Object-Oriented JavaScript
Create scalable, reusable high-quality JavaScript applications, and libraries

Stoyan Stefanov

Chapter No. 2
"Primitive Data Types, Arrays, Loops, and Conditions"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Primitive Data Types, Arrays, Loops, and Conditions"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Stoyan Stefanov is a Yahoo! web developer, Zend Certified Engineer, and book author. He talks regularly about JavaScript, PHP, and other web development topics at conferences and on his blog at www.phpied.com, and also runs a number of other sites, including JSPatterns.com—a site dedicated to exploring JavaScript patterns. Stoyan is the engineering lead of Yahoo!’s performance optimization tool YSlow, and contributes to other open-source projects such as Firebug and PEAR.

A "citizen of the world", Stoyan was born and raised in Bulgaria, but is also a Canadian citizen, and is currently residing in Los Angeles, California. In his rare offline moments, he enjoys playing the guitar and going to the Santa Monica beaches and pools with his family.

For More Information:
Object-Oriented JavaScript

Create scalable, reusable high-quality JavaScript applications, and libraries

This book explores JavaScript for what it is: a highly expressive and flexible prototype-based object-oriented programming language. Once dismissed as a toy for designers to make things such as rollover buttons, today this interesting and unique language is back, stronger than ever. Today's Web 2.0 world of AJAX, fat-client programming, desktop-like rich Internet applications, drag-and-drop maps and webmail clients, rely heavily on JavaScript to provide a highly interactive user experience. And if we never had the chance to properly explore JavaScript before, now is the time to sit down and (re-)learn it.

This book doesn't assume any prior knowledge of JavaScript and works from the ground up to give you a thorough understanding of the language.

What This Book Covers

Chapter 1 talks briefly about the history, present, and future of JavaScript, and then moves on to explore the basics of object-oriented programming (OOP) in general. You then learn how to set up your training environment (Firebug) in order to dive into the language on your own, using the book examples as a base.

Chapter 2 discusses the language basics: variables, data types, arrays, loops, and conditionals.

Chapter 3 covers functions. JavaScript has many uses for functions and here you learn to master them all. You also learn about the scope of variables and JavaScript's built-in functions. An interesting, but often misunderstood feature of the language—closures—is demystified at the end of the chapter.

Chapter 4 introduces objects: how to work with properties and methods, and the various ways to create your objects. There's also an overview of the built-in objects in JavaScript, such as Math and Date (just an overview, Appendix C has all the details).

Chapter 5 is dedicated to the all-important concept of prototypes in JavaScript.

Chapter 6 expands your "thinking in JavaScript" horizon, discussing a dozen ways to implement inheritance in JavaScript.

For More Information:

Chapter 7 is the browser chapter. In this chapter, you learn about BOM (Browser Object Model), DOM (W3C's Document Object Model), browser events, and AJAX.

Chapter 8 dives into various unique JavaScript coding patterns, as well as several language-independent design patterns, translated to JavaScript from the Book of Four, the most influential work of software design patterns.

Appendix A lists the reserved words in JavaScript.

Appendix B is a reference to the built-in JavaScript functions, together with sample uses.

Appendix C is a reference that provides detail and provides examples of the use of every method and property of every built-in object in JavaScript.

Appendix D is a regular expressions pattern reference.
Primitive Data Types, Arrays, Loops, and Conditions

Before diving into the object-oriented features of JavaScript, let's first take a look at some of the basics. This chapter walks you through:

- The primitive data types in JavaScript, such as strings and numbers
- Arrays
- Common operators, such as +, -, delete, and typeof
- Flow control statements, such as loops and if-else conditions

Variables

Variables are used to store data. When writing programs, it is convenient to use variables instead of the actual data, as it’s much easier to write \( \pi \) instead of 3.141592653589793 especially when it happens several times inside your program. The data stored in a variable can be changed after it was initially assigned, hence the name “variable”. Variables are also useful for storing data that is unknown to the programmer when the code is written, such as the result of later operations.

There are two steps required in order to use a variable. You need to:

- Declare the variable
- Initialize it, that is, give it a value

In order to declare a variable, you use the `var` statement, like this:

```javascript
var a;
var thisIsAVariable;
var _and_this_too;
var mix12three;
```
For the names of the variables, you can use any combination of letters, numbers, and the underscore character. However, you can't start with a number, which means that this is invalid:

```javascript
var 2three4five;
```

To initialize a variable means to give it a value for the first (initial) time. You have two ways to do so:

- Declare the variable first, then initialize it, or
- Declare and initialize with a single statement

An example of the latter is:

```javascript
var a = 1;
```

Now the variable named `a` contains the value `1`.

You can declare (and optionally initialize) several variables with a single `var` statement; just separate the declarations with a comma:

```javascript
var v1, v2, v3 = 'hello', v4 = 4, v5;
```

### Variables are Case Sensitive

Variable names are case-sensitive. You can verify this statement using the Firebug console. Try typing this, pressing `Enter` after each line:

```javascript
var case_matters = 'lower';
var CASE_MATTERS = 'upper';
console.log(case_matters, CASE_MATTERS)
```

To save keystrokes, when you enter the third line, you can only type `ca` and press the `Tab` key. The console will auto-complete the variable name to `case_matters`. Similarly, for the last line—type `CA` and press `Tab`. The end result is shown on the following figure.
Throughout the rest of this book, only the code for the examples will be given, instead of a screenshot:

```javascript
>>> var case_matters = 'lower';
>>> var CASE_MATTERS = 'upper';
>>> case_matters
"lower"
>>> CASE_MATTERS
"upper"
```

The three consecutive greater-than signs (>>>) show the code that you type, the rest is the result, as printed in the console. Again, remember that when you see such code examples, you're strongly encouraged to type in the code yourself and experiment tweaking it a little here and there, so that you get a better feeling of how it works exactly.

