Basic JavaScript: values, variables, and control flow

¶Inside the computer's world, there is only data. That which is not data, does not exist. Although all data is in essence just a sequence of bits[1], and is thus fundamentally alike, every piece of data plays its own role. In JavaScript's system, most of this data is neatly separated into things called values. Every value has a type, which determines the kind of role it can play. There are six basic types of values: Numbers, strings, booleans, objects, functions, and undefined values.

¶To create a value, one must merely invoke its name. This is very convenient. You don't have to gather building material for your values, or pay for them, you just call for one and *woosh*, you have it. They are not created from thin air, of course. Every value has to be stored somewhere, and if you want to use a gigantic number of them at the same time you might run out of computer memory. Fortunately, this is only a problem if you need them all simultaneously. As soon as you no longer use a value, it will dissipate, leaving behind only a few bits. These bits are recycled to make the next generation of values.

¶Values of the type number are, as you might have deduced, numeric values. They are written the way numbers are usually written:

```
144
```

¶Enter that in the console, and the same thing is printed in the output window. The text you typed in gave rise to a number value, and the console took this number and wrote it out to the screen again. In a case like this, that was a rather pointless exercise, but soon we will be producing values in less straightforward ways, and it can be useful to 'try them out' on the console to see what they produce.

¶This is what `144` looks like in bits[2]:

```
0100000001100010000000000000000000000000000000000000000000000000
```

¶The number above has 64 bits. Numbers in JavaScript always do. This has one important repercussion: There is a limited amount of different numbers that can be expressed. With three decimal digits, only the numbers 0 to 999 can be written, which is 10^3 = 1000 different numbers. With 64 binary digits, 2^{64} different numbers can be written. This is a lot, more than 10^{19} (a one with nineteen zeroes).

¶Not all whole numbers below 10^{19} fit in a JavaScript number though. For one, there are also negative numbers, so one of the bits has to be used to store the sign of the number. A bigger issue is that non-whole numbers must also be represented. To do this, 11 bits are used to store the position of the fractional dot within the number.

¶That leaves 52 bits[3]. Any whole number less than 2^{52} (which is more than 10^{15}) will safely fit in a JavaScript number. In most cases, the numbers we are using stay well below that, so we do not have to concern ourselves with bits at all. Which is good. I have nothing in particular against bits, but you *do* need a terrible lot of them to get anything done. When at all possible, it is more pleasant to deal with bigger things.

¶Fractional numbers are written by using a dot.

```
9.81
```

¶For very big or very small numbers, one can also use 'scientific' notation by adding an `e`, followed by the exponent of the number:

```
2.998e8
```

¶Which is 2.998 * 10^8 = 299800000.

¶Calculations with whole numbers (also called integers) that fit in 52 bits are guaranteed to always be precise. Unfortunately, calculations with fractional numbers are generally not. The same way that π (pi) can not be precisely expressed by a finite amount of decimal digits, many numbers lose some precision when only 64 bits are available to store them. This is a shame, but it only causes practical problems in very specific situations. The important thing is to be aware of it, and treat fractional digital numbers as approximations, not as precise values.

¶The main thing to do with numbers is arithmetic. Arithmetic operations such as addition or multiplication take two number values and produce a new number from them. Here is what they look like in JavaScript:

```
100 + 4 * 11
```

¶The `+` and `*` symbols are called operators. The first stands for addition, and the second for multiplication. Putting an operator between two values will apply it to those values, and produce a new value.

¶Does the example mean 'add 4 and 100, and multiply the result by 11', or is the multiplication done before the adding? As you might have guessed, the multiplication happens first. But, as in mathematics, this can be changed by wrapping the addition in parentheses:

```
(100 + 4) * 11
```

¶For subtraction, there is the `-` operator, and division can be done with `/`. When operators appear together without parentheses, the order in which they are applied is determined by the precedence of the operators. The first example shows that multiplication has a higher precedence than addition. Division and multiplication always come before subtraction and addition. When multiple operators with the same precedence appear next to each other (`1 - 1 + 1`) they are applied left-to-right.

