Reader

Eloquent JavaScript

Open Universiteit
Faculteit Management, Science & Technology

Cursusteam
ir. S. Stuurman, *cursusteamleider en auteur*
dr. ir. H.J.M. Passier, *auteur*
drs. H.J. Pootjes, *auteur*

Extern referent
prof. dr. ir. G.J. Houben (TU Delft)

Programmaleiding
prof. dr. M.C.J.D. van Eekelen

READER

Eloquent JavaScript

Open Universiteit
www.ou.nl

Productie
Open Universiteit

Redactie
Arnold van der Leer

Lay-out en illustraties
Maria Wienbröker-Kampermann

Omslag
Team Visuele communicatie, Open Universiteit

Druk- en bindwerk
OCÉ Business Services

Eerste druk: 2010
Tweede, gewijzigde druk: 2014

T58221_50089_14082014

ISBN 978 94 91825 41 5
ISBN 978 94 91825 44 6 (reader)

Cursuscode T.58.2.2.1

Structuur van de cursus Webapplicaties: de clientkant

Onderdeel	Deel	Blok	Leereenheid	Bladzijde
Werkboek	1		Introductie tot de cursus	
		1 De context	1 Internet en het World Wide Web	
		2 HTML en CSS	2 HTML	
			3 CSS	
		3 JavaScript	4 Inleiding JavaScript	
			5 Functies, objecten, arrays en exceptions	
			6 Functionele aspecten van JavaScript	
			7 Modulen en objecten in JavaScript	
			Lijst van Programmeeraanwijzingen	
			Lijst van Software design principes	
			Register	
Werkboek	2	4 JavaScript voor het web	8 jQuery en de DOM	
			9 jQuery en events	
			10 Dynamische webapplicaties: client en server	
			11 Form-validatie	
		5 Software engineering	12 Software engineering	
			Lijst van Programmeeraanwijzingen	
			Lijst van Software design principes	
			Register	
Reader	De reader bevat een aantal hoofdstukken van de Engelstalige website Eloquent JavaScript van Marijn Haverbeke (2014):			
	Chapter 1 Introduction			7
	Chapter 2 Basic JavaScript: values, variables, and control flow			13
	Chapter 3 Functions			31
	Chapter 4 Data structures: Objects and Arrays			43
	Chapter 5 Error Handling			63
	Index			68
Cursusweb	http://studienet.ou.nl			
	Eindtoets, bouwstenen voor opdrachten, weblinks, informatie over begeleiding en toetsing, nieuws, errata, informatie over gebruikte tools, discussiegroep, schatting studielast per leereenheid.			

Chapter 1

Introduction

When personal computers were first introduced, most of them came equipped with a simple programming language, usually a variant of BASIC. Interacting with the computer was closely integrated with this language, and thus every computer-user, whether he wanted to or not, would get a taste of it. Now that computers have become plentiful and cheap, typical users don't get much further than clicking things with a mouse. For most people, this works very well. But for those of us with a natural inclination towards technological tinkering, the removal of programming from every-day computer use presents something of a barrier.

Fortunately, as an effect of developments in the World Wide Web, it so happens that every computer equipped with a modern web-browser also has an environment for programming JavaScript. In today's spirit of not bothering the user with technical details, it is kept well hidden, but a web-page can make it accessible, and use it as a platform for learning to program.

That is what this (hyper-)book tries to do.

> I do not enlighten those who are not eager to learn, nor arouse those who are not anxious to give an explanation themselves. If I have presented one corner of the square and they cannot come back to me with the other three, I should not go over the points again.
> *Confucius*

1 Programming

Principles of programming

Besides explaining JavaScript, this book tries to be an introduction to the basic *principles of programming*. Programming, it turns out, is hard. The fundamental rules are, most of the time, simple and clear. But programs, while built on top of these basic rules, tend to become complex enough to introduce their own rules, their own complexity. Because of this, programming is rarely simple or predictable. As Donald Knuth, who is something of a founding father of the field, says, it is an art.

To get something out of this book, more than just passive reading is required. Try to stay sharp, make an effort to solve the exercises, and only continue on when you are reasonably sure you understand the material that came before. The computer programmer is a creator of universes for which he alone is responsible.

> Universes of virtually unlimited complexity can be created in the form of computer programs.
> *Joseph Weizenbaum, Computer Power and Human Reason*

A program is many things. It is a piece of text typed by a programmer, it is the directing force that makes the computer do what it does, it is data in the computer's memory, yet it controls the actions performed on this same memory. Analogies that try to compare programs to objects we are familiar with tend to fall short, but a superficially fitting one is that of a machine. The gears of a mechanical watch fit together ingeniously, and if the watchmaker was any good, it will accurately show the time for many years. The elements of a program fit together in a similar way, and if the programmer knows what he is doing, the program will run without crashing.

A computer is a machine built to act as a host for these immaterial machines. Computers themselves can only do stupidly straightforward things. The reason they are so useful is that they do these things at an incredibly high speed. A program can, by ingeniously combining many of these simple actions, do very complicated things.

To some of us, writing computer programs is a fascinating game. A program is a building of thought. It is costless to build, weightless, growing easily under our typing hands. If we get carried away, its size and complexity will grow out of control, confusing even the one who created it. This is the main problem of programming. It is why so much of today's software tends to crash, fail, screw up.

When a program works, it is beautiful. The art of programming is the skill of controlling complexity. The great program is subdued, made simple in its complexity. Today, many programmers believe that this complexity is best managed by using only a small set of well-understood techniques in their programs. They have composed strict rules about the form programs should have, and the more zealous among them will denounce those who break these rules as bad programmers.

What hostility to the richness of programming! To try to reduce it to something straightforward and predictable, to place a taboo on all the weird and beautiful programs. The landscape of programming techniques is enormous, fascinating in its diversity, still largely unexplored. It is certainly littered with traps and snares, luring the inexperienced programmer into all kinds of horrible mistakes, but that only means you should proceed with caution, keep your wits about you. As you learn, there will always be new challenges, new territory to explore. The programmer who refuses to keep exploring will surely stagnate, forget his joy, lose the will to program (and become a manager).

As far as I am concerned, the definite criterion for a program is whether it is correct. Efficiency, clarity, and size are also important, but how to balance these against each other is always a matter of judgement, a judgement that each programmer must make for himself. Rules of thumb are useful, but one should never be afraid to break them.

2 Programming languages

In the beginning, at the birth of computing, there were no programming languages. Programs looked something like this:.

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

That is a program to add the numbers from one to ten together, and print out the result `(1 + 2 + ... + 10 = 55)`. It could run on a very simple kind of computer. To program early computers, it was necessary to set large arrays of switches in the right position, or punch holes in strips of cardboard and feed them to the computer. You can imagine how this was a tedious, error-prone procedure. Even the writing of simple programs required much cleverness and discipline, complex ones were nearly inconceivable.

Of course, manually entering these arcane patterns of bits (which is what the 1s and 0s above are generally called) did give the programmer a profound sense of being a mighty wizard. And that has to be worth something, in terms of job satisfaction. Each line of the program contains a single instruction. It could be written in English like this:

a Store the number `0` in memory location `0`

b Store the number `1` in memory location `1`

c Store the value of memory location `1` in memory location `2`

d Decrement the value in memory location `2` by the number `11`

e If the value in memory location `2` is the number `0`, continue with instruction `9`

f Increment the value in memory location `0` by the value in memory location `1`

g Increment the value in memory location `1` by the number `1`

h Continue with instruction `3`

i Output the value of memory location `0`

While that is more readable than the binary soup, it is still rather unpleasant. It might help to use names instead of numbers for the instructions and memory locations:

```
Set 'total' to 0
Set 'count' to 1 [loop]
Set 'compare' to 'count'
Subtract 11 from 'compare'
If 'compare' is zero, continue at [end]
Add 'count' to 'total'
Add 1 to 'count'
Continue at [loop] [end]
Output 'total'
```

At this point it is not too hard to see how the program works. Can you? The first two lines give two memory locations their starting values: total will be used to build up the result of the program, and count keeps track of the number that we are currently looking at. The lines using compare are probably the weirdest ones. What the program wants to do is see if count is equal to `11`, in order to decide whether it can stop yet. Because the machine is so primitive, it can only test whether a number is zero, and make a decision (jump) based on that. So it uses the memory location labelled compare to compute the value of count `-11`, and makes a decision based on that value. The next two lines add the value of count to the result, and increment count by one every time the program has decided that it is not `11` yet.

Here is the same program in JavaScript:

```
var total = 0, count = 1;
while (count <= 10) {
  total += count; count += 1;
}
print(total);
```

This gives us a few more improvements. Most importantly, there is no need to specify the way we want the program to jump back and forth anymore. The magic word while takes care of that. It continues executing the lines below it as long as the condition it was given holds: `count <= 10`, which means "count is less than or equal to

10". Apparently, there is no need anymore to create a temporary value and compare that to zero. This was a stupid little detail, and the power of programming languages is that they take care of stupid little details for us.

Finally, here is what the program could look like if we happened to have the convenient operations range and sum available, which respectively create a collection of numbers within a range and compute the sum of a collection of numbers:

```
print(sum(range(1, 10)));
```

The moral of this story, then, is that the same program can be expressed in long and short, unreadable and readable ways. The first version of the program was extremely obscure, while this last one is almost English: print the sum of the range of numbers from `1` to `10`. (We will see in later chapters how to build things like sum and range.)

A good programming language helps the programmer by providing a more abstract way to express himself. It hides uninteresting details, provides convenient building blocks (such as the while construct), and, most of the time, allows the programmer to add building blocks himself (such as the sum and range operations).