**Operators**

Operators take one or two values (or variables), perform an operation, and return a value. Let's check out a simple example of using an operator, just to clarify the terminology.

```javascript
>>> 1 + 2
3
```
In this code:

- + is the *operator*
- The *operation* is addition
- The input values are 1 and 2 (the input values are also called *operands*)
- The *result* value is 3

Instead of using the values 1 and 2 directly in the operation, you can use variables. You can also use a variable to store the result of the operation, as the following example demonstrates:

```javascript
>>> var a = 1;
>>> var b = 2;
>>> a + 1
2
>>> b + 2
4
>>> a + b
3
>>> var c = a + b;
>>> c
3
```

The following table lists the basic arithmetic operators:

<table>
<thead>
<tr>
<th>Operator symbol</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>&gt;&gt;&gt; 1 + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>&gt;&gt;&gt; 99.99 - 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>88.99</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>&gt;&gt;&gt; 2 * 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>&gt;&gt;&gt; 6 / 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5</td>
</tr>
</tbody>
</table>
## Chapter 2

<table>
<thead>
<tr>
<th>Operator symbol</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Modulo, the remainder of a division</td>
<td>&gt;&gt;&gt; 6 % 3 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; 5 % 3 2</td>
</tr>
<tr>
<td></td>
<td>It's sometimes useful to test if a number is even or odd. Using the modulo operator it's easy. All odd numbers will return 1 when divided by 2, while all even numbers will return 0.</td>
<td>&gt;&gt;&gt; 4 % 2 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; 5 % 2 1</td>
</tr>
<tr>
<td>++</td>
<td>Increment a value by 1</td>
<td>Post-increment is when the input value is incremented after it's returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; var a = 123; var b = a++; &gt;&gt;&gt; b 123 &gt;&gt;&gt; a 124</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The opposite is pre-increment; the input value is first incremented by 1 and then returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; var a = 123; var b = ++a; &gt;&gt;&gt; b 124 &gt;&gt;&gt; a 124</td>
</tr>
<tr>
<td>--</td>
<td>Decrement a value by 1</td>
<td>Post-decrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; var a = 123; var b = a--; &gt;&gt;&gt; b 123 &gt;&gt;&gt; a 122</td>
</tr>
<tr>
<td></td>
<td>Pre-decrement</td>
<td>&gt;&gt;&gt; var a = 123; var b = --a; &gt;&gt;&gt; b 122 &gt;&gt;&gt; a 122</td>
</tr>
</tbody>
</table>

For More Information:  
When you type `var a = 1;` this is also an operation; it's the simple assignment operation and `=` is the simple assignment operator.

There is also a family of operators that are a combination of an assignment and an arithmetic operator. These are called compound operators. They can make your code more compact. Let's see some of them with examples.

```plaintext
>>> var a = 5;
>>> a += 3;
8

In this example `a += 3;` is just a shorter way of doing `a = a + 3;`

```plaintext
>>> a -= 3;
5
```
Here `a -= 3;` is the same as `a = a - 3;`

Similarly:

```plaintext
>>> a *= 2;
10
```%

```plaintext
>>> a /= 5;
2
```%

```plaintext
>>> a %= 2;
0
```

In addition to the arithmetic and assignment operators discussed above, there are other types of operators, as you'll see later in this and the following chapters.

**Primitive Data Types**

Any value that you use is of a certain type. In JavaScript, there are the following primitive data types:

1. Number—this includes floating point numbers as well as integers, for example 1, 100, 3.14.
2. String—any number of characters, for example "a", "one", "one 2 three".
3. Boolean—can be either true or false.
4. Undefined—when you try to access a variable that doesn't exist, you get the special value undefined. The same will happen when you have declared a variable, but not given it a value yet. JavaScript will initialize it behind the scenes, with the value undefined.

5. Null—this is another special data type that can have only one value, the null value. It means no value, an empty value, nothing. The difference with undefined is that if a variable has a value null, it is still defined, it only happens that its value is nothing. You'll see some examples shortly.

Any value that doesn't belong to one of the five primitive types listed above is an object. Even null is considered an object, which is a little awkward—having an object (something) that is actually nothing. We'll dive into objects in Chapter 4, but for the time being just remember that in JavaScript the data types are either:

- Primitive (the five types listed above), or
- Non-primitive (objects)

**Finding out the Value Type — the typeof Operator**

If you want to know the data type of a variable or a value, you can use the special typeof operator. This operator returns a string that represents the data type. The return values of using typeof can be one of the following—"number", "string", "boolean", "undefined", "object", or "function". In the next few sections, you'll see typeof in action using examples of each of the five primitive data types.

**Numbers**

The simplest number is an integer. If you assign 1 to a variable and then use the typeof operator, it will return the string "number". In the following example you can also see that the second time we set a variable's value, we don't need the var statement.

```javascript
>>> var n = 1;
>>> typeof n;
"number"

>>> n = 1234;
>>> typeof n;
"number"
```
Numbers can also be floating point (decimals):

```javascript
>>> var n2 = 1.23;
>>> typeof n;
"number"
```

You can call `typeof` directly on the value, without assigning it to a variable first:

```javascript
>>> typeof 123;
"number"
```

### Octal and Hexadecimal Numbers

When a number starts with a 0, it's considered an octal number. For example, the octal 0377 is the decimal 255.