¶Try to figure out what value this produces, and then run it to see if you were correct...

```
115 * 4 - 4 + 88 / 2
```

¶These rules of precedence are not something you should worry about. When in doubt, just add parentheses.

¶There is one more arithmetic operator which is probably less familiar to you. The `%` symbol is used to represent theremainder operation. `X % Y` is the remainder of dividing `X` by `Y`. For example `314 % 100` is `14`, `10 % 3` is `1`, and`144 % 12` is `0`. Remainder has the same precedence as multiplication and division.

¶The next data type is the string. Its use is not as evident from its name as with numbers, but it also fulfills a very basic role. Strings are used to represent text, the name supposedly derives from the fact that it strings together a bunch of characters. Strings are written by enclosing their content in quotes:

```
"Patch my boat with chewing gum."
```

¶Almost anything can be put between double quotes, and JavaScript will make a string value out of it. But a few characters are tricky. You can imagine how putting quotes between quotes might be hard. Newlines, the things you get when you press enter, can also not be put between quotes, the string has to stay on a single line.

¶To be able to have such characters in a string, the following trick is used: Whenever a backslash (`'\'`) is found inside quoted text, it indicates that the character after it has a special meaning. A quote that is preceded by a backslash will not end the string, but be part of it. When an '`n`' character occurs after a backslash, it is interpreted as a newline. Similarly, a '`t`' after a backslash means a tab character[4].

```
"This is the first line\nAnd this is the second"
```

¶Note that if you type this into the console, it'll display it back in 'source' form, with the quotes and backslash escapes. To see only the actual text, you can type `print("a\nb")`. What that does precisely will be clarified a little further on.

¶There are of course situations where you want a backslash in a string to be just a backslash, not a special code. If two backslashes follow each other, they will collapse right into each other, and only one will be left in the resulting string value:

```
"A newline character is written like \"\\n\"."
```

¶Strings can not be divided, multiplied, or subtracted. The `+` operator *can* be used on them. It does not add, but it concatenates, it glues two strings together.

```
"con" + "cat" + "e" + "nate"
```

¶There are more ways of manipulating strings, but these are discussed later.

¶Not all operators are symbols, some are written as words. For example, the `typeof` operator, which produces a string value naming the type of the value you give it.

```
typeof 4.5
```

¶The other operators we saw all operated on two values, `typeof` takes only one. Operators that use two values are called binary operators, while those that take one are called unary operators. The minus operator can be used both as a binary and a unary operator:

```
- (10 - 2)
```

¶Then there are values of the boolean type. There are only two of these: `true` and `false`. Here is one way to produce a `true` value:

```
3 > 2
```

¶And `false` can be produced like this:

```
3 < 2
```

¶I hope you have seen the `>` and `<` signs before. They mean, respectively, 'is greater than' and 'is less than'. They are binary operators, and the result of applying them is a boolean value that indicates whether they hold in this case.

¶Strings can be compared in the same way:

```
"Aardvark" < "Zoroaster"
```

¶The way strings are ordered is more or less alphabetic. More or less... Uppercase letters are always 'less' than lowercase ones, so `"Z" < "a"` (upper-case Z, lower-case a) is `true`, and non-alphabetic characters (`'!'`, `'@'`, etc) are also included in the ordering. The actual way in which the comparison is done is based on the Unicode standard. This standard assigns a number to virtually every character one would ever need, including characters from Greek, Arabic, Japanese, Tamil, and so on. Having such numbers is practical for storing strings inside a computer — you can represent them as a list of numbers. When comparing strings, JavaScript just compares the numbers of the characters inside the string, from left to right.

¶Other similar operators are `>=` ('is greater than or equal to'), `<=` (is less than or equal to), `==` ('is equal to'), and `!=`('is not equal to').

```
"Itchy" != "Scratchy"
5e2 == 500
```

¶There are also some useful operations that can be applied to boolean values themselves. JavaScript supports three logical operators: *and*, *or*, and *not*. These can be used to 'reason' about booleans.