3 JavaScript

JavaScript

JavaScript is the language that is, at the moment, mostly being used to do all kinds of clever and horrible things with pages on the World Wide Web. Some people claim that the next version of JavaScript will become an important language for other tasks too. I am unsure whether that will happen, but if you are interested in programming, JavaScript is definitely a useful language to learn. Even if you do not end up doing much web programming, the mind-bending programs I will show you in this book will always stay with you, haunt you, and influence the programs you write in other languages.

There are those who will say terrible things about JavaScript. Many of these things are true. When I was for the first time required to write something in JavaScript, I quickly came to despise the language. It would accept almost anything I typed, but interpret it in a way that was completely different from what I meant. This had a lot to do with the fact that I did not have a clue what I was doing, but there is also a real issue here: JavaScript is ridiculously liberal in what it allows. The idea behind this design was that it would make programming in JavaScript easier for beginners. In actuality, it mostly makes finding problems in your programs harder, because the system will not point them out to you.

However, the flexibility of the language is also an advantage. It leaves space for a lot of techniques that are impossible in more rigid languages, and it can be used to overcome some of JavaScript's shortcomings. After learning it properly, and working with it for a while, I have really learned to like this language.

Contrary to what the name suggests, JavaScript has very little to do with the programming language named Java. The similar name was inspired by marketing considerations, rather than good judgement. In 1995, when JavaScript was introduced by Netscape, the Java language was being heavily marketed and gaining in popularity. Apparently, someone thought it a good idea to try and ride along on this marketing. Now we are stuck with the name.

ECMAScript

Related to JavaScript is a thing called *ECMAScript*. When browsers other than Netscape started to support JavaScript, or something that looked like it, a document was written to describe precisely how the language should work. The language described in this document is called ECMAScript, after the organisation that standardised it.

ECMAScript describes a general-purpose programming language, and does not say anything about the integration of this language in an Internet browser. JavaScript is ECMAScript plus extra tools for dealing with Internet pages and browser windows. A few other pieces of software use the language described in the ECMAScript document. Most importantly, the ActionScript language used by Flash is based on ECMAScript (though it does not precisely follow the standard). Flash is a system for adding things that move and make lots of noise to web-pages. Knowing JavaScript won't hurt if you ever find yourself learning to build Flash movies.

JavaScript is still evolving. Since this book came out, ECMAScript 5 has been released, which is compatible with the version described here, but adds some of the functionality we will be writing ourselves as built-in methods. The newest generation of browsers provides this expanded version of JavaScript. As of 2011, "ECMAScript harmony", a more radical extension of the language, is in the process of being standardised. You should not worry too much about these new versions making the things you learn from this book obsolete. For one thing, they will be an extension of the language we have now, so almost everything written in this book will still hold.

Most chapters in this book contain quite a lot of code. In my experience, reading and writing code is an important part of learning to program. Try to not just glance over these examples, but read them attentively and understand them. This can be slow and confusing at first, but you will quickly get the hang of it. The same goes for the exercises. Don't assume you understand them until you've actually written a working solution.

Because of the way the web works, it is always possible to look at the JavaScript programs that people put in their web-pages. This can be a good way to learn how some things are done. Because most web programmers are not "professional" programmers, or consider JavaScript programming so uninteresting that they never properly learned it, a lot of the code you can find like this is of a very bad quality. When learning from ugly or incorrect code, the ugliness and confusion will propagate into your own code, so be careful who you learn from.

Basic JavaScript: values, variables, and control flow

Chapter 2

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 bits1, 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.

1 Numbers

Numeric values

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

```
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:

```
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. 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

Fractional numbers are written by using a dot.

```
9.81
```

Scientific notation

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`.

Integers

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.

1.1 ARYTHMETIC OPERATIONS

Arithmetic operations

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. This is what they look like in JavaScript:

```
100 + 4 * 11
```

Operator

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 expression 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 the remainder 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.

2 Strings

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 :

```
"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.

```
"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\hnb")`. 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\"."
```

Concatenate

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"
```

3 Operators

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
```

Binary operator
Unary operator

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)
```

4 Booleans

Boolean

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.

Logical and

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
```

Logical or

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

```
true || false
```

Not

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`.

EXERCISE 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)
```

I hope you noticed that `"grass" != "green"` is `true`. Grass may be green, but it is not equal to green.

Precedence

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.

5 Variables

Expression

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.

Statement
Program

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;
```

Side effect

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. This leaves no impression on the world at all, and is not a side effect.

Variable

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.

var

The keyword *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;
```

Environment

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.

6 Functions

Function

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.

6.1 SOME USEFUL 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.

7 Control statements

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));
```

Number

Boolean

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.

7.1 WHILE

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;
}
```

Loop

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`.

Block

EXERCISE 2.2

Use the techniques shown so far to write a program that calculates and shows the value of 2^{10} (`2` to the `10`th 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.

EXERCISE 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.

Indentation

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 the tab key will re-indent the line your cursor is currently on.

Semicolon

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.

7.2 FOR

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.

Constructor

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.

Keywords

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.

EXERCISE 2.4

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

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-`.

7.3 IF

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.

EXERCISE 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.

7.4 BREAK

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 `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);
```

Empty statement

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.

EXERCISE 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.

8 Special values

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);
```

undefined

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."));
```

null

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`.

9 Comments

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);
```

10 More on values

NaN

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.

isNaN

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`.

```
false || alert("I'm happening!");
true || alert("Not me.");
```

FEEDBACK

Answers to the exercises

2.1 Yes, it is true. You can reduce it step by step like this:

```
(false || true) && !(false && true)
true && !false
true
```

2.2

```
var result = 1;
var counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
show(result);
```

The counter could also start at 1 and check for <= 10, but, for reasons that will become apparent later on, it is a good idea to get used to counting from `0`. Obviously, your own solutions aren't required to be precisely the same as mine. They should work. And if they are very different, make sure you also understand my solution.

2.3

```
var line = "";
var counter = 0;
while (counter < 10) {
  line = line + "#";
  print(line);
  counter = counter + 1;
}
```

2.4

```
var result = 1;
for (var counter = 0; counter < 10; counter = counter + 1)
  result = result * 2;
show(result);
}
```

Note that even if no block is opened with a '`{`', the statement in the loop is still indented two spaces to make it clear that it 'belongs' to the line above it.

```
var line = "";
for (var counter = 0; counter < 10; counter = counter + 1) {
  line = line + "#";
  print(line);
}
```

2.5

```
var answer = prompt("You!  What is the value of 2 + 2?", "");
if (answer == "4")
  alert("You must be a genius or something.");
else if (answer == "3" || answer == "5")
  alert("Almost!");
else alert("You're an embarrassment.");
```

2.6

```
var answer;
while (true) {
  answer = prompt("You!  What is the value of 2 + 2?", "");
  if (answer == "4") {
    alert("You must be a genius or something.");
    break;
  }
  else if (answer == "3" || answer == "5") {
    alert("Almost!");
  }
  else {
    alert("You're an embarrassment.");
  }
}
```

Because the first `if`'s body now has two statements, I added braces around all the bodies. This is a matter of taste. Having an `if`/`else` chain where some of the bodies are blocks and others are single statements looks a bit lopsided to me, but you can make up your own mind about that.

Another solution, arguably nicer, but without `break`:

```
var value = null;
while (value != "4") {
  value = prompt("You!  What is the value of 2 + 2?", "");
  if (value == "4")
    alert("You must be a genius or something.");
  else if (value == "3" || value == "5")
    alert("Almost!");
  else
    alert("You're an embarrassment.");
}
```

Functions

Chapter 3

Functions

A program often needs to do the same thing in different places. Repeating all the necessary statements every time is tedious and error-prone. It would be better to put them in one place, and have the program take a detour through there whenever necessary. This is what functions were invented for: They are canned code that a program can go through whenever it wants. Putting a string on the screen requires quite a few statements, but when we have a print function we can just write print("Aleph") and be done with it.

To view functions merely as canned chunks of code doesn't do them justice though. When needed, they can play the role of pure functions, algorithms, indirections, abstractions, decisions, modules, continuations, data structures, and more. Being able to effectively use functions is a necessary skill for any kind of serious programming. This chapter provides an introduction into the subject, chapter 6 discusses the subtleties of functions in more depth.

1 Pure functions

Pure functions, for a start, are the things that were called functions in the mathematics classes that I hope you have been subjected to at some point in your life. Taking the cosine or the absolute value of a number is a pure function of one argument. Addition is a pure function of two arguments.

The defining properties of pure functions are that they always return the same value when given the same arguments, and never have side effects. They take some arguments, return a value based on these arguments, and do not monkey around with anything else.

In JavaScript, addition is an operator, but it could be wrapped in a function like this (and as pointless as this looks, we will come across situations where it is actually useful):

```
function add(a, b) {
  return a + b;
}
show(add(2, 2));
```

Argument
Body

`add` is the name of the function. `a` and `b` are the names of the two *arguments*. `return a + b;` is the *body* of the function.

The keyword `function` is always used when creating a new function. When it is followed by a variable name, the resulting function will be stored under this name. After the name comes a list of argument names, and then finally the body of the function. Unlike those around the body of while loops or if statements, the braces around a function body are obligatory.

return

The keyword *`return`*, followed by an expression, is used to determine the value the function returns. When control comes across a return statement, it immediately jumps out of the current function and gives the returned value to the code that called the function. A return statement without an expression after it will cause the function to return `undefined`.