```javascript
>>> var n3 = 0377;
>>> typeof n3;
"number"
>>> n3;
255
```

The last line in the example above prints the decimal representation of the octal value. While you may not be very familiar with octal numbers, you've probably used hexadecimal values to define, for example, colors in CSS stylesheets.

In CSS, you have several options to define a color, two of them being:

- Using decimal values to specify the amount of R (red), G (green) and B (blue) ranging from 0 to 255. For example `rgb(0, 0, 0)` is black and `rgb(255, 0, 0)` is red (maximum amount of red and no green or blue).
- Using hexadecimals, specifying two characters for each R, G and B. For example, `#000000` is black and `#ff0000` is red. This is because `ff` is the hexadecimal for 255.

In JavaScript, you put `0x` before a hexadecimal value (also called `hex` for short).

```javascript
>>> var n4 = 0x00;
>>> typeof n4;
"number"
>>> n4;
0
```
>>> var n5 = 0xff;
>>> typeof n5;

"number"

>>> n5;
255

Exponent Literals

1e1 (can also be written as 1e+1 or 1E1 or 1E+1) represents the number one with one zero after it, or in other words 10. Similarly, 2e+3 means the number 2 with 3 zeros after it, or 2000.

>>> 1e1
10

>>> 1e+1
10

>>> 2e+3
2000

>>> typeof 2e+3;
"number"

2e+3 means moving the decimal point 3 digits to the right of the number 2. There's also 2e-3 meaning you move the decimal point 3 digits to the left of the number 2.

\[
\begin{array}{c|c|c|c}
2e+3 & 2 & . & 0 & . & 0 & . & 0 & . & 2000 \\
1 & 2 & 3 \\

2e-3 & 0 & . & 0 & . & 0 & . & 0 & . & 002 \\
3 & 2 & 1 \\
\end{array}
\]

>>> 2e-3
0.002

>>> 123.456E-3
0.123456
Primitive Data Types, Arrays, Loops, and Conditions

```javascript
>>> typeof 2e-3
"number"
```

### Infinity

There is a special value in JavaScript called `Infinity`. It represents a number too big for JavaScript to handle. `Infinity` is indeed a number, as typing `typeof Infinity` in the console will confirm. You can also quickly check that a number with 308 zeros is ok, but 309 zeros is too much. To be precise, the biggest number JavaScript can handle is `1.7976931348623157e+308` while the smallest is `5e-324`.

```javascript
>>> Infinity
Infinity
>>> typeof Infinity
"number"
```

```javascript
>>> 1e309
Infinity
```

```javascript
>>> 1e308
1e+308
```

Dividing by 0 will give you infinity.

```javascript
>>> var a = 6 / 0;
>>> a
Infinity
```

Infinity is the biggest number (or rather a little bigger than the biggest), but how about the smallest? It's infinity with a minus sign in front of it, minus infinity.

```javascript
>>> var i = -Infinity;
>>> i
-Infinity
```

```javascript
>>> typeof i
"number"
```
Does this mean you can have something that's exactly twice as big as \textit{Infinity}—from 0 up to infinity and then from 0 down to minus infinity? Well, this is purely for amusement and there's no practical value to it. When you sum infinity and minus infinity, you don't get 0, but something that is called \textbf{NaN} (Not A Number).

>>> Infinity - Infinity

\textbf{NaN}

>>> -Infinity + Infinity

\textbf{NaN}

Any other arithmetic operation with \textit{Infinity} as one of the operands will give you \textit{Infinity}:

>>> Infinity - 20

\textit{Infinity}

>>> -Infinity * 3

\textit{-Infinity}

>>> Infinity / 2

\textit{Infinity}

>>> Infinity - 99999999999999999

\textit{Infinity}

\textbf{NaN}

What was this \textbf{NaN} you saw in the example above? It turns out that despite its name, "Not A Number", \textbf{NaN} is a special value that is also a number.

>>> typeof NaN

"number"

>>> var a = NaN;

>>> a

\textbf{NaN}

You get \textbf{NaN} when you try to perform an operation that assumes numbers but the operation fails. For example, if you try to multiply 10 by the character "f", the result is \textbf{NaN}, because "f" is obviously not a valid operand for a multiplication.
>> var a = 10 * "f";
>>> a
NaN

**NaN** is contagious, so if you have even only one **NaN** in your arithmetic operation, the whole result goes down the drain.

```javascript
>>> 1 + 2 + NaN
NaN
```

### Strings

A **string** is a sequence of characters used to represent text. In JavaScript, any value placed between single or double quotes is considered a string. This means that 1 is a number but "1" is a string. When used on strings, `typeof` returns the string "string".

```javascript
>>> var s = "some characters";
>>> typeof s;
"string"

>>> var s = 'some characters and numbers 123 5.87';
>>> typeof s;
"string"
```

Here's an example of a number used in string context:

```javascript
>>> var s = '1';
>>> typeof s;
"string"
```

If you put nothing in quotes, it's still a string (an empty string):

```javascript
>>> var s = ""; typeof s;
"string"
```

As you saw before, when you use the plus sign with two numbers, this is the arithmetic operation addition. However, if you use the plus sign on strings, this is a **string concatenation** operation and it returns the two strings glued together.

```javascript
>>> var s1 = "one"; var s2 = "two"; var s = s1 + s2; s;
"onetwo"

>>> typeof s;
"string"
```
The dual function of the + operator can be a source of errors. Therefore, it is always best to make sure that all of the operands are strings if you intend to concatenate them, and are all numbers if you intend to add them. You will learn various ways to do so further in the chapter and the book.

**String Conversions**

When you use a number-like string as an operand in an arithmetic operation, the string is converted to a number behind the scenes. (This works for all operations except addition, because of addition's ambiguity)

```javascript
>>> var s = '1'; s = 3 * s; typeof s;
"number"

>>> s
3

>>> var s = '1'; s++; typeof s;
"number"

>>> s
2
```

A lazy way to convert any number-like string to a number is to multiply it by 1 (a better way is to use a function called `parseInt()`, as you'll see in the next chapter):

```javascript
>>> var s = "100"; typeof s;
"string"

>>> s = s * 1;
100

>>> typeof s;
"number"
```

If the conversion fails, you'll get `NaN`:

```javascript
>>> var d = '101 dalmatians';
>>> d * 1
NaN
```
A lazy way to convert anything to a string is to concatenate it with an empty string.

