CARDSPERSUIT) + cardNumber;
```

Why do we do it this way? The reason is that a little complexity now saves us a lot of complexity later. First, we can print a single list of 52 cards easier than 4 lists of 13 cards. After all, the "4 suits of 13" concept relates very little to a shuffled deck of cards. Second, we can shuffle a single list of 52 cards much easier than 4 lists of 13 cards. We can switch two entries of a single array much easier than we can switch two entries out of four arrays.

So given the suit number (outer loop) and card number (inner loop), line 38 computes the index into the array that has room for 52 cards.

String preparation
Lines 40-41 prepare a variable to eventually hold a string like "*One-eyed* Jack of Hearts." At this point, the string is empty.

Conditional Logic
Lines 43-47 check if the card is a one-eyed card or not. It is a very typical example of conditional statements inside an if-statement.



```
// Check if it's a one-eyed card
if(((suitNumber == DIAMONDS) && (cardNumber == KING)) ||
   ((suitNumber == HEARTS) && (cardNumber == JACK)) ||
   ((suitNumber == SPADES) && (cardNumber == JACK)))
   returnString += " <i>One-eyed</i> ";
```

When we convert this statement into English, we read it something like: "If the card is a 'King of Diamonds' OR 'Jack of Hearts' OR 'Jack of Spades' then add "One-eyed" to the string we're preparing. Notice the HTML code <i> and </i> that puts our words in italics.

Case statement
Lines 49-59 illustrate a typical case-statement. In it, we map the named card-numbers (Ace, Jack, Queen, King) as defined earlier into their English equivalents. If there is a match, we add this slang name to the string we're preparing. Otherwise, we simply add the card number (plus one! Don't forget that our *zero*-based card number 1 is really the *second card*.)

Suit name
As I mentioned earlier, lines 64-65 use a lookup-table to convert a suit number to a proper name.

Add the card

```
// Add the card name to the deck
deck[arrayIndex] = returnString;
```

To complete this function, we place the full name of the card (ie: "2 of Hearts") into the part of the deck that we computed earlier.

Function getRandomNumber()

Lines 70-75 define a function to return a random number between 1 and the parameter you pass to the function. I've simply wrapped our dice-rolling random number generator neatly into a function.

Function shuffleTwoCards()

Lines 77-92 describe a function that swaps two cards in our deck. The temporary-value technique that this function uses is the most common (and efficient) way to swap two variables. Note that our program chooses the *firstIndex* and *secondIndex* with the random number function that we wrote.

```
var swapCard = deck[firstIndex];
deck[firstIndex] = deck[secondIndex];
deck[secondIndex] = swapCard;
```

Function shuffle()

Lines 94-107 shuffle the deck. As you might notice, it does nothing more than shuffleTwoCards() many times. This is a perfect example of good functional programming.

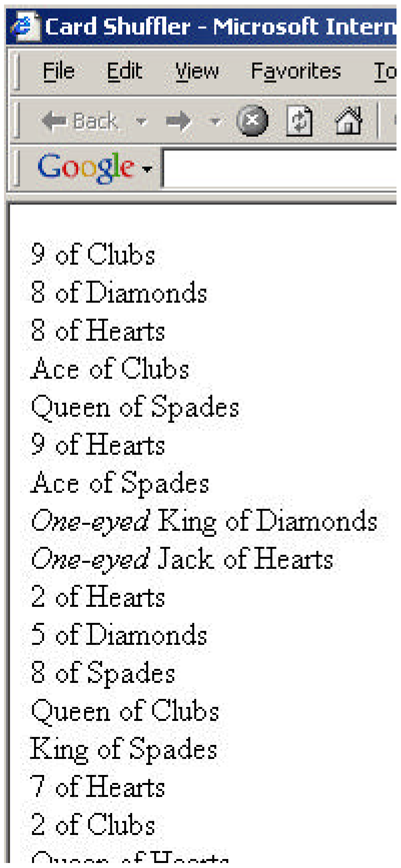
You might notice that this function uses a while loop and break statement to control a counted loop. If it weren't for tutorial purposes, I should have written this as the for loop:

```
for(var counter = 0; counter < howMuch; counter++)
  shuffleTwoCards();
```

The remainder

Lines 109-120 actually make the high-level structure of our program. First, lines 111 and 112 delegate the responsibility of creating and shuffling the deck. Finally, we cycle through the deck and print each card name on its own line. The "<BR>" is our way to tell Javascript to start a new line. Again, notice the while loop that we've cobbled together to simulate a for-loop. As you may guess, I should have written this as a for-loop.

The results



```
9 of Clubs
8 of Diamonds
8 of Hearts
Ace of Clubs
Queen of Spades
9 of Hearts
Ace of Spades
One-eyed King of Diamonds
One-eyed Jack of Hearts
2 of Hearts
5 of Diamonds
8 of Spades
Queen of Clubs
King of Spades
7 of Hearts
2 of Clubs
```

Your Programming Future

As I mentioned near the beginning of this piece, I've aimed to *introduce* you to computer programming – not teach you its entirety. Do not let this statement dissuade you, though. The trait that unites all good computer programmers is their *programming skill*. You'll need to learn new syntax rules as you move between programming languages, but the core programming principles rarely change. In fact, you can pick up any programming language with the fundamentals that this tutorial has taught you.

If you decide to pursue computer programming, you'll need to decide on a language and learn it well. Once you think you've learned a language, you'll have to practice it well. Start with a well-written book that has a good index. There are many propellerheads that have written 1000 page programming books. Ignore that unfortunate majority and look first for a book that reads well. Next, come up with some sample questions ("What are the parameters to the document.writeln() method?") and see how quickly you can locate their answers. Once you've learned a language or two, you'll find that you can easily use internet searches and tutorials to teach yourself a new language.

Javascript is an excellent language to add functionality to your website. In fact, Javascript is a major component of a new trend in web-development called *Dynamic HTML*, or DHTML for short. Aside from web development, though, Javascript offers very little power.

To create more powerful programs for your web site, ask your hosting company (ie: school administrator, internet service provider) if they support the "Perl," "PHP," or "Python" languages. Any of these are powerful enough to support almost any application you might write. However, they are *interpreted scripting languages* and carry with them the burden of that title.

To create programs that are fast, powerful, and easy to distribute, I would highly recommend that you explore compiled languages like Java, C#, or C++. These languages offer tremendous amounts of pre-written code in standard libraries, software development kits (SDKs,) and application programming interfaces (APIs.) Nearly every professional software developer has this type of language as his or her "native tongue."

Whatever language you choose, though, make sure that you use its object-oriented capabilities to the fullest. Object-oriented programming has literally transformed the software development industry – and for good reason.

I wish you luck and perseverance.