¶The `&&` operator represents logical *and*. It is a binary operator, and its result is only `true` if both of the values given to it are `true`.

```
true && false
```

¶`||` is the logical *or*, it is `true` if either of the values given to it is `true`:

```
true || false
```

¶*Not* is written as an exclamation mark, `!`, it is a unary operator that flips the value given to it, `!true` is `false`, and`!false` is `true`.

Ex. 2.1

```
((4 >= 6) || ("grass" != "green")) &&    !(((12 * 2) == 144) && true)
```

¶Is this true? For readability, there are a lot of unnecessary parentheses in there. This simple version means the same thing:

```
(4 >= 6 || "grass" != "green") &&    !(12 * 2 == 144 && true)
```

[show solution]

¶It is not always obvious when parentheses are needed. In practice, one can usually get by with knowing that of the operators we have seen so far, `||` has the lowest precedence, then comes `&&`, then the comparison operators (`>`, `==`, etcetera), and then the rest. This has been chosen in such a way that, in simple cases, as few parentheses as possible are necessary.

¶All the examples so far have used the language like you would use a pocket calculator. Make some values and apply operators to them to get new values. Creating values like this is an essential part of every JavaScript program, but it is only a part. A piece of code that produces a value is called an expression. Every value that is written directly (such as `22` or `"psychoanalysis"`) is an expression. An expression between parentheses is also an expression. And a binary operator applied to two expressions, or a unary operator applied to one, is also an expression.

¶There are a few more ways of building expressions, which will be revealed when the time is ripe.

¶There exists a unit that is bigger than an expression. It is called a statement. A program is built as a list of statements. Most statements end with a semicolon (`;`). The simplest kind of statement is an expression with a semicolon after it. This is a program:

```
1; !false;
```

¶It is a useless program. An expression can be content to just produce a value, but a statement only amounts to something if it somehow changes the world. It could print something to the screen — that counts as changing the

world — or it could change the internal state of the program in a way that will affect the statements that come after it. These changes are called 'side effects'. The statements in the example above just produce the values `1` and `true`, and then immediately throw them into the bit bucket[5]. This leaves no impression on the world at all, and is not a side effect.

¶How does a program keep an internal state? How does it remember things? We have seen how to produce new values from old values, but this does not change the old values, and the new value has to be immediately used or it will dissipate again. To catch and hold values, JavaScript provides a thing called a variable.

```
var caught = 5 * 5;
```

¶A variable always has a name, and it can point at a value, holding on to it. The statement above creates a variable called `caught` and uses it to grab hold of the number that is produced by multiplying `5` by `5`.

¶After running the above program, you can type the word `caught` into the console, and it will retrieve the value `25`for you. The name of a variable is used to fetch its value. `caught + 1` also works. A variable name can be used as an expression, and thus can be part of bigger expressions.

¶The word `var` is used to create a new variable. After `var`, the name of the variable follows. Variable names can be almost every word, but they may not include spaces. Digits can be part of variable names, `catch22` is a valid name, but the name must not start with a digit. The characters '`$`' and '`_`' can be used in names as if they were letters, so`$_$` is a correct variable name.

¶If you want the new variable to immediately capture a value, which is often the case, the `=` operator can be used to give it the value of some expression.

¶When a variable points at a value, that does not mean it is tied to that value forever. At any time, the `=` operator can be used on existing variables to yank them away from their current value and make them point to a new one.

```
caught = 4 * 4;
```

¶You should imagine variables as tentacles, rather than boxes. They do not *contain* values, they *grasp* them — two variables can refer to the same value. Only the values that the program still has a hold on can be accessed by it. When you need to remember something, you grow a tentacle to hold on to it, or re-attach one of your existing tentacles to a new value: To remember the amount of dollars that Luigi still owes you, you could do...

```
var luigiDebt = 140;
```

¶Then, every time Luigi pays something back, this amount can be decremented by giving the variable a new number:

```
luigiDebt = luigiDebt - 35;
```

¶The collection of variables and their values that exist at a given time is called the environment. When a program starts up, this environment is not empty. It always contains a number of standard variables. When your browser loads a page, it creates a new environment and attaches these standard values to it. The variables created and modified by programs on that page survive until the browser goes to a new page.