```javascript
>>> var n = 1;
>>> typeof n;
"number"
>>> n = "" + n;
"1"
>>> typeof n;
"string"
```

## Special Strings

Some strings that have a special meaning, as listed in the following table:

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
</table>
| \
   \| is the escape character.                                             | "I don't know"; |
| \
   \| When you want to have quotes inside your string, you escape them, so that you mean the end of the string. | "I don" and the rest is invalid code. The following are valid: |
| "    If you want to have an actual backslash in the string, escape it with another backslash. | "Hello", he said.; |
| 
   \| End of line                                                            | "Hello", he said.; |

```javascript
>>> var s = 'I don\'t know';
>>> var s = "I don\'t know";
>>> var s = "I don't know";
>>> var s = "Hello", he said.';
>>> var s = "\"Hello\", he said.";

Escaping the escape:

```javascript
>>> var s = "1\2"; s;
"1\2"
```
### String Meaning Example

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r</td>
<td>Carriage return</td>
<td>All these:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; var s = '1\r2';</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; var s = '1\n\r2';</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; var s = '1\r\n2';</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result in:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2&quot;</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
<td>&gt;&gt;&gt; var s = &quot;1\t2&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;1 2&quot;</td>
</tr>
<tr>
<td>\u</td>
<td>\u followed by a character code allows you to use Unicode</td>
<td>Here’s my name in Bulgarian written with Cyrillic characters:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;&gt;&gt; &quot;\u0421\u0442\u043E\u044F\u043D&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Стойн&quot;</td>
</tr>
</tbody>
</table>

There are some additional characters which are rarely used: \b (backspace), \v (vertical tab), and \f (form feed).

### Booleans

There are only two values that belong to the boolean data type: the values true and false, used without quotes.

```javascript
>>> var b = true; typeof b;
"boolean"

>>> var b = false; typeof b;
"boolean"
```

If you quote true or false, they become strings.

```javascript
>>> var b = "true"; typeof b;
"string"
```
Logical Operators
There are three operators, called logical operators, that work with boolean values. These are:

- `!`—logical NOT (negation)
- `&&`—logical AND
- `||`—logical OR

In everyday meaning, if something is not true, it is false. Here's the same statement expressed using JavaScript and the logical `!` operator.

```javascript
>>> var b = !true;
>>> b;
false
```

If you use the logical NOT twice, you get the original value:

```javascript
>>> var b = !!true;
>>> b;
true
```

If you use a logical operator on a non-boolean value, the value is converted to boolean behind the scenes.

```javascript
>>> var b = "one";
>>> !b;
false
```

In the case above, the string value "one" was converted to a boolean `true` and then negated. The result of negating `true` is `false`. In the next example, we negate twice so the result is `true`.

```javascript
>>> var b = "one";
>>> !!b;
true
```

Using double negation is an easy way to convert any value to its boolean equivalent. This is rarely useful, but on the other hand understanding how any value converts to a boolean is important. Most values convert to `true` with the exception of the following (which convert to `false`):

- The empty string ""
- `null`
undefined
The number 0
The number NaN
The boolean false

These six values are sometimes referred to as being falsy, while all others are truthy (including, for example, the strings "0", ",", and "false").

Let's see some examples of the other two operators—the logical AND and the logical OR. When you use AND, the result is true only if all of the operands are true. When using OR, the result is true if at least one of the operands is true.

```javascript
>>> var b1 = true; var b2 = false;
>>> b1 || b2
true
```

```javascript
>>> b1 && b2
false
```

Here's a table that lists the possible operations and their results:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true &amp;&amp; true</td>
<td>true</td>
</tr>
<tr>
<td>true &amp;&amp; false</td>
<td>false</td>
</tr>
<tr>
<td>false &amp;&amp; true</td>
<td>false</td>
</tr>
<tr>
<td>false &amp;&amp; false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

You can use several logical operations one after the other:

```javascript
>>> true && true && false && true
false
```

```javascript
>>> false || true || false
true
```

---

For More Information:
Primitive Data Types, Arrays, Loops, and Conditions

You can also mix && and || in the same expression. In this case, you should use parentheses to clarify how you intend the operation to work. Consider these:

```python
>>> false && false || true && true
true

>>> false && (false || true) && true
false
```

**Operator Precedence**

You might wonder why the expression above (false && false || true && true) returned true. The answer lies in operator precedence. As you know from mathematics:

```python
>>> 1 + 2 * 3
7
```

This is because multiplication has precedence over addition, so 2 * 3 is evaluated first, as if you've typed:

```python
>>> 1 + (2 * 3)
7
```

Similarly for logical operations, ! has the highest precedence and is executed first, assuming there are no parentheses that demand otherwise. Then, in the order of precedence, comes && and finally ||. In other words:

```python
>>> false && false || true && true
true
```

is the same as:

```python
>>> (false && false) || (true && true)
true
```

**Best Practice**

Use parentheses instead of relying on operator precedence. This makes your code easier to read and understand.
Lazy Evaluation

If you have several logical operations one after the other, but the result becomes clear at some point before the end, the final operations will not be performed, because they can't affect the end result. Consider this:

```javascript
>>> true || false || true || false || true
true
```

Since these are all OR operations and have the same precedence, the result will be `true` if at least one of the operands is `true`. After the first operand is evaluated, it becomes clear that the result will be `true`, no matter what values follow. So the JavaScript engine decides to be lazy (ok, efficient) and not do unnecessary work by evaluating code that doesn't affect the end result. You can verify this behavior by experimenting in the console:

```javascript
>>> var b = 5;
>>> true || (b = 6)
true
>>> b
5
```

```javascript
>>> true && (b = 6)
6
>>> b
6
```