A body can, of course, have more than one statement in it. Here is a function for computing powers (with positive, integer exponents):

```
function power(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}
show(power(2, 10));
```

If you solved exercise 2.2, this technique for computing a power should look familiar.

Creating a variable (`result`) and updating it are side effects. Didn't I just say pure functions had no side effects?

Side effect

A variable created inside a function exists only inside the function. This is fortunate, or a programmer would have to come up with a different name for every variable he needs throughout a program. Because result only exists inside power, the changes to it only last until the function returns, and from the perspective of code that calls it there are no *side effects*.

EXERCISE 3.1

Write a function called `absolute`, which returns the absolute value of the number it is given as its argument. The absolute value of a negative number is the positive version of that same number, and the absolute value of a positive number (or zero) is that number itself.

Pure functions have two very nice properties. They are easy to think about, and they are easy to re-use.

If a function is pure, a call to it can be seen as a thing in itself. When you are not sure that it is working correctly, you can test it by calling it directly from the console, which is simple because it does not depend on any context. It is easy to make these tests automatic - to write a program that tests a specific function. Non-pure functions might return different values based on all kinds of factors, and have side effects that might be hard to test and think about.

Because pure functions are self-sufficient, they are likely to be useful and relevant in a wider range of situations than non-pure ones. Take `show`, for example. This function's usefulness depends on the presence of a special place on the screen for printing output. If that place is not there, the function is useless. We can imagine a related function, let's call it `format`, that takes a value as an argument and returns a string that represents this value. This function is useful in more situations than `show`.

Of course, `format` does not solve the same problem as `show`, and no pure function is going to be able to solve that problem, because it requires a side effect. In many cases, non-pure functions are precisely what you need. In other cases, a problem can be solved with a pure function but the non-pure variant is much more convenient or efficient.

Thus, when something can easily be expressed as a pure function, write it that way. But never feel dirty for writing non-pure functions.

Functions with side effects do not have to contain a `return` statement. If no return statement is encountered, the function returns `undefined`.

```
function yell(message) {
  alert(message + "!!");
}
yell("Yow");
```

2 Local environment

The names of the arguments of a function are available as variables inside it. They will refer to the values of the arguments the function is being called with, and like normal variables created inside a function, they do not exist outside it. Aside from the top-level environment, there are smaller, local environments created by function calls. When looking up a variable inside a function, the local environment is checked first, and only if the variable does not exist there is it looked up in the top-level environment. This makes it possible for variables inside a function to "shadow" top-level variables that have the same name.

```
function alertIsPrint(value) {
  var alert = print;
  alert(value);
}
alertIsPrint("Troglodites");
```

The variables in this local environment are only visible to the code inside the function. If this function calls another function, the newly called function does not see the variables inside the first function:

```
var variable = "top-level";
function printVariable() {
  print("inside printVariable, the variable holds '" + variable + "'.");
}
function test() {
  var variable = "local";
  print("inside test, the variable holds '" + variable + "'.");
  printVariable();
}
test();
```

2.1 INNER FUNCTIONS

However, and this is a subtle but extremely useful phenomenon, when a function is defined *inside* another function, its local environment will be based on the local environment that surrounds it instead of the top-level environment.

```
var variable = "top-level";
function parentFunction() {
  var variable = "local";
  function childFunction() {
    print(variable);
  }
  childFunction();
}
parentFunction();
```

What this comes down to is that which variables are visible inside a function is determined by the place of that function in the program text. All variables that were defined "above" a function's definition are visible, which means both those in function bodies that enclose it, and those at the top-level of the program. This approach to variable visibility is called *lexical scoping*.

Lexical scoping

2.2 BLOCK OF CODE: NO NEW ENVIRONMENT

People who have experience with other programming languages might expect that a block of code (between braces) also produces a new local environment. Not in JavaScript. Functions are the only things that create a new scope. You are allowed to use free-standing blocks like this...

```
var something = 1;
{
  var something = 2;
  print("Inside: " + something);
}
print("Outside: " + something);
```

... but the something inside the block refers to the same variable as the one outside the block. In fact, although blocks like this are allowed, they are utterly pointless. Most people agree that this is a bit of a design blunder by the designers of JavaScript, and ECMAScript Harmony will add some way to define variables that stay inside blocks (the `let` keyword).

3 Closure

Here is a case that might surprise you:

```
var variable = "top-level";
function parentFunction() {
  var variable = "local";
  function childFunction() {
    print(variable);
  }
  return childFunction;
}
var child = parentFunction();
child();
```

`parentFunction` returns its internal function, and the code at the bottom calls this function. Even though `parentFunction` has finished executing at this point, the local environment where `variable` has the value `"local"` still exists, and `childFunction` still uses it. This phenomenon is called *closure*.

Closure

Apart from making it very easy to quickly see in which part of a program a variable will be available by looking at the shape of the program text, lexical scoping also allows us to "synthesise" functions. By using some of the variables from an enclosing function, an inner function can be made to do different things. Imagine we need a few different but similar functions, one that adds `2` to its argument, one that adds `5`, and so on.

```
function makeAddFunction(amount) {
  function add(number) {
    return number + amount;
  }
  return add;
}
```

```
var addTwo = makeAddFunction(2);
var addFive = makeAddFunction(5);
show(addTwo(1) + addFive(1));
```

To wrap your head around this, you should consider functions to not just package up a computation, but also an environment. Top-level functions simply execute in the top-level environment, that much is obvious. But a function defined inside another function retains access to the environment that existed in that function at the point when it was defined.

Thus, the `add` function in the above example, which is created when `makeAddFunction` is called, captures an environment in which amount has a certain value. It packages this environment, together with the computation `return number + amount`, into a value, which is then returned from the outer function.

When this returned function (`addTwo` or `addFive`) is called, a new environment (in which the variable number has a value) is created, as a sub-environment of the captured environment (in which amount has a value). These two values are then added, and the result is returned.

4 Recursion

Recursive

On top of the fact that different functions can contain variables of the same name without getting tangled up, these scoping rules also allow functions to call *themselves* without running into problems. A function that calls itself is called *recursive*. Recursion allows for some interesting definitions. Look at this implementation of `power`:

```
function power(base, exponent) {
  if (exponent == 0)
    return 1;
  else
    return base * power(base, exponent -1);
}
```

This is rather close to the way mathematicians define exponentiation, and to me it looks a lot nicer than the earlier version. It sort of loops, but there is no `while`, `for`, or even a local side effect to be seen. By calling itself, the function produces the same effect.

There is one important problem though: In most browsers, this second version is about ten times slower than the first one. In JavaScript, running through a simple loop is a lot cheaper than calling a function multiple times.

The dilemma of speed versus elegance is an interesting one. It not only occurs when deciding for or against recursion. In many situations, an elegant, intuitive, and often short solution can be replaced by a more convoluted but faster solution.

In the case of the `power` function above the un-elegant version is still sufficiently simple and easy to read. It doesn't make very much sense to replace it with the recursive version. Often, though, the concepts a program is dealing with get so complex that giving up some efficiency in order to make the program more straightforward becomes an attractive choice.

The basic rule, which has been repeated by many programmers and with which I wholeheartedly agree, is to not worry about efficiency until your program is probably too slow. When it is, find out which parts are too slow, and start exchanging elegance for efficiency in those parts.

Of course, the above rule doesn't mean one should start ignoring performance altogether. In many cases, like the `power` function, not much simplicity is gained by the "elegant" approach. In other cases, an experienced programmer can see right away that a simple approach is never going to be fast enough.

Efficiency

The reason I am making a big deal out of this is that surprisingly many programmers focus fanatically on *efficiency*, even in the smallest details. The result is bigger, more complicated, and often less correct programs, which take longer to write than their more straightforward equivalents and often run only marginally faster.

4.1 PROGRAM STACK

Stack

But I was talking about recursion. A concept closely related to recursion is a thing called the stack. When a function is called, control is given to the body of that function. When that body returns, the code that called the function is resumed. While the body is running, the computer must remember the context from which the function was called, so that it knows where to continue afterwards. The place where this context is stored is called the *stack*.

The fact that it is called "stack" has to do with the fact that, as we saw, a function body can again call a function. Every time a function is called, another context has to be stored. One can visualise this as a stack of contexts. Every time a function is called, the current context is thrown on top of the stack. When a function returns, the context on top is taken off the stack and resumed.

This stack requires space in the computer's memory to be stored. When the stack grows too big, the computer will give up with a message like"out of stack space" or "too much recursion". This is something that has to be kept in mind when writing recursive functions.