¶A lot of the values provided by the standard environment have the type 'function'. A function is a piece of program wrapped in a value. Generally, this piece of program does something useful, which can be invoked using the function

value that contains it. In a browser environment, the variable `alert` holds a function that shows a little dialog window with a message. It is used like this:

```
alert("Avocados");
```

¶Executing the code in a function is called invoking, calling, or applying it. The notation for doing this uses parentheses. Every expression that produces a function value can be invoked by putting parentheses after it. In the example, the value `"Avocados"` is given to the function, which uses it as the text to show in the dialog window. Values given to functions are called parameters or arguments. `alert` needs only one of them, but other functions might need a different number.

¶Showing a dialog window is a side effect. A lot of functions are useful because of the side effects they produce. It is also possible for a function to produce a value, in which case it does not need to have a side effect to be useful. For example, there is a function `Math.max`, which takes any number of numeric arguments and gives back the greatest:

```
alert(Math.max(2, 4));
```

¶When a function produces a value, it is said to return it. Because things that produce values are always expressions in JavaScript, function calls can be used as a part of bigger expressions:

```
alert(Math.min(2, 4) + 100);
```

¶Chapter 3 discusses writing your own functions.

¶As the previous examples show, `alert` can be useful for showing the result of some expression. Clicking away all those little windows can get on one's nerves though, so from now on we will prefer to use a similar function, called`print`, which does not pop up a window, but just writes a value to the output area of the console. `print` is not a standard JavaScript function, browsers do not provide it for you, but it is made available by this book, so you can use it on these pages.

```
print("N");
```

¶A similar function, also provided on these pages, is `show`. While `print` will display its argument as flat text, `show`tries to display it the way it would look in a program, which can give more information about the type of the value. For example, string values keep their quotes when given to `show`:

```
show("N");
```

¶The standard environment provided by browsers contains a few more functions for popping up windows. You can ask the user an OK/Cancel question using `confirm`. This returns a boolean, `true` if the user presses 'OK', and `false` if he presses 'Cancel'.

```
show(confirm("Shall we, then?"));
```

¶`prompt` can be used to ask an 'open' question. The first argument is the question, the second one is the text that the user starts with. A line of text can be typed into the window, and the function will return this as a string.

```
show(prompt("Tell us everything you know.", "..."));
```

¶It is possible to give almost every variable in the environment a new value. This can be useful, but also dangerous. If you give `print` the value `8`, you won't be able to print things anymore. Fortunately, there is a big 'Reset' button on the console, which will reset the environment to its original state.

¶One-line programs are not very interesting. When you put more than one statement into a program, the statements are, predictably, executed one at a time, from top to bottom.

```
var theNumber = Number(prompt("Pick a number", "")); print("Your number is the
square root of " +       (theNumber * theNumber));
```

¶The function `Number` converts a value to a number, which is needed in this case because the result of `prompt` is a string value. There are similar functions called `String` and `Boolean` which convert values to those types.

¶Consider a program that prints out all even numbers from 0 to 12. One way to write this is:

```
print(0); print(2); print(4); print(6); print(8); print(10); print(12);
```

¶That works, but the idea of writing a program is to make something *less* work, not more. If we needed all even numbers below 1000, the above would be unworkable. What we need is a way to automatically repeat some code.

```
var currentNumber = 0; while (currentNumber <= 12) {   print(currentNumber);
currentNumber = currentNumber + 2; }
```

¶You may have seen `while` in the introduction chapter. A statement starting with the word `while` creates a loop. A loop is a disturbance in the sequence of statements — it may cause the program to repeat some statements multiple times. In this case, the word `while` is followed by an expression in parentheses (the parentheses are compulsory here), which is used to determine whether the loop will loop or finish. As long as the boolean value produced by this expression is `true`, the code in the loop is repeated. As soon as it is false, the program goes to the bottom of the loop and continues as normal.