This example also shows another interesting behavior—if JavaScript encounters a non-boolean expression as an operand in a logical operation, the non-boolean is returned as a result.

```javascript
>>> true || "something"
true
>>> true && "something"
"something"
```
This behavior is something to watch out for and avoid, because it makes the code harder to understand. Sometimes you might see this behavior being used to define variables when you’re not sure whether they were previously defined. In the next example, if the variable `v` is defined, its value is kept; otherwise, it’s initialized with the value 10.

```javascript
var mynumber = mynumber || 10;
```

This is simple and looks elegant, but be aware that it is not completely bulletproof. If `mynumber` is defined and initialized to 0 (or to any of the six falsy values), this code might not behave in exactly the way it was designed to work.

### Comparison

There’s another set of operators that all return a boolean value as a result of the operation. These are the comparison operators. The following table lists them, together with some examples.

<table>
<thead>
<tr>
<th>Operator symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>Equality comparison: Returns <code>true</code> when both operands are equal. The operands are converted to the same type before being compared.</td>
<td><code>1 == 1</code> <code>true</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1 == 2</code> <code>false</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1 == '1'</code> <code>true</code></td>
</tr>
<tr>
<td><code>===</code></td>
<td>Equality and type comparison: Returns <code>true</code> if both operands are equal and of the same type. It’s generally better and safer if you compare this way, because there’s no behind-the-scenes type conversions.</td>
<td><code>1 === '1'</code> <code>false</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1 === 1</code> <code>true</code></td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Non-equality comparison: Returns <code>true</code> if the operands are not equal to each other (after a type conversion)</td>
<td><code>1 != 1</code> <code>false</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1 != '1'</code> <code>false</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1 != '2'</code> <code>true</code></td>
</tr>
<tr>
<td><code>!==</code></td>
<td>Non-equality comparison without type conversion: Returns <code>true</code> if the operands are not equal OR they are different types.</td>
<td><code>1 !== 1</code> <code>false</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1 !== '1'</code> <code>true</code></td>
</tr>
</tbody>
</table>
### Chapter 2

<table>
<thead>
<tr>
<th>Operator symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Returns <code>true</code> if the left operand is greater than the right one.</td>
<td>&gt;&gt;&gt; 1 &gt; 1 false &gt;&gt;&gt; 33 &gt; 22 true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Returns <code>true</code> if the left operand is greater than or equal to the right one.</td>
<td>&gt;&gt;&gt; 1 &gt;= 1 true</td>
</tr>
<tr>
<td>&lt;</td>
<td>Returns <code>true</code> if the left operand is less than the right one.</td>
<td>&gt;&gt;&gt; 1 &lt; 1 false &gt;&gt;&gt; 1 &lt; 2 true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Returns <code>true</code> if the left operand is less than or equal to the right one.</td>
<td>&gt;&gt;&gt; 1 &lt;= 1 true &gt;&gt;&gt; 1 &lt;= 2 true</td>
</tr>
</tbody>
</table>

An interesting thing to note is that `NaN` is not equal to anything, not even itself.

```javascript
>>> NaN == NaN
false
```

**Undefined and null**

You get the `undefined` value when you try to use a variable that doesn't exist, or one that hasn't yet been assigned a value. When you declare a variable without initializing it, JavaScript automatically initializes it to the value `undefined`.

If you try using a non-existing variable, you'll get an error message.

```javascript
>>> foo
foo is not defined
```

If you use the `typeof` operator on a non-existing variable, you get the string "undefined".

```javascript
>>> typeof foo
"undefined"
```
Primitive Data Types, Arrays, Loops, and Conditions

If you declare a variable without giving it a value, you won’t get an error when you use that variable. But the typeof still returns "undefined".

```javascript
>>> var somevar;
>>> somevar
>>> typeof somevar
"undefined"
```

The null value, on the other hand, is not assigned by JavaScript behind the scenes; it can only be assigned by your code.

```javascript
>>> var somevar = null
null
>>> somevar
null
>>> typeof somevar
"object"
```

Although the difference between null and undefined is small, it may be important at times. For example, if you attempt an arithmetic operation, you can get different results:

```javascript
>>> var i = 1 + undefined; i;
NaN
>>> var i = 1 + null; i;
1
```

This is because of the different ways null and undefined are converted to the other primitive types. Below are examples that show the possible conversions.

Conversion to a number:

```javascript
>>> 1*undefined
NaN
>>> 1*null
0
```
Conversion to a boolean:

```javascript
>>> !!undefined
false

>>> !!null
false
```

Conversion to a string:

```javascript
>>> "" + null
"null"

>>> "" + undefined
"undefined"
```

**Primitive Data Types Recap**

Let's quickly summarize what has been discussed so far:

- There are five primitive data types in JavaScript:
  - number
  - string
  - boolean
  - undefined
  - null

- Everything that is not a primitive is an object
- The number data type can store positive and negative integers or floats, hexadecimal numbers, octal numbers, exponents, and the special numbers NaN, Infinity, and -Infinity
- The string data type contains characters in quotes
- The only values of the boolean data type are `true` and `false`
- The only value of the null data type is the value `null`
- The only value of the undefined data type is the value `undefined`
Primitive Data Types, Arrays, Loops, and Conditions

- All values become true when converted to a boolean, with the exception of the six falsy values:
  - ""
  - null
  - undefined
  - 0
  - NaN
  - false

Arrays

Now that you know the basic primitive data types in JavaScript, it's time to move to a more interesting data structure—the array.