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
print(chicken() + " came first.");
```

4.2 INDIRECT RECURSION

In addition to demonstrating a very interesting way of writing a broken program, this example shows that a function does not have to call itself directly to be recursive. If it calls another function which (directly or indirectly) calls the first function again, it is still recursive.

Recursion is not always just a less-efficient alternative to looping. Some problems are much easier to solve with recursion than with loops. Most often these are problems that require exploring or processing several "branches", each of which might branch out again into more branches.

Consider this puzzle: By starting from the number `1` and repeatedly either adding `5` or multiplying by `3`, an infinite amount of new numbers can be produced. How would you write a function that, given a number, tries to find a sequence of additions and multiplications that produce that number?

For example, the number 13 could be reached by first multiplying 1 by 3, and then adding 5 twice. The number 15 can not be reached at all.

Here is the solution:

```
function findSequence(goal) {
  function find(start, history) {
    if (start == goal)
      return history;
    else if (start > goal)
      return null;
    else
      return find(start + 5, "(" + history + " + 5)") ||
        find(start * 3, "(" + history + " * 3)");
  }
  return find(1, "1");
}
print(findSequence(24));
```

Note that it doesn't necessarily find the shortest sequence of operations, it is satisfied when it finds any sequence at all.

The inner `find` function, by calling itself in two different ways, explores both the possibility of adding 5 to the current number and of multiplying it by 3. When it finds the number, it returns the `history` string, which is used to record all the operators that were performed to get to this number. It also checks whether the current number is bigger than `goal`, because if it is, we should stop exploring this branch, it is not going to give us our number.

The use of the `||` operator in the example can be read as “return the solution found by adding 5 to start, and if that fails, return the solution found by multiplying start by 3”. It could also have been written in a more wordy way like this:

```
else {
  var found = find(start + 5, "(" + history + " + 5)");
  if (found == null)
    found = find(start * 3, "(" + history + " * 3)");
  return found;
}
```

5 Timeline for function definitions

Even though function definitions occur as statements between the rest of the program, they are not part of the same time-line:

```
print("The future says: ", future());
function future() {
  return "We STILL have no flying cars.";
}
```

What is happening is that the computer looks up all function definitions, and stores the associated functions, before it starts executing the rest of the program. The same happens with functions that are defined inside other functions. When the outer function is called, the first thing that happens is that all inner functions are added to the new environment.

6 Anonymous functions

There is another way to define function values, which more closely resembles the way other values are created. When the `function` keyword is used in a place where an expression is expected, it is treated as an expression producing a function value. Functions created in this way do not have to be given a name (though it is allowed to give them one).

```
var add = function(a, b) {
  return a + b;
};
show(add(5, 5));
```

Note the semicolon after the definition of `add`. Normal function definitions do not need these, but this statement has the same general structure as `var add = 22;`, and thus requires the semicolon.

Anonymous function

This kind of function value is called an *anonymous function*, because it does not have a name. Sometimes it is useless to give a function a name, like in the `makeAddFunction` example we saw earlier:

```
function makeAddFunction(amount) {
  return function (number) {
    return number + amount;
  };
}
```

Since the function named `add` in the first version of `makeAddFunction` was referred to only once, the name does not serve any purpose and we might as well directly return the function value.

EXERCISE 3.2

Write a function `greaterThan`, which takes one argument, a number, and returns a function that represents a test. When this returned function is called with a single number as argument, it returns a boolean: `true` if the given number is greater than the number that was used to create the test function, and `false` otherwise.

7 The number of arguments

Try the following:

```
alert("Hello", "
Good Evening", "How do you do?", "Goodbye");
```

The function `alert` officially only accepts one argument. Yet when you call it like this, the computer does not complain at all, but just ignores the other arguments.

```
show();
```

You can, apparently, even get away with passing too few arguments. When an argument is not passed, its value inside the function is undefined.

In the next chapter, we will see a way in which a function body can get at the exact list of arguments that were passed to it. This can be useful, as it makes it possible to have a function accept any number of arguments. print makes use of this:

```
print("R", 2, "D", 2);
```

Of course, the downside of this is that it is also possible to accidentally pass the wrong number of arguments to functions that expect a fixed amount of them, like `alert`, and never be told about it.

FEEDBACK

Answers to the exercises

3.1
```
function absolute(number) {
  if (number < 0)
    return -number;
  else
    return number; } show(absolute(-144));
```

3.2
```
function greaterThan(x) {
  return function(y) {
    return y > x;
  };
}
var greaterThanTen = greaterThan(10);
show(greaterThanTen(9));
```

Data structures: Objects and Arrays

Chapter 4

Data structures: Objects and Arrays

This chapter will be devoted to solving a few simple problems. In the process, we will discuss two new types of values: arrays and objects, and look at some techniques related to them.

Consider the following situation: Your crazy aunt Emily, who is rumoured to have over fifty cats living with her (you never managed to count them), regularly sends you e-mails to keep you up to date on her exploits. They usually look like this:

> Dear nephew,
>
> Your mother told me you have taken up skydiving. Is this true? You watch yourself, young man! Remember what happened to my husband? And that was only from the second floor!
>
> Anyway, things are very exciting here. I have spent all week trying to get the attention of Mr. Drake, the nice gentleman who moved in next door, but I think he is afraid of cats. Or allergic to them? I am going to try putting Fat Igor on his shoulder next time I see him, very curious what will happen.
>
> Also, the scam I told you about is going better than expected. I have already gotten back five "payments", and only one complaint. It is starting to make me feel a bit bad though. And you are right that it is probably illegal in some way.
>
> (... etc ...)
>
> Much love, Aunt Emily
>
> died 27/04/2006: Black Leclère
>
> born 05/04/2006 (mother Lady Penelope): Red Lion, Doctor Hobbles the 3rd, Little Iroquois

To humour the old dear, you would like to keep track of the genealogy of her cats, so you can add things like"P.S. I hope Doctor Hobbles the 2nd enjoyed his birthday this Saturday!", or "How is old Lady Penelope doing? She's five years old now, isn't she?", preferably without accidentally asking about dead cats. You are in the possession of a large quantity of old e-mails from your aunt, and fortunately she is very consistent in always putting information about the cats' births and deaths at the end of her mails in precisely the same format.

You are hardly inclined to go through all those mails by hand. Fortunately, we were just in need of an example problem, so we will try to work out a program that does the work for us. For a start, we write a program that gives us a list of cats that are still alive after the last e-mail.

Before you ask, at the start of the correspondence, aunt Emily had only a single cat: Spot. (She was still rather conventional in those days.)

It usually pays to have some kind of clue what one's program is going to do before starting to type. Here's a plan:

a Start with a set of cat names that has only"Spo" in it.

b Go over every e-mail in our archive, in chronological order.

c Look for paragraphs that start with "born" or "died".

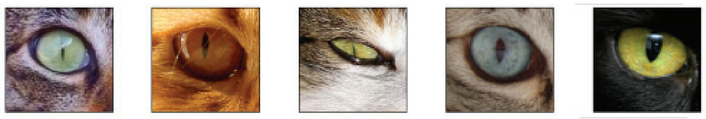

FIGURE 4.1 The eyes of the cats

d Add the names from paragraphs that start with "born" to our set of names.

e Remove the names from paragraphs that start with "died" from our set.

Where taking the names from a paragraph goes like this:

a Find the colon in the paragraph.

b Take the part after this colon.

c Split this part into separate names by looking for commas.

It may require some suspension of disbelief to accept that aunt Emily always used this exact format, and that she never forgot or misspelled a name, but that is just how your aunt is.

1 Properties

Properties

First, let me tell you about properties. A lot of JavaScript values have other values associated with them. These associations are called *properties*. Every string has a property called `length`, which refers to a number, the amount of characters in that string.

Properties can be accessed in two ways:

```
var text = "purple haze";
show(text["length"]);
show(text.length);
```

The second way is a shorthand for the first, and it only works when the name of the property would be a valid variable name - when it doesn't have any spaces or symbols in it and does not start with a digit character.

The values `null` and `undefined` do not have any properties. Trying to read properties from such a value produces an error. Try the following code, if only to get an idea about the kind of error-message your browser produces in such a case (which, for some browsers, can be rather cryptic).

```
var nothing = null;
show(nothing.length);
```

The properties of a string value can not be changed. There are quite a few more than just `length`, as we will see, but you are not allowed to add or remove any.

2 Objects

Object

This is different with values of the type *object*. Their main role is to hold other values. They have, you could say, their own set of tentacles in the form of properties. You are free to modify these, remove them, or add new ones.

An object can be written like this:

```
var cat = {colour: "grey", name: "Spot", size: 46};
cat.size = 47;
show(cat.size);
delete cat.size;
show(cat.size);
show(cat);
```

Like variables, each property attached to an object is labelled by a string. The first statement creates an object in which the property `colour` holds the string `"grey"`, the property `name` is attached to the string `"Spot"`, and the property `size` refers to the number `46`. The second statement gives the property named `size` a new value, which is done in the same way as modifying a variable.

delete

The keyword *`delete`* cuts off properties. Trying to read a non-existent property gives the value `undefined`. If a property that does not yet exist is set with the `=` operator, it is added to the object.

```
var empty = {};
empty.notReally = 1000;
show(empty.notReally);
```

Properties whose names are not valid variable names have to be quoted when creating the object, and approached using brackets:

```
var thing = {"gabba gabba": "hey", "5": 10};
show(thing["5"]);
thing["5"] = 20;
show(thing[2 + 3]);
delete thing["gabba gabba"];
```

As you can see, the part between the brackets can be any expression. It is converted to a string to determine the property name it refers to. One can even use variables to name properties:

```
var propertyName = "length";
var text = "mainline";
show(text[propertyName]);
```

in

The operator *`in`* can be used to test whether an object has a certain property. It produces a boolean.

```
var chineseBox = {};
chineseBox.content = chineseBox;
show("content" in chineseBox);
show("content" in chineseBox.content);
```

When object values are shown on the console, they can be clicked to inspect their properties. This changes the output window to an "inspect" window. The little "x" at the top-right can be used to return to the output window, and the left-arrow can be used to go back to the properties of the previously inspected object.

