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*

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.

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:

1. Store the number 0 in memory location 0
2. Store the number 1 in memory location 1
3. Store the value of memory location 1 in memory location 2
4. Subtract the number 11 from the value in memory location 2
5. If the value in memory location 2 is the number 0, continue with instruction 9
6. Add the value of memory location 1 to memory location 0
7. Add the number 1 to the value of memory location 1
8. Continue with instruction 3
9. 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).

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.

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.

At the time I am writing this, people are working on a thing called ECMAScript 4. This is a new version of the ECMAScript language, which adds a number of new features. You should not worry too much about this new version making the things you learn from this book obsolete. For one thing, ECMAScript 4 will mostly be an extension of the language we have now, so almost everything written in this book will still hold. On top of that, it will most likely take quite a while before all major browsers add these new features to their JavaScript systems, and until they do, ECMAScript 4 won't be very practical for web programming.

Most chapters in this book contain quite a lot of code[1]. 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.

To allow you to try out programs, both the examples and the code you write yourself, this book makes use of something called a console. If you are using a modern graphical browser (Internet Explorer version 6 or higher, Firefox 1.5 or higher, Opera 9 or higher, Safari 3 or higher), the pages in this book will show a bar at the bottom of your screen. You can open the console by clicking on the little arrow on the far right of this bar.

The console contains three important elements. There is the output window, which is used to show error messages and things that programs print out. Below that, there is a line where you can type in a piece of JavaScript. Try typing in a number, and pressing the enter key to run what you typed. If the text you typed produced something meaningful, it will be shown in the output window. Now try typing `wrong!`, and press enter again. The output window will show an error message. You can use the arrow-up and arrow-down keys to go back to previous commands that you typed.

For bigger pieces of code, those that span multiple lines and which you want to keep around for a while, the field on the right can be used. The 'Run' button is used to execute programs written in this field. It is possible to have multiple programs open at the same time. Use the 'New' and 'Load' buttons to add a new program (empty or from a file on the web). When there is more than one open program, the menu next to the 'Run' button can be used to choose which one is being shown. The 'Close' button, as you might expect,

closes a program.

Example programs in this book always have a small button with an arrow in their top-right corner, which can be used to run them. The example we saw earlier looked like this:

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

Run it by clicking the arrow. There is also another button, which is used to load the program into the console. Do not hesitate to modify it and try out the result. The worst that could happen is that you create an endless loop. An endless loop is what you get when the condition of the `while` never becomes false, for example if you choose to add `0` instead of `1` to the count variable. Now the program will run forever.

Fortunately, browsers keep an eye on the programs running inside them. Whenever one of them is taking suspiciously long to finish, they will ask you if you want to cut it off.

In some later chapters, we will build example programs that consist of many blocks of code. Often, you have to run every one of them for the program to work. As you may have noticed, the arrow in a block of code turns purple after the block has been run. When reading a chapter, try to run every block of code you come across, especially those that 'define' something new (you will see what that means in the next chapter).

It is, of course, possible that you can not read a chapter in one sitting. This means you will have to start halfway when you continue reading, but if you don't run all the code starting from the top of the chapter, some things might not work. By holding the shift key while pressing the 'run' arrow on a block of code, all blocks before that one will be run as well, so when you start in the middle of a chapter, hold shift the first time you run a piece of code, and everything should work as expected.

Finally, the little face in the top-left corner of your screen can be used to send me, the author, a message. If you have a comment, or you find a passage ridiculously confusing, or you just spot a spelling error, tell me about it. Sending a message can be done without leaving the page, so it won't interrupt your reading.

1. 'Code' is the substance that programs are made of. Every piece of a program, whether it is a single line or the whole thing, can be referred to as 'code'.

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 bits[1], and is thus fundamentally alike, every piece of data plays its own role. In JavaScript's system, most of this data is neatly separated into things called values. Every value has a type, which determines the kind of role it can play. There are six basic types of values: Numbers, strings, booleans, objects, functions, and undefined values.

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

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

```
144
```

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

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

```
0100000001100010000000000000000000000000000000000000000000000000
```

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

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

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

Fractional numbers are written by using a dot.

```
9.81
```

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

```
2.998e8
```

Which is 2.998 * 10^8 = 299800000.

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

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

```
100 + 4 * 11
```

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

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

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

For subtraction, there is the `-` operator, and division can be done with `/`. When operators appear together without parentheses, the order in which they are applied is determined by the precedence of the operators. The first example shows that multiplication has a higher precedence than addition. The full ordering of the arithmetic operators is: first division, then multiplication, then subtraction, and finally addition.

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

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

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

There is one more arithmetic operator which is probably less familiar to you. The `%` symbol is used to represent the modulo operation. `X` modulo `Y` is the remainder of dividing `X` by `Y`. For example `314 % 100` is `14`, `10 % 3` is `1`, and `144 % 12` is `0`. Modulo's precedence lies between that of multiplication and subtraction.