To declare a variable that contains an empty array, you use square brackets with nothing between them:

```javascript
>>> var a = [];  
>>> typeof a;

"object"
```

typeof returns "object", but don't worry about this for the time being, we'll get to that when we take a closer look at objects.

To define an array that has three elements, you do this:

```javascript
>>> var a = [1,2,3];
```

When you simply type the name of the array in the Firebug console, it prints the contents of the array:

```javascript
>>> a

[1, 2, 3]
```

So what is an array exactly? It's simply a list of values. Instead of using one variable to store one value, you can use one array variable to store any number of values as elements of the array. Now the question is how to access each of these stored values?
The elements contained in an array are indexed with consecutive numbers starting from zero. The first element has index (or position) 0, the second has index 1 and so on. Here's the three-element array from the previous example:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

In order to access an array element, you specify the index of that element inside square brackets. So `a[0]` gives you the first element of the array `a`, `a[1]` gives you the second, and so on.

```python
>>> a[0]
1
>>> a[1]
2
```

### Adding/Updating Array Elements

Using the index, you can also update elements of the array. The next example updates the third element (index 2) and prints the contents of the new array.

```python
>>> a[2] = 'three';
"three"
>>> a
[1, 2, "three"]
```

You can add more elements, by addressing an index that didn't exist before.

```python
>>> a[3] = 'four';
"four"
>>> a
[1, 2, "three", "four"]
```
If you add a new element, but leave a gap in the array, those elements in between are all assigned the `undefined` value. Check out this example:

```javascript
>>> var a = [1, 2, 3];
>>> a[6] = 'new';

"new"

>>> a
[1, 2, 3, undefined, undefined, undefined, "new"]
```

**Deleting Elements**

In order to delete an element, you can use the `delete` operator. It doesn't actually remove the element, but sets its value to `undefined`. After the deletion, the length of the array does not change.

```javascript
>>> var a = [1, 2, 3];
>>> delete a[1];

true

>>> a
[1, undefined, 3]
```

**Arrays of arrays**

An array can contain any type of values, including other arrays.

```javascript
>>> var a = [1, "two", false, null, undefined];

>>> a
[1, "two", false, null, undefined]

>>> a[5] = [1, 2, 3]

[1, 2, 3]

>>> a
[1, "two", false, null, undefined, [1, 2, 3]]
```
Let's see an example where you have an array of two elements, each of them being an array.

```javascript
>>> var a = [[1,2,3],[4,5,6]];
>>> a
[[1, 2, 3], [4, 5, 6]]
```

The first element of the array is `a[0]` and it is an array itself.

```javascript
>>> a[0]
[1, 2, 3]
```

To access an element in the nested array, you refer to the element index in another set of square brackets.

```javascript
>>> a[0][0]
1
>>> a[1][2]
6
```

Note also that you can use the array notation to access individual characters inside a string.

```javascript
>>> var s = 'one';
>>> s[0]
"o"
>>> s[1]
"n"
>>> s[2]
"e"
```

There are more ways to have fun with arrays (and we'll get to that in Chapter 4), but let's stop here for now, remembering that:

- An array is a data store
- An array contains indexed elements
- Indexes start from zero and increment by one for each element in the array
- To access array elements we use the index in square brackets
- An array can contain any type of data, including other arrays
Conditions and Loops

Conditions provide a simple but powerful way to control the flow of execution through a piece of code. Loops allow you to perform repeating operations with less code. Let's take a look at:

- if conditions,
- switch statements,
- while, do-while, for, and for-in loops.

Code Blocks

Let's start by clarifying what a block of code is, as blocks are used extensively when constructing conditions and loops.

A block of code consists of zero or more expressions enclosed in curly brackets.

```javascript
{
    var a = 1;
    var b = 3;
}
```

You can nest blocks within each other indefinitely:

```javascript
{
    var a = 1;
    var b = 3;
    var c, d;
    {
        c = a + b;
        {
            d = a - b;
        }
    }
}
```
Best Practice Tips

- Use end-of-line semicolons. Although the semicolon is optional when you have one expression per line, it's good to develop the habit of using them. For best readability, the individual expressions inside a block should be placed one per line and separated by semi-colons.

- Indent any code placed within curly brackets. Some people use one tab indentation, some use four spaces, and some use two spaces. It really doesn't matter, as long as you're consistent. In the example above the outer block is indented with two spaces, the code in the first nested block is indented with four spaces and the innermost block is indented with six spaces.

- Use curly brackets. When a block consists of only one expression, the curly brackets are optional, but for readability and maintainability, you should get into the habit of always using them, even when they're optional.

Ready to jump into loops and ifs? Note that the examples in the following sections require you to switch to the multi-line Firebug console.

if Conditions

Here's a simple example of an if condition:

```javascript
var result = '';
if (a > 2) {
    result = 'a is greater than 2';
}
```

The parts of the if condition are:

- The if statement
- A condition in parentheses — "is a greater than 2?"
- Code block to be executed if the condition is satisfied

The condition (the part in parentheses) always returns a boolean value and may contain:

- A logical operation: !, && or ||
- A comparison, such as ===, !=, >, and so on
- Any value or variable that can be converted to a boolean
- A combination of the above
There can also be an optional else part of the if condition. The else statement is followed by a block of code to be executed if the condition was evaluated to false.

```javascript
if (a > 2) {
    result = 'a is greater than 2';
} else {
    result = 'a is NOT greater than 2';
}
```

In between the if and the else, there can also be an unlimited number of else if conditions. Here's an example:

```javascript
if (a > 2 || a < -2) {
    result = 'a is not between -2 and 2';
} else if (a === 0 && b === 0) {
    result = 'both a and b are zeros';
} else if (a === b) {
    result = 'a and b are equal';
} else {
    result = 'I give up';
}
```