```
show(chineseBox);
```

EXERCISE 4.1

The solution for the cat problem talks about a "set" of names. A set is a collection of values in which no value may occur more than once. If names are strings, can you think of a way to use an object to represent a set of names?

Show how a name can be added to this set, how one can be removed, and how you can check whether a name occurs in it.

Object values, apparently, can change. The types of values discussed in chapter 2 are all immutable, it is impossible to change an existing value of those types. You can combine them and derive new values from them, but when you take a specific string value, the text inside it can not change. With objects, on the other hand, the content of a value can be modified by changing its properties.

2.1 VALUES OF OBJECTS

When we have two numbers, `120` and `120`, they can for all practical purposes be considered the precise same number. With objects, there is a difference between having two references to the same object and having two different objects that contain the same properties. Consider the following code:

```
var object1 = {value: 10};
var object2 = object1;
var object3 = {value: 10};
show(object1 == object2);
show(object1 == object3);
object1.value = 15;
show(object2.value);
show(object3.value);
```

`object1` and `object2` are two variables grasping the same value. There is only one actual object, which is why changing `object1` also changes the value of `object2`. The variable `object3` points to another object, which initially contains the same properties as `object1`, but lives a separate life.

JavaScript's == operator, when comparing objects, will only return `true` if both values given to it are the precise same value. Comparing different objects with identical contents will give `false`. This is useful in some situations, but impractical in others.

3 Arrays

Object values can play a lot of different roles. Behaving like a set is only one of those. We will see a few other roles in this chapter, and chapter 8 shows another important way of using objects.

In the plan for the cat problem - in fact, call it an algorithm, not a plan, that makes it sound like we know what we are talking about - in the algorithm, it talks about going over all the e-mails in an archive. What does this archive look like? And where does it come from?

Do not worry about the second question for now. Chapter 14 talks about some ways to import data into your programs, but for now you will find that the e-mails are just magically there. Some magic is really easy, inside computers.

The way in which the archive is stored is still an interesting question. It contains a number of e-mails. An e-mail can be a string, that should be obvious. The whole archive could be put into one huge string, but that is hardly practical. What we want is a collection of separate strings.

Collections of things are what objects are used for. One could make an object like this:

```
var mailArchive = {"the first e-mail": "Dear nephew, ...",
  "the second e-mail": "..."
  /* and so on ... */};
```

But that makes it hard to go over the e-mails from start to end - how does the program guess the name of these properties? This can be solved by more predictable property names:

```
var mailArchive = {0: "Dear nephew, ... (mail number 1)",
  1: "(mail number 2)",
  2: "(mail number 3)"};
for (var current = 0; current in mailArchive; current++)
  print("Processing e-mail #", current, ": ", mailArchive[current]);
```

arrays

Luck has it that there is a special kind of objects specifically for this kind of use. They are called *arrays*, and they provide some conveniences, such as a `length` property that contains the amount of values in the array, and a number of operations useful for this kind of collection.

New arrays can be created using brackets ([and]):

```
var mailArchive = ["mail one", "mail two", "mail three"];
for (var current = 0; current < mailArchive.length; current++)
  print("Processing e-mail #", current, ": ", mailArchive[current]);
```

In this example, the numbers of the elements are not specified explicitly anymore. The first one automatically gets the number `0`, the second the number `1`, and so on.

Why start at `0`? People tend to start counting from `1`. As unintuitive as it seems, numbering the elements in a collection from `0` is often more practical. Just go with it for now, it will grow on you.

Starting at element `0` also means that in a collection with `X` elements, the last element can be found at position `X`-1. This is why the for loop in the example checks for `current < mailArchive.length`. There is no element at position `mailArchive.length`, so as soon as current has that value, we stop looping.

EXERCISE 4.2

Write a function range that takes one argument, a positive number, and returns an array containing all numbers from `0` up to and including the given number.

An empty array can be created by simply typing `[]`. Also remember that adding properties to an object, and thus also to an array, can be done by assigning them a value with the = operator. The length property is automatically updated when elements are added.

4 Properties and methods of strings and arays

Both string and array objects contain, in addition to the `length` property, a number of properties that refer to function values.

```
var doh = "Doh";
print(typeof doh.toUpperCase);
print(doh.toUpperCase());
```

Every string has a toUpperCase property. When called, it will return a copy of the string, in which all letters have been converted to uppercase. There is also toLowerCase. Guess what that does.

Notice that, even though the call to toUpperCase does not pass any arguments, the function does somehow have access to the string "Doh", the value of which it is a property. How this works precisely is described in chapter 8.

Methods

Properties that contain functions are generally called *methods*, as in "toUpperCase is a method of a string object".

```
var mack = [];
mack.push("Mack");
mack.push("the");
mack.push("Knife");
show(mack.join(" "));
show(mack.pop());
show(mack);
```

The method push, which is associated with arrays, can be used to add values to it. It could have been used in the last exercise, as an alternative to result[i] = i. Then there is pop, the opposite of push: it takes off and returns the last value in the array. join builds a single big string from an array of strings. The parameter it is given is pasted between the values in the array.

Coming back to those cats, we now know that an array would be a good way to store the archive of e-mails. On this page, the function retrieveMails can be used to (magically) get hold of this array. Going over them to process them one after another is not rocket science anymore either:

```
var mailArchive = retrieveMails();
for (var i = 0; i < mailArchive.length; i++) {
  var email = mailArchive[i];
  print("Processing e-mail #", i);
  // Do more things...
}
```

We have also decided on a way to represent the set of cats that are alive. The next problem, then, is to find the paragraphs in an e-mail that start with"bor" or "died".

The first question that comes up is what exactly a paragraph is. In this case, the string value itself can't help us much: JavaScript's concept of text does not go any deeper than the "sequence of characters" idea, so we must define paragraphs in those terms.

Earlier, we saw that there is such a thing as a newline character. These are what most people use to split paragraphs. We consider a paragraph, then, to be a part of an e-mail that starts at a newline character or at the start of the content, and ends at the next newline character or at the end of the content.

And we don't even have to write the algorithm for splitting a string into paragraphs ourselves. Strings already have a method named split, which is (almost) the opposite of the join method of arrays. It splits a string into an array, using the string given as its argument to determine in which places to cut.

```
var words = "Cities of the Interior";
show(words.split(" "));
```

Thus, cutting on newlines ("
n"), can be used to split an e-mail into paragraphs.

EXERCISE 4.3

`split` and `join` are not precisely each other's inverse. `string.split(x).join(x)` always produces the original value, but `array.join(x).split(x)` does not. Can you give an example of an array where `.join(" ").split(" ")` produces a different value?

Paragraphs that do not start with either "born" or"died" can be ignored by the program. How do we test whether a string starts with a certain word? The method `charAt` can be used to get a specific character from a string. `x.charAt(0)` gives the first character, `1` is the second one, and so on. One way to check whether a string starts with "born" is:

```
var paragraph = "born 15-11-2003 (mother Spot): White Fang";
show(paragraph.charAt(0) == "b" && paragraph.charAt(1) == "o" &&
 paragraph.charAt(2) == "r" && paragraph.charAt(3) == "n");
```

But that gets a bit clumsy - imagine checking for a word of ten characters. There is something to be learned here though: when a line gets ridiculously long, it can be spread over multiple lines. The result can be made easier to read by lining up the start of the new line with the first element on the original line that plays a similar role.

Strings also have a method called `slice`. It copies out a piece of the string, starting from the character at the position given by the first argument, and ending before (not including) the character at the position given by the second one. This allows the check to be written in a shorter way.

```
show(paragraph.slice(0, 4) == "born");
```

EXERCISE 4.4

Write a function called `startsWith` that takes two arguments, both strings. It returns `true` when the first argument starts with the characters in the second argument, and `false` otherwise.

What happens when `charAt` or `slice` are used to take a piece of a string that does not exist? Will the `startsWith` I showed still work when the pattern is longer than the string it is matched against?

```
show("Pip".charAt(250));
show("Nop".slice(1, 10));
```

`charAt` will return `""` when there is no character at the given position, and `slice` will simply leave out the part of the new string that does not exist.

So yes, that version of `startsWith` works. When `startsWith("Idiots", "Most honoured colleagues")` is called, the call to `slice` will, because string does not have enough characters, always return a string that is shorter than `pattern`. Because of that, the comparison with == will return `false`, which is correct.

Corner cases

It helps to always take a moment to consider abnormal (but valid) inputs for a program. These are usually called *corner cases*, and it is very common for programs that work perfectly on all the "normal" inputs to screw up on corner cases.

The only part of the cat-problem that is still unsolved is the extraction of names from a paragraph. The algorithm was this:

a Find the colon in the paragraph.

b Take the part after this colon.

c Split this part into separate names by looking for commas.

This has to happen both for paragraphs that start with "died", and paragraphs that start with "born". It would be a good idea to put it into a function, so that the two pieces of code that handle these different kinds of paragraphs can both use it.

EXERCISE 4.5

Can you write a function `catNames` that takes a paragraph as an argument and returns an array of names?

Strings have an `indexOf` method that can be used to find the (first) position of a character or sub-string within that string. Also, when `slice` is given only one argument, it will return the part of the string from the given position all the way to the end.

It can be helpful to use the console to "explore" functions. For example, type `"foo: bar".indexOf(":")` and see what you get.

All that remains now is putting the pieces together. One way to do that looks like this:

```
var mailArchive = retrieveMails();
var livingCats = {"Spot": true};
for (var mail = 0; mail < mailArchive.length; mail++) {
  var paragraphs = mailArchive[mail].split("\n");
  for (var paragraph = 0; paragraph < paragraphs.length; paragraph++) {
    if (startsWith(paragraphs[paragraph], "born")) {
      var names = catNames(paragraphs[paragraph]);
      for (var name = 0; name < names.length; name++)
        livingCats[names[name]] = true;
      }
    else if (startsWith(paragraphs[paragraph], "died")) {
      var names = catNames(paragraphs[paragraph]);
      for (var name = 0; name < names.length; name++)
        delete livingCats[names[name]];
    }
  }
}
show(livingCats);
```

That is quite a big dense chunk of code. We'll look into making it a bit lighter in a moment. But first let us look at our results. We know how to check whether a specific cat survives:

```
if ("Spot" in livingCats)
  print("Spot lives!");
else
  print("Good old Spot, may she rest in peace.");
```

But how do we list all the cats that are alive? The `in` keyword has a somewhat different meaning when it is used together with `for`:

```
for (var cat in livingCats) print(cat);
```

A loop like that will go over the names of the properties in an object, which allows us to enumerate all the names in our set.

5 Elegant code

Some pieces of code look like an impenetrable jungle. The example solution to the cat problem suffers from this. One way to make some light shine through it is to just add some strategic blank lines. This makes it look better, but doesn't really solve the problem.

What is needed here is to break the code up. We already wrote two helper functions, `startsWith` and `catNames`, which both take care of a small, understandable part of the problem. Let us continue doing this.