¶The variable `currentNumber` demonstrates the way a variable can track the progress of a program. Every time the loop repeats, it is incremented by `2`, and at the beginning of every repetition, it is compared with the number `12` to decide whether to keep on looping.

¶The third part of a `while` statement is another statement. This is the body of the loop, the action or actions that must take place multiple times. If we did not have to print the numbers, the program could have been:

```
var currentNumber = 0; while (currentNumber <= 12)   currentNumber = currentNumber
+ 2;
```

¶Here, `currentNumber = currentNumber + 2;` is the statement that forms the body of the loop. We must also print the number, though, so the loop statement must consist of more than one statement. Braces (`{` and `}`) are used to group statements into blocks. To the world outside the block, a block counts as a single statement. In the earlier example, this is used to include in the loop both the call to `print` and the statement that updates`currentNumber`.

Ex. 2.2

¶Use the techniques shown so far to write a program that calculates and shows the value of 2^{10} (2 to the 10th power). You are, obviously, not allowed to use a cheap trick like just writing `2 * 2 * ...`.

¶If you are having trouble with this, try to see it in terms of the even-numbers example. The program must perform an action a certain amount of times. A counter variable with a `while` loop can be used for that. Instead of printing the counter, the program must multiply something by 2. This something should be another variable, in which the result value is built up.

¶Don't worry if you don't quite see how this would work yet. Even if you perfectly understand all the techniques this chapter covers, it can be hard to apply them to a specific problem. Reading and writing code will help develop a feeling for this, so study the solution, and try the next exercise.

[show solution]

Ex. 2.3

¶With some slight modifications, the solution to the previous exercise can be made to draw a triangle. And when I say 'draw a triangle' I mean 'print out some text that almost looks like a triangle when you squint'.

¶Print out ten lines. On the first line there is one '#' character. On the second there are two. And so on.

¶How does one get a string with X '#' characters in it? One way is to build it every time it is needed with an 'inner loop' — a loop inside a loop. A simpler way is to reuse the string that the previous iteration of the loop used, and add one character to it.

[show solution]

¶You will have noticed the spaces I put in front of some statements. These are not required: The computer will accept the program just fine without them. In fact, even the line breaks in programs are optional. You could write them as a single long line if you felt like it. The role of the indentation inside blocks is to make the structure of the code clearer to a reader. Because new blocks can be opened inside other blocks, it can become hard to see where one block ends and another begins in a complex piece of code. When lines are indented, the visual shape of a program corresponds to the shape of the blocks inside it. I like to use two spaces for every open block, but tastes differ.

¶The field in the console where you can type programs will help you by automatically adding these spaces. This may seem annoying at first, but when you write a lot of code it becomes a huge time-saver. Pressing shift+tab will re-indent the line your cursor is currently on.

¶In some cases, JavaScript allows you to omit the semicolon at the end of a statement. In other cases, it has to be there or strange things will happen. The rules for when it can be safely omitted are complex and weird. In this book, I won't leave out any semicolons, and I strongly urge you to do the same in your own programs.

¶The uses of `while` we have seen so far all show the same pattern. First, a 'counter' variable is created. This variable tracks the progress of the loop. The `while` itself contains a check, usually to see whether the counter has reached some boundary yet. Then, at the end of the loop body, the counter is updated.