You can also nest conditions by putting new conditions within any of the blocks.

```javascript
if (a === 1) {
    if (b === 2) {
        result = 'a is 1 and b is 2';
    } else {
        result = 'a is 1 but b is not 2';
    }
} else {
    result = 'a is not 1, no idea about b';
}
```

### Checking if a Variable Exists

It's often useful to check whether a variable exists. The laziest way to do this is simply putting the variable in the condition part of the if, for example if (somevar) {...}, but this is not necessarily the best method. Let's take a look at an example that tests whether a variable called somevar exists, and if so, sets the result variable to yes:

```javascript
>>> var result = '';
>>> if (somevar){result = 'yes';}

somevar is not defined

>>> result;

""
```
This code obviously works, because at the end result was not "yes". But firstly, the code generated a warning: **somevar is not defined** and as a JavaScript whiz you don't want your code to do anything like that. Secondly, just because `if (somevar)` returned false doesn't mean that `somevar` is not defined. It could be that `somevar` is defined and initialized but contains a falsy value, like false or 0.

A better way to check if a variable is defined is to use `typeof`.

```javascript
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
""
```

typeof will always return a string and you can compare this string with "undefined". Note that the variable `somevar` may have been declared but not assigned a value yet and you'll still get the same result. So when testing with typeof like this, you're really testing whether the variable has any value (other than the value undefined).

```javascript
>>> var somevar;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
""
```

```javascript
>>> somevar = undefined;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
""
```

If a variable is defined and initialized with any value other than undefined, its type returned by `typeof` is no longer "undefined".

```javascript
>>> somevar = 123;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
"yes"
```
Alternative if Syntax

When you have a very simple condition you can consider using an alternative if syntax. Take a look at this:

```javascript
var a = 1;
var result = ''; 
if (a === 1) {
    result = "a is one";
} else {
    result = "a is not one";
}
```

The if condition can be expressed simply as:

```javascript
var result = (a === 1) ? "a is one" : "a is not one";
```

You should only use this syntax for very simple conditions. Be careful not to abuse it, as it can easily make your code unreadable.

The ? is called **ternary operator**.

Switch

If you find yourself using an if condition and having too many else if parts, you could consider changing the if to a switch.

```javascript
var a = '1';
var result = '';
switch (a) {
    case 1:
        result = 'Number 1';
        break;
    case '1':
        result = 'String 1';
        break;
    default:
        result = 'I don\'t know';
        break;
}
result;
```

The result of executing this will be "String 1". Let's see what the parts of a switch are:

- The **switch** statement.
- Some expression in parentheses. The expression most often contains a variable, but can be anything that returns a value.
Chapter 2

- A number of case blocks enclosed in curly brackets.
- Each case statement is followed by an expression. The result of the expression is compared to the expression placed after the switch statement. If the result of the comparison is true, the code that follows the colon after the case is executed.
- There is an optional break statement to signal the end of the case block. If this break statement is reached, we're all done with the switch. Otherwise, if the break is missing, we enter the next case block, which should be avoided.
- There's an optional default statement, which is followed by a block of code that is executed if none of the previous cases evaluated to true.

In other words, the step-by-step procedure for executing a switch statement is as follows:

1. Evaluate the switch expression found in parentheses, remember it.
2. Move to the first case, compare its value with the one from step 1.
3. If the comparison in step 2 returns true, execute the code in the case block.
4. After the case block is executed, if there's a break statement at the end of it, exit the switch.
5. If there's no break or step 2 returned false, move on to the next case block. Repeat steps 2 to 5.
6. If we're still here (we didn't exit in step 4), execute the code following the default statement.

Best Practice Tips

- Indent the case line, and then further indent the code that follows it.
- Don't forget to break. Sometimes you may want to omit the break intentionally, but that's rare. It's called a fall-through and should always be documented because it may look like an accidental omission. On the other hand, sometimes you may want to omit the whole code block following a case and have two cases sharing the same code. This is fine, but doesn't change the rule that if there's code that follows a case statement, this code should end with a break. In terms of indentation, aligning the break with the case or with the code inside the case is a personal preference; again, being consistent is what matters.
- Use the default case. This will help you make sure you have a meaningful result after the switch, even if none of the cases matched the value being switched.
Loops

if-else and switch statements allow your code to take different paths, as if you're at a crossroads and decide which way to go depending on a condition. Loops, on the other hand, allow your code to take a few roundabouts before merging back into the main road. How many repetitions? That depends on the result of evaluating a condition before (or after) each iteration.

Let's say you are (your program execution is) traveling from A to B. At some point, you reach a place where you evaluate a condition C. The result of evaluating C tells you if you should go into a loop L. You make one iteration. Then you evaluate the condition once again to see if another iteration is needed. Eventually, you move on your way to B.

An infinite loop is when the condition is always true and your code gets stuck in the loop "forever". This is, of course, is a logical error and you should look out for such scenarios.

In JavaScript, there are four types of loops:

- while loops
- do-while loops
- for loops
- for-in loops

While Loops

while loops are the simplest type of loop. They look like this:

```javascript
var i = 0;
while (i < 10) {
    i++;
}
```

The while statement is followed by a condition in parentheses and a code block in curly brackets. As long as the condition evaluates to true, the code block is executed over and over again.
Do-while loops

do-while loops are a slight variation of the while loops. An example:

```javascript
var i = 0;
do {
i++;
} while (i < 10)
```

Here, the `do` statement is followed by a code block and a condition after the block. This means that the code block will always be executed, at least once, before the condition is evaluated.

If you initialize `i` to 11 instead of 0 in the last two examples, the code block in the first example (the while loop) will not be executed and `i` will still be 11 at the end, while in the second (do-while loop), the code block will be executed once and `i` will become 12.

For Loops

`for` is the most widely used type of loop and you should make sure you're comfortable with this one. It requires a just little bit more in terms of syntax.