```
function addToSet(set, values) {
  for (var i = 0; i < values.length; i++)
    set[values[i]] = true;
}
function removeFromSet(set, values) {
  for (var i = 0; i < values.length; i++)
    delete set[values[i]];
}
```

These two functions take care of the adding and removing of names from the set. That already cuts out the two most inner loops from the solution:

```
var livingCats = {Spot: true};
for (var mail = 0; mail < mailArchive.length; mail++) {
  var paragraphs = mailArchive[mail].split("\n");
  for (var paragraph = 0; paragraph < paragraphs.length; paragraph++) {
    if (startsWith(paragraphs[paragraph], "born"))
      addToSet(livingCats, catNames(paragraphs[paragraph]));
    else if (startsWith(paragraphs[paragraph], "died"))
      removeFromSet(livingCats, catNames(paragraphs[paragraph]));
  }
}
```

Quite an improvement, if I may say so myself.

Why do `addToSet` and `removeFromSet` take the set as an argument? They could use the variable `livingCats` directly, if they wanted to. The reason is that this way they are not completely tied to our current problem. If `addToSet` directly changed `livingCats`, it would have to be called `addCatsToCatSet`, or something similar. The way it is now, it is a more generally useful tool.

Even if we are never going to use these functions for anything else, which is quite probable, it is useful to write them like this. Because they are "self sufficient", they can be read and understood on their own, without needing to know about some external variable called `livingCats`.

The functions are not pure: They change the object passed as their set argument. This makes them slightly trickier than real pure functions, but still a lot less confusing than functions that run amok and change any value or variable they please.

We continue breaking the algorithm into pieces:

```
function findLivingCats() {
  var mailArchive = retrieveMails();
  var livingCats = {"Spot": true};
  function handleParagraph(paragraph) {
    if (startsWith(paragraph, "born"))
      addToSet(livingCats, catNames(paragraph));
    else if (startsWith(paragraph, "died"))
```

```
      removeFromSet(livingCats, catNames(paragraph));
  }
  for (var mail = 0; mail < mailArchive.length; mail++) {
    var paragraphs = mailArchive[mail].split("\n");
    for (var i = 0; i < paragraphs.length; i++)
      handleParagraph(paragraphs[i]);
  }
  return livingCats;
}

var howMany = 0;
for (var cat in findLivingCats())
  howMany++;
print("There are ", howMany, " cats.");
```

The whole algorithm is now encapsulated by a function. This means that it does not leave a mess after it runs: `livingCats` is now a local variable in the function, instead of a top-level one, so it only exists while the function runs. The code that needs this set can call `findLivingCats` and use the value it returns.

It seemed to me that making `handleParagraph` a separate function also cleared things up. But this one is so closely tied to the cat-algorithm that it is meaningless in any other situation. On top of that, it needs access to the `livingCats` variable. Thus, it is a perfect candidate to be a function-inside-a-function. When it lives inside `findLivingCats`, it is clear that it is only relevant there, and it has access to the variables of its parent function.

This solution is actually bigger than the previous one. Still, it is tidier and I hope you'll agree that it is easier to read.

6 Date

The program still ignores a lot of the information that is contained in the e-mails. There are birth-dates, dates of death, and the names of mothers in there.

To start with the dates: What would be a good way to store a date? We could make an object with three properties, year, month, and day, and store numbers in them.

```
var when = {year: 1980, month: 2, day: 1};
```

new

But JavaScript already provides a kind of object for this purpose. Such an object can be created by using the keyword *new*:

```
var when = new Date(1980, 1, 1);
show(when);
```

Constructor

Just like the notation with braces and colons we have already seen, new is a way to create object values. Instead of specifying all the property names and values, a function is used to build up the object. This makes it possible to define a kind of standard procedure for creating objects. Functions like this are called *constructors*, and in chapter 8 we will see how to write them.

The Date constructor can be used in different ways.

```
show(new Date());
show(new Date(1980, 1, 1));
show(new Date(2007, 2, 30, 8, 20, 30));
```

As you can see, these objects can store a time of day as well as a date. When not given any arguments, an object representing the current time and date is created. Arguments can be given to ask for a specific date and time. The order of the arguments is `year`, `month`, `day`, `hour`, `minute`, `second`, `milliseconds`. These last four are optional, they become `0` when not given.

The month numbers these objects use go from `0` to `11`, which can be confusing. Especially since day numbers do start from `1`.

The content of a Date object can be inspected with a number of `get`... methods.

```
var today = new Date();
print("Year: ", today.getFullYear(), ",
 month: ", today.getMonth(), ",
 day: ", today.getDate());
print("Hour: ", today.getHours(), ",
 minutes: ", today.getMinutes(), ",
 seconds: ", today.getSeconds());
print("Day of week: ", today.getDay());
```

All of these, except for `getDay`, also have a `set`... variant that can be used to change the value of the date object.

Inside the object, a date is represented by the amount of milliseconds it is away from January 1st 1970. You can imagine this is quite a large number.

```
var today = new Date();
show(today.getTime());
```

A very useful thing to do with dates is comparing them.

```
var wallFall = new Date(1989, 10, 9);
var gulfWarOne = new Date(1990, 6, 2);
show(wallFall < gulfWarOne);
show(wallFall == wallFall);
// but
show(wallFall == new Date(1989, 10, 9));
```

Comparing dates with <, >, <=, and >= does exactly what you would expect. When a date object is compared to itself with == the result is `true`, which is also good. But when == is used to compare a date object to a different, equal date object, we get `false`. Huh?

As mentioned earlier, == will return `false` when comparing two different objects, even if they contain the same properties. This is a bit clumsy and error-prone here, since one would expect >= and == to behave in a more or less similar way. Testing whether two dates are equal can be done like this:

```
var wallFall1 = new Date(1989, 10, 9),
 wallFall2 = new Date(1989, 10, 9);
show(wallFall1.getTime() == wallFall2.getTime());
```

In addition to a date and time, `Date` objects also contain information about a timezone. When it is one o'clock in Amsterdam, it can, depending on the time of year, be noon in London, and seven in the morning in New York. Such times can only be compared when you take their time zones into account. The `getTimezoneOffset` function of a `Date` can be used to find out how many minutes it differs from GMT (Greenwich Mean Time).

```
var now = new Date();
print(now.getTimezoneOffset());
```

EXERCISE 4.6

```
"died 27/04/2006: Black Leclère"
```

The date part is always in the exact same place of a paragraph. How convenient. Write a function `extractDate` that takes such a paragraph as its argument, extracts the date, and returns it as a date object.

7 The Cat program

Storing cats will work differently from now on. Instead of just putting the value `true` into the set, we store an object with information about the cat. When a cat dies, we do not remove it from the set, we just add a property death to the object to store the date on which the creature died.

This means our `addToSet` and `removeFromSet` functions have become useless. Something similar is needed, but it must also store birth-dates and, later, the mother's name.

```
function catRecord(name, birthdate, mother) {
  return {name: name, birth: birthdate, mother: mother};
}
function addCats(set, names, birthdate, mother) {
  for (var i = 0; i < names.length; i++)
    set[names[i]] = catRecord(names[i], birthdate, mother);
}
function deadCats(set, names, deathdate) {
  for (var i = 0; i < names.length; i++)
    set[names[i]].death = deathdate;
}
```

`catRecord` is a separate function for creating these storage objects. It might be useful in other situations, such as creating the object for Spot. "Record" is a term often used for objects like this, which are used to group a limited number of values.

So let us try to extract the names of the mother cats from the paragraphs.

```
"born 15/11/2003 (mother Spot): White Fang"
```

One way to do this would be...