¶A lot of loops fall into this pattern. For this reason, JavaScript, and similar languages, also provide a slightly shorter and more comprehensive form:

```
for (var number = 0; number <= 12; number = number + 2)    show(number);
```

¶This program is exactly equivalent to the earlier even-number-printing example. The only change is that all the statements that are related to the 'state' of the loop are now on one line. The parentheses after the `for` should contain two semicolons. The part before the first semicolon *initialises* the loop, usually by defining a variable. The

second part is the expression that *checks* whether the loop must still continue. The final part *updates* the state of the loop. In most cases this is shorter and clearer than a `while` construction.

¶I have been using some rather odd capitalisation in some variable names. Because you can not have spaces in these names — the computer would read them as two separate variables — your choices for a name that is made of several words are more or less limited to the following: `fuzzylittleturtle`, `fuzzy_little_turtle`,`FuzzyLittleTurtle`, or `fuzzyLittleTurtle`. The first one is hard to read. Personally, I like the one with the underscores, though it is a little painful to type. However, the standard JavaScript functions, and most JavaScript programmers, follow the last one. It is not hard to get used to little things like that, so I will just follow the crowd and capitalise the first letter of every word after the first.

¶In a few cases, such as the `Number` function, the first letter of a variable is also capitalised. This was done to mark this function as a constructor. What a constructor is will become clear in chapter 8. For now, the important thing is not to be bothered by this apparent lack of consistency.

¶Note that names that have a special meaning, such as `var`, `while`, and `for` may not be used as variable names. These are called keywords. There are also a number of words which are 'reserved for use' in future versions of JavaScript. These are also officially not allowed to be used as variable names, though some browsers do allow them. The full list is rather long:

```
abstract boolean break byte case catch char class const continue debugger default
delete do double else enum export extends false final finally float for function
goto if implements import in instanceof int interface long native new null package
private protected public return short static super switch synchronized this throw
throws transient true try typeof var void volatile while with
```

¶Don't worry about memorising these for now, but remember that this might be the problem when something does not work as expected. In my experience, `char` (to store a one-character string) and `class` are the most common names to accidentally use.

Ex. 2.4

¶Rewrite the solutions of the previous two exercises to use `for` instead of `while`.

[show solution]

¶A program often needs to 'update' a variable with a value that is based on its previous value. For example `counter = counter + 1`. JavaScript provides a shortcut for this: `counter += 1`. This also works for many other operators, for example `result *= 2` to double the value of `result`, or `counter -= 1` to count downwards.

¶For `counter += 1` and `counter -= 1` there are even shorter versions: `counter++` and `counter--`.

¶Loops are said to affect the control flow of a program. They change the order in which statements are executed. In many cases, another kind of flow is useful: skipping statements.

¶We want to show all numbers below 20 which are divisible both by 3 and by 4.

```
for (var counter = 0; counter < 20; counter++) {   if (counter % 3 == 0 && counter
% 4 == 0)     show(counter); }
```

¶The keyword `if` is not too different from the keyword `while`: It checks the condition it is given (between parentheses), and executes the statement after it based on this condition. But it does this only once, so that the statement is executed zero or one time.

¶The trick with the remainder (`%`) operator is an easy way to test whether a number is divisible by another number. If it is, the remainder of their division, which is what remainder gives you, is zero.

¶If we wanted to print all numbers below 20, but put parentheses around the ones that are not divisible by 4, we can do it like this:

```
for (var counter = 0; counter < 20; counter++) {   if (counter % 4 == 0)
print(counter);   if (counter % 4 != 0)     print("(" + counter + ")"); }
```

¶But now the program has to determine whether `counter` is divisible by `4` two times. The same effect can be obtained by appending an `else` part after an `if` statement. The `else` statement is executed only when the `if`'s condition is false.

```
for (var counter = 0; counter < 20; counter++) {   if (counter % 4 == 0)
print(counter);   else     print("(" + counter + ")"); }
```

¶To stretch this trivial example a bit further, we now want to print these same numbers, but add two stars after them when they are greater than 15, one star when they are greater than 10 (but not greater than 15), and no stars otherwise.

```
for (var counter = 0; counter < 20; counter++) {   if (counter > 15)
print(counter + "**");   else if (counter > 10)     print(counter + "*");   else
print(counter); }
```

¶This demonstrates that you can chain `if` statements together. In this case, the program first looks if `counter` is greater than `15`. If it is, the two stars are printed and the other tests are skipped. If it is not, we continue to check if`counter` is greater than `10`. Only if `counter` is also not greater than `10` does it arrive at the last `print` statement.