In addition to the condition `C` and the code block `L`, you have the following:

- **Initialization**—some code that is executed *before* you even enter the loop (marked with 0 in the diagram)
- **Increment**—some code that is executed *after every iteration* (marked with `++` in the diagram)

The most widely used pattern of using a `for` loop is:

- In the initialization part you define a variable, most often called `i`, like this:
  ```javascript
  var i = 0;
  ```
- In the condition part you compare `i` to a boundary value, like `i < 100`
- In the increment part, you increase `i` by 1, like `i++`
Here's an example:

```javascript
var punishment = '';  
for (var i = 0; i < 100; i++) {
    punishment += 'I will never do this again, ';  
}
```

All three parts (initialization, condition, increment) can contain multiple expressions separated by commas. You can rewrite the example and define the variable `punishment` inside the initialization part of the loop.

```javascript
for (var i = 0, punishment = ''; i < 100; i++) {
    punishment += 'I will never do this again, ';  
}
```

Can you move the body of the loop inside the increment part? Yes, you can, especially as it's a one-liner. This will give you a loop that looks a little awkward, as it has no body:

```javascript
for (var i = 0, punishment = ''; i < 100;
    i++, punishment += 'I will never do this again, ')
{
    // nothing here
}
```

These three parts are actually all optional. Here's another way of rewriting the same example:

```javascript
var i = 0, punishment = '';  
for (;;) {
    punishment += 'I will never do this again, ';  
    if (++i == 100) {
        break;
    }
}
```

Although the last rewrite works exactly the same way as the original, it is longer and harder to read. It's also possible to achieve the same result by using a `while` loop. But `for` loops make the code tighter and more robust, because the mere syntax of the `for` loop makes you think about the three parts (initialization, condition, increment) and thus, helps you reconfirm your logic and avoid situations such as being stuck in an infinite loop.
for loops can be nested within each other. Here's an example of a loop that is nested inside another loop and assembles a string containing 10 rows and 10 columns of asterisks. Think of i being the row and j being the column of an "image".

```javascript
var res = '
';
for(var i = 0; i < 10; i++) {
    for(var j = 0; j < 10; j++) {
        res += '* ';  
    }
    res+= '
';
}
```

The result is a string like:

```
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

Here's another example that uses nested loops and a modulus operation in order to draw a little snowflake-like result.

```javascript
var res = '\n', i, j;
for(i = 1; i <= 7; i++) {
    for(j = 1; j <= 15; j++) {
        res += (i * j) % 8 ? ' ' : '*';
    }
    res+= '\n';
}
```
**For-in Loops**

The `for-in` loop is used to iterate over the elements of an array (or an object, as we'll see later). This is its only use; it cannot be used as a general-purpose repetition mechanism that replaces `for` or `while`. Let's see an example of using a `for-in` to loop through the elements of an array. But bear in mind that this is for informational purposes only, as `for-in` is mostly suitable for objects, and the regular `for` loop should be used for arrays.

In this example, we'll iterate over all of the elements of an array and print out the index (the key) and the value of each element:

```javascript
var a = ['a', 'b', 'c', 'x', 'y', 'z'];
var result = '\n';
for (var i in a) {
    result += 'index: ' + i + ', value: ' + a[i] + '\n';
}
```

The result is:

```
index: 0, value: a
index: 1, value: b
index: 2, value: c
index: 3, value: x
```
index: 4, value: y
index: 5, value: z

Comments
One last thing for this chapter: comments. Inside your JavaScript code you can put comments. These are ignored by the JavaScript engine and don't have any effect on how the program works. But they can be invaluable when you revisit your code after a few months, or transfer the code to someone else for maintenance.

Two types of comments are allowed:

- Single line comments — start with `//` and end at the end of the line
- Multi-line comments — start with `/*` and end with `*/` on the same line or any subsequent line. Note that any code in between the comment start and the comment end will be ignored.

Some examples:

```javascript
// beginning of line
var a = 1; // anywhere on the line
/* multi-line comment on a single line */
/*
   comment
   that spans
   several lines
*/
```

There are even utilities, such as JSDoc, that can parse your code and extract meaningful documentation based on your comments.

Summary
In this chapter, you learned a lot about the basic building blocks of a JavaScript program. Now you know the primitive data types:

- number
- string
- boolean
- undefined
- null

For More Information:
Primitive Data Types, Arrays, Loops, and Conditions

You also know quite a few operators:

- Arithmetic operators: +, -, *, /, and %.
- Increment operators: ++ and --.
- Assignment operators: =, +=, -=, *=, /=, and %=.
- Special operators: `typeof` and `delete`.
- Logical operators: `&&`, `||`, and `!`.
- Comparison operators: `==`, `===`, `!=`, `!==`, `<`, `>`, `>=`, and `<=`.

Then you learned how to use arrays to store and access data, and finally you saw different ways to control the flow of your program—using conditions (if-else or `switch`) and loops (while, do-while, for, for-in).

This is quite a bit of information and it is recommended that you now go through the exercises below, then give yourself a well-deserved pat on the back before diving into the next chapter. More fun is coming up!

**Exercises**

1. What is the result of executing each of these lines in the console? Why?
   - `var a; typeof a;`
   - `var s = 'ls'; s++;
   - `!!"false"
   - `!!undefined`
   - `typeof -Infinity`
   - `10 % "0"
   - `undefined == null`
   - `false === ""`
   - `typeof "2E+2"
   - `a = 3e+3; a++;`

2. What is the value of `v` after the following?
   ```javascript
   >>> var v = v || 10;
   ```
   Experiment by first setting `v` to 100, 0, null, or unset it (`delete v`).

3. Write a script that prints out the multiplication table. **Hint:** use a loop nested inside another loop.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.