```
function extractMother(paragraph) {
  var start = paragraph.indexOf("(mother ") + "(mother ".length;
  var end = paragraph.indexOf(")");
  return paragraph.slice(start, end);
}
show(extractMother("born 15/11/2003 (mother Spot): White Fang"));
```

Notice how the start position has to be adjusted for the length of "(mother ", because `indexOf` returns the position of the start of the pattern, not its end.

EXERCISE 4.7

The thing that `extractMother` does can be expressed in a more general way. Write a function `between` that takes three arguments, all of which are strings. It will return the part of the first argument that occurs between the patterns given by the second and the third arguments.

So `between("born 15/11/2003 (mother Spot): White Fang", "(mother ", ")")` gives `"Spot"`.

`between("bu ] boo [ bah ] gzz", "[ ", " ]")` returns `"bah"`.

To make that second test work, it can be useful to know that `indexOf` can be given a second, optional parameter that specifies at which point it should start searching.

Having `between` makes it possible to express `extractMother` in a simpler way:

```
function extractMother(paragraph) {
  return between(paragraph, "(mother ", ")");
}
```

The new, improved cat-algorithm looks like this:

```
function findCats() {
  var mailArchive = retrieveMails();
  var cats = {
    "Spot": catRecord("Spot", new Date(1997, 2, 5), "unknown")
  };
  function handleParagraph(paragraph) {
    if (startsWith(paragraph, "born"))
      addCats(cats, catNames(paragraph), extractDate(paragraph), extractMother
          (paragraph));    else if (startsWith(paragraph, "died"))
        deadCats(cats, catNames(paragraph), extractDate(paragraph));
  }
  for (var mail = 0; mail < mailArchive.length; mail++) {
    var paragraphs = mailArchive[mail].split("\n");
    for (var i = 0; i < paragraphs.length; i++)
      handleParagraph(paragraphs[i]);
  }
  return cats;
}
var catData = findCats();
```

Having that extra data allows us to finally have a clue about the cats aunt Emily talks about. A function like this could be useful:

```
function formatDate(date) {
  return date.getDate() + "/" + (date.getMonth() + 1) + "/" + date.
      getFullYear();
}
function catInfo(data, name) {
  if (!(name in data))
    return "No cat by the name of " + name + " is known.";
  var cat = data[name];
  var message = name + ", born " + formatDate(cat.birth) +
          " from mother " + cat.mother;
  if ("death" in cat)
    message += ", died " + formatDate(cat.death);
  return message + ".";
}
print(catInfo(catData, "Fat Igor"));
```

The first `return` statement in `catInfo` is used as an escape hatch. If there is no data about the given cat, the rest of the function is meaningless, so we immediately return a value, which prevents the rest of the code from running.

Multiple return statements

In the past, certain groups of programmers considered functions that contain *multiple* *return statements* sinful. The idea was that this made it hard to see which code was executed and which code was not. Other techniques, which will be discussed in chapter 5, have made the reasons behind this idea more or less obsolete, but you might still occasionally come across someone who will criticise the use of "shortcut" `return` statements.

EXERCISE 4.8

The `formatDate` function used by `catInfo` does not add a zero before the month and the day part when these are only one digit long. Write a new version that does this.

EXERCISE 4.9

Write a function `oldestCat` which, given an object containing cats as its argument, returns the name of the oldest living cat.

8 Arguments

arguments

Now that we are familiar with arrays, I can show you something related. Whenever a function is called, a special variable named *`arguments`* is added to the environment in which the function body runs. This variable refers to an object that resembles an array. It has a property `0` for the first argument, `1` for the second, and so on for every argument the function was given. It also has a `length` property.

This object is not a real array though, it does not have methods like `push`, and it does not automatically update its `length` property when you add something to it. Why not, I never really found out, but this is something one needs to be aware of.

```
function argumentCounter() {
  print("You gave me ", arguments.length, " arguments.");
}
argumentCounter("Death", "Famine", "Pestilence");
```

Default value

Some functions can take any number of arguments, like `print` does. These typically loop over the values in the arguments object to do something with them. Others can take optional arguments which, when not given by the caller, get some sensible *default value*.

```
function add(number, howmuch) {
  if (arguments.length < 2)
    howmuch = 1;
  return number + howmuch;
}
show(add(6));
show(add(6, 4));
```

EXERCISE 4.10

Extend the `range` function from exercise 4.2 to take a second, optional argument. If only one argument is given, it behaves as earlier and produces a range from `0` to the given number. If two arguments are given, the first indicates the start of the range, the second the end.

EXERCISE 4.11

You may remember this line of code from the introduction:

```
print(sum(range(1, 10)));
```

We have `range` now. All we need to make this line work is a `sum` function. This function takes an array of numbers, and returns their sum. Write it, it should be easy.

9 Math

Chapter 2 mentioned the functions `Math.max` and `Math.min`. With what you know now, you will notice that these are really the properties `max` and `min` of the object stored under the name *`Math`*. This is another role that objects can play: A warehouse holding a number of related values.

Math

There are quite a lot of values inside `Math`, if they would all have been placed directly into the global environment they would, as it is called, pollute it. The more names have been taken, the more likely one is to accidentally overwrite the value of some variable. For example, it is not a far shot to want to name something `max`.

Most languages will stop you, or at least warn you, when you are defining a variable with a name that is already taken. Not JavaScript.

In any case, one can find a whole outfit of mathematical functions and constants inside `Math`. All the trigonometric functions are there - `cos`, `sin`, `tan`, `acos`, `asin`, `atan` - π and e, which are written with all capital letters (`PI` and `E`, which was, at one time, a fashionable way to indicate something is a constant. `pow` is a good replacement for the power functions we have been writing, it also accepts negative and fractional exponents. `sqrt` takes square roots. `max` and `min` can give the maximum or minimum of two values. `round`, `floor`, and `ceil` will round numbers to the closest whole number, the whole number below it, and the whole number above it respectively.

There are a number of other values in `Math`, but this text is an introduction, not a reference. References are what you look at when you suspect something exists in the language, but need to find out what it is called or how it works exactly. Unfortunately, there is no one comprehensive complete reference for JavaScript. This is mostly because its current form is the result of a chaotic process of different browsers adding different extensions at different times. The ECMA standard document that was mentioned in the introduction provides a solid documentation of the basic language, but is more or less unreadable. For most things, your best bet is the Mozilla Developer Network.

Maybe you already thought of a way to find out what is available in the `Math` object:

```
for (var name in Math)
  print(name);
```

But alas, nothing appears. Similarly, when you do this:

```
for (var name in ["Huey", "Dewey", "Loui"])
  print(name);
```

You only see `0`, `1`, and `2`, not `length`, or `push`, or `join`, which are definitely also in there. Apparently, some properties of objects are hidden. There is a good reason for this: All objects have a few methods, for example `toString`, which converts the object into some kind of relevant string, and you do not want to see those when you are, for example, looking for the cats that you stored in the object.

Why the properties of `Math` are hidden is unclear to me. Someone probably wanted it to be a mysterious kind of object.

All properties your programs add to objects are visible. There is no way to make them hidden, which is unfortunate because, as we will see in chapter 8, it would be nice to be able to add methods to objects without having them show up in our `for/in` loops.

Some properties are read-only, you can get their value but not change it. For example, the properties of a string value are all read-only.

Other properties can be "active". Changing them causes things to happen. For example, lowering the length of an array causes excess elements to be discarded:

```
var array = ["Heaven", "Earth", "Man"];
array.length = 2;
show(array);
```

FEEDBACK

Answers to the exercises

4.1 This can be done by storing the content of the set as the properties of an object. Adding a name is done by setting a property by that name to a value, any value. Removing a name is done by deleting this property. The `in` operator can be used to determine whether a certain name is part of the set.

```
 var set = "Spot":  true;
// Add "White Fang" to the set
set["White Fang"] = true;
// Remove "Spot"
delete set["Spot"];
// See if "Asoka" is in the set
show("Asoka" in set);
```

4.2
```
 function range(upto) {
  var result = [];
  for (var i = 0; i <= upto; i++)
     result[i] = i;
  return result; }
show(range(4));
}
```

Instead of naming the loop variable counter or current, as I have been doing so far, it is now called simply `i`. Using single letters, usually `i`, `j`, or `k` for loop variables is a widely spread habit among programmers. It has its origin mostly in laziness: We'd rather type one character than seven, and names like counter and current do not really clarify the meaning of the variable much.

If a program uses too many meaningless single-letter variables, it can become unbelievably confusing. In my own programs, I try to only do this in a few common cases. Small loops are one of these cases. If the loop contains another loop, and that one also uses a variable named i, the inner loop will modify the variable that the outer loop is using, and everything will break. One could use `j` for the inner loop, but in general, when the body of a loop is big, you should come up with a variable name that has some clear meaning.

4.3
```
 var array = ["a", "b", "c d"];
show(array.join(" ").split(" "));
```

4.4
```
 function startsWith(string, pattern)
tabto0.5cmreturn string.slice(0, pattern.length) == pattern;

show(startsWith("rotation", "rot"));
```

4.5
```
 function catNames(paragraph) {
  var colon = paragraph.indexOf(":");
  return paragraph.slice(colon + 2).split(", ");
}
show(catNames("born 20/09/2004 (mother Yellow Bess):  " +
  "Doctor Hobbles the 2nd, Noog"));
}
```

The tricky part, which the original description of the algorithm ignored, is dealing with spaces after the colon and the commas. The `+2` used when slicing the string is needed to leave out the colon itself and the space after it. The argument to `split` contains both a comma and a space, because that is what the names are really separated by, rather than just a comma.

This function does not do any checking for problems. We assume, in this case, that the input is always correct.

4.6
```
 function extractDate(paragraph) {
  function numberAt(start, length) {
    return Number(paragraph.slice(start, start + length));
  }
  return new Date(numberAt(11, 4),
    numberAt(8, 2) -1,
    numberAt(5, 2));
}
show(extractDate("died 27-04-2006:  Black Leclère"));
```

It would work without the calls to `Number`, but as mentioned earlier, I prefer not to use strings as if they are numbers. The inner function was introduced to prevent having to repeat the `Number` and `slice` part three times.

Note the `-1` for the month number. Like most people, Aunt Emily counts her months from 1, so we have to adjust the value before giving it to the `Date` constructor. (The day number does not have this problem, since `Date` objects count days in the usual human way.)

In chapter 10 we will see a more practical and robust way of extracting pieces from strings that have a fixed structure.

4.7
```
 function between(string, start, end) {
  var startAt = string.indexOf(start) + start.length;
  var endAt = string.indexOf(end, startAt);
  return string.slice(startAt, endAt);
}
show(between("bu ] boo [ bah ] gzz", "[ ", " ]"));
```

4.8
```
 function formatDate(date) {
  function pad(number) {
    if (number < 10)
      return "0" + number;
    else
      return number;
}
  return pad(date.getDate()) + "/" + pad(date.getMonth() + 1) +
      "/" + date.getFullYear();
}
print(formatDate(new Date(2000, 0, 1)));
```

4.9
```
 function oldestCat(data) {
  var oldest = null;

tabto0.5cmfor (var name in data) {
    var cat = data[name];
    if (!("death" in cat) &&
        (oldest == null ||
          oldest.birth > cat.birth))
        oldest = cat;
    }
    if (oldest == null)
      return null;
    else
      return oldest.name;
}
print(oldestCat(catData));
```

The condition in the `if` statement might seem a little intimidating. It can be read as "only store the current cat in the variable oldest if it is not dead, and oldest is either `null` or a cat that was born after the current cat".

Note that this function returns `null` when there are no living cats in data. What does your solution do in that case?

4.10