Ex. 2.5

¶Write a program to ask yourself, using `prompt`, what the value of 2 + 2 is. If the answer is "4", use `alert` to say something praising. If it is "3" or "5", say "Almost!". In other cases, say something mean.

[show solution]

¶When a loop does not always have to go all the way through to its end, the `break` keyword can be useful. It immediately jumps out of the current loop, continuing after it. This program finds the first number that is greater than or equal to 20 and divisible by 7:

```
for (var current = 20; ; current++) {   if (current % 7 == 0)     break; }
print(current);
```

¶The `for` construct shown above does not have a part that checks for the end of the loop. This means that it is dependent on the `break` statement inside it to ever stop. The same program could also have been written as simply...

```
for (var current = 20; current % 7 != 0; current++)   ; print(current);
```

¶In this case, the body of the loop is empty. A lone semicolon can be used to produce an empty statement. Here, the only effect of the loop is to increment the variable `current` to its desired value. But I needed an example that uses`break`, so pay attention to the first version too.

Ex. 2.6

¶Add a `while` and optionally a `break` to your solution for the previous exercise, so that it keeps repeating the question until a correct answer is given.

¶Note that `while (true) ...` can be used to create a loop that does not end on its own account. This is a bit silly, you ask the program to loop as long as `true` is `true`, but it is a useful trick.

[show solution]

¶In the solution to the previous exercise there is a statement `var answer;`. This creates a variable named `answer`, but does not give it a value. What happens when you take the value of this variable?

```
var mysteryVariable; show(mysteryVariable);
```

¶In terms of tentacles, this variable ends in thin air, it has nothing to grasp. When you ask for the value of an empty place, you get a special value named `undefined`. Functions which do not return an interesting value, such as `print`and `alert`, also return an `undefined` value.

```
show(alert("I am a side effect."));
```

¶There is also a similar value, `null`, whose meaning is 'this variable is defined, but it does not have a value'. The difference in meaning between `undefined` and `null` is mostly academic, and usually not very interesting. In practical programs, it is often necessary to check whether something 'has a value'. In these cases, the expression`something == undefined` may be used, because, even though they are not exactly the same value, `null == undefined` will produce `true`.

¶Which brings us to another tricky subject...

```
show(false == 0); show("" == 0); show("5" == 5);
```

¶All these give the value `true`. When comparing values that have different types, JavaScript uses a complicated and confusing set of rules. I am not going to try to explain them precisely, but in most cases it just tries to convert one of the values to the type of the other value. However, when `null` or `undefined` occur, it only produces `true` if both sides are `null` or `undefined`.

¶What if you want to test whether a variable refers to the value `false`? The rules for converting strings and numbers to boolean values state that `0` and the empty string count as `false`, while all the other values count as `true`. Because of this, the expression `variable == false` is also `true` when `variable` refers to `0` or `""`. For cases like this, where you do *not* want any automatic type conversions to happen, there are two extra operators: `===` and `!==`. The first tests whether a value is precisely equal to the other, and the second tests whether it is not precisely equal.

```
show(null === undefined); show(false === 0); show("" === 0); show("5" === 5);
```

¶All these are `false`.

¶Values given as the condition in an `if`, `while`, or `for` statement do not have to be booleans. They will be automatically converted to booleans before they are checked. This means that the number `0`, the empty string `""`,`null`, `undefined`, and of course `false`, will all count as false.