```
function range(start, end) {
    if (arguments.length < 2) {
      end = start;
      start = 0;
    }
    var result = [];
    for (var i = start; i <= end; i++)
      result.push(i);
    return result;
}
show(range(4));
show(range(2, 4));
```

The optional argument does not work precisely like the one in the `add` example above. When it is not given, the first argument takes the role of end, and start becomes `0`.

4.11

```
function sum(numbers) {
    var total = 0;
    for (var i = 0; i < numbers.length; i++)
      total += numbers[i];
    return total;
}
print(sum(range(1, 10)));
```

Error Handling

Chapter 5

Error Handling

Writing programs that work when everything goes as expected is a good start. Making your programs behave properly when encountering unexpected conditions is where it really gets challenging.

1 Programmer mistakes and genuine problems

The problematic situations that a program can encounter fall into two categories: Programmer mistakes and genuine problems. If someone forgets to pass a required argument to a function, that is an example of the first kind of problem. On the other hand, if a program asks the user to enter a name and it gets back an empty string, that is something the programmer can not prevent.

In general, one deals with programmer errors by finding and fixing them, and with genuine errors by having the code check for them and perform some suitable action to remedy them (for example, asking for the name again), or at least fail in a well-defined and clean way.

1.1 TYPE PROBLEMS

It is important to decide into which of these categories a certain problem falls. For example, consider our old power function:

```
function power(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base; return result;
}
```

When some geek tries to call `power("Rabbit", 4)`, that is quite obviously a programmer error, but how about `power(9, 0.5)`? The function can not handle fractional exponents, but, mathematically speaking, raising a number to the halfth power is perfectly reasonable (`Math.pow` can handle it). In situations where it is not entirely clear what kind of input a function accepts, it is often a good idea to explicitly state the kind of arguments that are acceptable in a comment.

1.2 VALUE PROBLEMS

If a function encounters a problem that it can not solve itself, what should it do? In chapter 4 we wrote the function between:

```
function between(string, start, end) {
  var startAt = string.indexOf(start) + start.length;
  var endAt = string.indexOf(end, startAt);
  return string.slice(startAt, endAt);
}
```

If the given `start` and `end` do not occur in the string, `indexOf` will return `-1` and this version of between will return a lot of nonsense: `between("Your mother!", "-", "-")` returns `"our mother"`.

When the program is running, and the function is called like that, the code that called it will get a string value, as it expected, and happily continue doing something with it. But the value is wrong, so whatever it ends up doing with it will also be wrong. And if you are unlucky, this wrongness only causes a problem after having passed through twenty other functions. In cases like that, it is extremely hard to find out where the problem started.

In some cases, you will be so unconcerned about these problems that you don't mind the function misbehaving when given incorrect input. For example, if you know for sure the function will only be called from a few places, and you can prove that these places give it decent input, it is generally not worth the trouble to make the function bigger and uglier so that it can handle problematic cases.

2 Solutions

2.1 RETURN A SPECIAL VALUE

But most of the time, functions that fail "silently" are hard to use, and even dangerous. What if the code calling between wants to know whether everything went well? At the moment, it can not tell, except by re-doing all the work that between did and checking the result of between with its own result. That is bad. One solution is to make between return a special value, such as false or undefined, when it fails.

```
function between(string, start, end) {
  var startAt = string.indexOf(start);
  if (startAt == -1)
    return undefined;
  startAt += start.length;
  var endAt = string.indexOf(end, startAt);
  if (endAt == -1)
    return undefined;
  return string.slice(startAt, endAt);
}
```

You can see that error checking does not generally make functions prettier. But now code that calls between can do something like:

```
var input = prompt("Tell me something", "");
var parenthesized = between(input, "(", ")");
if (parenthesized != undefined)
  print("You parenthesized '", parenthesized, "'.");
```

2.1.1 *Problems with this solution*

In many cases returning a special value is a perfectly fine way to indicate an error. It does, however, have its downsides. Firstly, what if the function can already return every possible kind of value? For example, consider this function that gets the last element from an array:

```
function lastElement(array) {
  if (array.length > 0)
    return array[array.length -1];
  else
    return undefined;
}
```

```
show(lastElement([1, 2, undefined]));
```

So did the array have a last element? Looking at the value lastElement returns, it is impossible to say.

The second issue with returning special values is that it can sometimes lead to a whole lot of clutter. If a piece of code calls between ten times, it has to check ten times whether undefined was returned. Also, if a function calls between but does not have a strategy to recover from a failure, it will have to check the return value of between, and if it is undefined, this function can then return undefined or some other special value to its caller, who in turn also checks for this value.

2.2 EXCEPTION HANDLING

Sometimes, when something strange occurs, it would be practical to just stop doing what we are doing and immediately jump back to a place that knows how to handle the problem.

Exception handling

Well, we are in luck, a lot of programming languages provide such a thing. Usually, it is called *exception handling*.

The theory behind exception handling goes like this: It is possible for code to raise (or throw) an exception, which is a value. Raising an exception somewhat resembles a super-charged return from a function - it does not just jump out of the current function, but also out of its callers, all the way up to the top-level call that started the current execution. This is called unwinding the stack. You may remember the stack of function calls that was mentioned in chapter 3. An exception zooms down this stack, throwing away all the call contexts it encounters.

If they always zoomed right down to the base of the stack, exceptions would not be of much use, they would just provide a novel way to blow up your program. Fortunately, it is possible to set obstacles for exceptions along the stack. These 'catch' the exception as it is zooming down, and can do something with it, after which the program continues running at the point where the exception was caught.

An example:

```
function lastElement(array) {
 if (array.length > 0)
   return array[array.length -1];
 else throw "Can not take the last element of an empty array.";
}
function lastElementPlusTen(array) {
 return lastElement(array) + 10;
}
try {
 print(lastElementPlusTen([]));
}
catch (error) {
 print("Something went wrong: ", error);
}
```

throw
try
catch

`throw` is the keyword that is used to raise an exception. The keyword `try` sets up an obstacle for exceptions: When the code in the block after it raises an exception, the *catch* block will be executed. The variable named in parentheses after the word `catch` is the name given to the exception value inside this block.

Note that the function `lastElementPlusTen` completely ignores the possibility that `lastElement` might go wrong. This is the big advantage of exceptions - error-handling code is only necessary at the point where the error occurs, and the point where it is handled. The functions in between can forget all about it.

Well, almost.

Consider the following: A function `processThing` wants to set a top-level variable `currentThing` to point to a specific thing while its body executes, so that other functions can have access to that thing too. Normally you would of course just pass the thing as an argument, but assume for a moment that that is not practical. When the function finishes, `currentThing` should be set back to `null`.

```
var currentThing = null;
function processThing(thing) {
  if (currentThing != null)
    throw "Oh no! We are already processing a thing!";
  currentThing = thing;
  /* do complicated processing... */
  currentThing = null;
}
```

But what if the complicated processing raises an exception? In that case the call to `processThing` will be thrown off the stack by the exception, and `currentThing` will never be reset to `null`.

finally

`try` statements can also be followed by a *finally* keyword, which means “no matter what happens, run this code after trying to run the code in the try block”. If a function has to clean something up, the cleanup code should usually be put into a finally block:

```
function processThing(thing) {
  if (currentThing != null)
    throw "Oh no! We are already processing a thing!";
  currentThing = thing;
  try {
  /* do complicated processing... */
  }
  finally {
    currentThing = null;
  }
}
```

2.2.1 *Exceptions in the JavaScript environment*

A lot of errors in programs cause the JavaScript environment to raise an exception. For example:

```
try {
  print(Sasquatch);
}
catch (error) {
  print("Caught: " + error.message);
}
```

In cases like this, special error objects are raised. These always have a `message` property containing a description of the problem. You can raise similar objects using the `new` keyword and the `Error` constructor:

```
throw new Error("Fire!");
```

When an exception goes all the way to the bottom of the stack without being caught, it gets handled by the environment. What this means differs between the different browsers, sometimes a description of the error is written to some kind of log, sometimes a window pops up describing the error.

The errors produced by entering code in the console on this page are always caught by the console, and displayed among the other output.

Throwing string values, as some of the examples in this chapter do, is rarely a good idea, because it makes it hard to recognise the type of the exception. A better idea is to introduce a new type of objects, as described in chapter 8.

Index