¶The fact that all other values are converted to `true` in this case makes it possible to leave out explicit comparisons in many situations. If a variable is known to contain either a string or `null`, one could check for this very simply...

```
var maybeNull = null; // ... mystery code that might put a string into maybeNull
... if (maybeNull)   print("maybeNull has a value");
```

¶... except in the case where the mystery code gives `maybeNull` the value `""`. An empty string is false, so nothing is printed. Depending on what you are trying to do, this might be *wrong*. It is often a good idea to add an explicit `=== null` or `=== false` in cases like this to prevent subtle mistakes. The same occurs with number values that might be `0`.

¶The line that talks about 'mystery code' in the previous example might have looked a bit suspicious to you. It is often useful to include extra text in a program. The most common use for this is adding some explanations in human language to a program.

```
// The variable counter, which is about to be defined, is going // to start with a
value of 0, which is zero. var counter = 0; // Now, we are going to loop, hold on
to your hat. while (counter < 100 /* counter is less than one hundred */) /* Every
time we loop, we INCREMENT the value of counter,    Seriously, we just add one to
it. */   counter++; // And then, we are done.
```

¶This kind of text is called a comment. The rules are like this: '`/*`' starts a comment that goes on until a '`*/`' is found. '`//`' starts another kind of comment, which goes on until the end of the line.

¶As you can see, even the simplest programs can be made to look big, ugly, and complicated by simply adding a lot of comments to them.

¶There are some other situations that cause automatic type conversions to happen. If you add a non-string value to a string, the value is automatically converted to a string before it is concatenated. If you multiply a number and a string, JavaScript tries to make a number out of the string.

```
show("Apollo" + 5); show(null + "ify"); show("5" * 5); show("strawberry" * 5);
```

¶The last statement prints `NaN`, which is a special value. It stands for 'not a number', and is of type number (which might sound a little contradictory). In this case, it refers to the fact that a strawberry is not a number. All arithmetic operations on the value `NaN` result in `NaN`, which is why multiplying it by `5`, as in the example, still gives a `NaN` value. Also, and this can be disorienting at times, `NaN == NaN` equals `false`, checking whether a value is `NaN` can be done with the `isNaN` function. `NaN` is another (the last) value that counts as `false` when converted to a boolean.

¶These automatic conversions can be very convenient, but they are also rather weird and error prone. Even though `+`and `*` are both arithmetic operators, they behave completely different in the example. In my own code, I use `+` to combine strings and non-strings a lot, but make it a point not to use `*` and the other numeric operators on string values. Converting a number to a string is always possible and straightforward, but converting a string to a number may not even work (as in the last line of the example). We can use `Number` to explicitly convert the string to a number, making it clear that we might run the risk of getting a `NaN` value.

```
show(Number("5") * 5);
```

¶When we discussed the boolean operators `&&` and `||` earlier, I claimed they produced boolean values. This turns out to be a bit of an oversimplification. If you apply them to boolean values, they will indeed return booleans. But they can also be applied to other kinds of values, in which case they will return one of their arguments.

¶What `||` really does is this: It looks at the value to the left of it first. If converting this value to a boolean would produce `true`, it returns this left value, otherwise it returns the one on its right. Check for yourself that this does the correct thing when the arguments are booleans. Why does it work like that? It turns out this is very practical.

Consider this example:

```
var input = prompt("What is your name?", "Kilgore Trout"); print("Well hello " +
(input || "dear"));
```

¶If the user presses 'Cancel' or closes the `prompt` dialog in some other way without giving a name, the variable`input` will hold the value `null` or `""`. Both of these would give `false` when converted to a boolean. The expression`input || "dear"` can in this case be read as 'the value of the variable `input`, or else the string `"dear"`'. It is an easy way to provide a 'fallback' value.

¶The `&&` operator works similarly, but the other way around. When the value to its left is something that would give`false` when converted to a boolean, it returns that value, otherwise it returns the value on its right.

¶Another property of these two operators is that the expression to their right is only evaluated when necessary. In the case of `true || X`, no matter what `X` is, the result will be `true`, so `X` is never evaluated, and if it has side effects they never happen. The same goes for `false && X`.