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

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

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

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

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

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

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

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

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

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

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

```
typeof 4.5
```

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

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

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

```
3 > 2
```

And `false` can be produced like this:

```
3 < 2
```

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

Strings can be compared in the same way:

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

The way strings are ordered is more or less alphabetic. More or less... Uppercase letters are always 'less' than lowercase ones, so `"Z" < "a"` 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"
```

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

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

```
true && false
```

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

```
true || false
```

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

Ex. 2.1

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

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

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

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

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

```
true && !false
```

```
true
```

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

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

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

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

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

```
1;
!false;
```

It is a useless program. An expression can be content to just produce a value, but a statement only amounts to something if it somehow changes the world. It could print something to the screen — that counts as changing the world — or it could change the internal state of the program in a way that will affect the statements that come after it. These changes are called 'side effects'. The statements in the example above just produce the values `1` and `true`, and then immediately throw them into the bit bucket[5]. This leaves no impression on the world at all, and is not a side effect.

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

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

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

produced by multiplying `5` by `5`.

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

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

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

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

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

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

```
var luigiDebt = 140;
```

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

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

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

A lot of the values provided by the standard environment have the type 'function'. A function is a piece of program wrapped in a value. Generally, this piece of program does something useful, which can be evoked 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("Also, your hair is on fire.");
```

Executing the code in a function is called invoking 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. The string value between the parentheses 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 two arguments and gives back the biggest of the two:

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

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

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

Chapter 3 discusses writing your own functions.

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

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

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

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

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

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

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

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

It is possible to give almost every variable in the environment a new value. This can be useful, but also dangerous. If you give `print` the value `8`, you won't be able to print things

anymore. Fortunately, there is a big 'Reset' button on the console, which will reset the environment to its original state.

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

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

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

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

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

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

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

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

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

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

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

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

Ex. 2.2

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

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

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

```
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.

Ex. 2.3

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

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

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

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

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.

On browsers other than Opera, 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.

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, every statement that needs a semicolon will always be terminated by one, and I strongly urge you to do the same with your own statements.

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

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

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

This program is exactly equivalent to the earlier even-number-printing example. The only change is that all the statements that are related to the 'state' of the loop are now on one line. The parentheses after the `for` should contain two semicolons. The part before the first semicolon *initialises* the loop, usually by defining a variable. The second part is the expression that *checks* whether the loop must still continue. The final part *updates* the state of the loop. In most cases this is shorter and clearer than a `while` construction.

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

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

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

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

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

Ex. 2.4 Rewrite the solutions of the previous two exercises to use `for` instead of `while`.

```
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);
}
```

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

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

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

We want to show all numbers between 0 and 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 modulo (`%`) 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 modulo gives you, is zero.

If we wanted to print all of the numbers between 0 and 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 gotten by appending an `else` part after an `if` statement. The `else` statement is executed only when the `if`'s condition is false.

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

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

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

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

Ex. 2.5 Write a program to ask yourself, using `prompt`, what the value of 2 + 2 is. If the answer is "4", use `alert` to say something praising. If it is "3" or "5", say "Almost!". In other cases, say something mean.

```
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.");
```

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

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

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

Ex. 2.6 Add a `while` and optionally a `break` to your solution for the previous exercise, so that it keeps repeating the question until a correct answer is given.

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

```
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.");
}
```

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

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

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

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

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

Which brings us to another tricky subject...

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

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

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

`!==`. The first tests whether a value is precisely equal to the other, and the second tests whether it is not precisely equal.

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

All these are `false`.

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

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

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

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

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

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

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

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

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

out of the string.

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

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

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 `+` on 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, and 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, and 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.");
```

1. Bits are any kinds of two-valued things, usually described as `0`s and `1`s. Inside the computer, they take forms like a high or low electrical charge, a strong or weak signal, a shiny or dull spot on the surface of a CD.
2. If you were expecting something like `10010000` here — good call, but read on. JavaScript's numbers are not stored as integers.
3. Actually, 53, because of a trick that can be used to get one bit for free. Look up the 'IEEE 754' format if you are curious about the details.
4. When you type string values at the console, you'll notice that they will come back with the quotes and backslashes the way you typed them. To get special characters to show properly, you can do `print("a\nb")` — why this works, we will see in a moment.
5. The bit bucket is supposedly the place where old bits are kept. On some systems, the programmer has to manually empty it now and then. Fortunately, JavaScript comes with a fully-automatic bit-recycling system.

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.

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));
```

`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[1].

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?

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.

Ex. 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.

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

show(absolute(-144));
```

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[2]. 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");
```

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();
```

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.

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 4 is expected to add some way to define variables that stay inside blocks.

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.

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));
```

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 provably 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.

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.

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.");
```

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;
}
```

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.

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.

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.

Ex. 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.

```
function greaterThan(x) {
  return function(y) {
    return y > x;
  };
}

var greaterThanTen = greaterThan(10);
show(greaterThanTen(9));
```

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.

1. Technically, this wouldn't have been necessary, but I suppose the designers of JavaScript felt it would clarify things if function bodies always had braces.
2. Technically, a pure function can not use the value of any external variables. These values might change, and this could make the function return a different value for the same arguments. In practice, the programmer may consider some variables 'constant' — they are not expected to change — and consider functions that use only constant variables pure. Variables that contain a function value are often good examples of constant variables.

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:

1. Start with a set of cat names that has only "Spot" in it.
2. Go over every e-mail in our archive, in chronological order.
3. Look for paragraphs that start with "born" or "died".
4. Add the names from paragraphs that start with "born" to our set of names.
5. Remove the names from paragraphs that start with "died" from our set.

Where taking the names from a paragraph goes like this:

1. Find the colon in the paragraph.
2. Take the part after this colon.
3. 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.

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.

Numbers, booleans, the value `null`, and the value `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.

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.

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]);
```

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);
```

Ex. 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.

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[1].

```
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);
```

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.

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 object with identical contents will give `false`. This is useful in some situations, but unpractical in others.

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]);
```

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 collections.

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.

Ex. 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.

```
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.

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 is a property. How this works precisely is described in chapter 8.

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. The function `retrieveMails` can be used to (magically) get hold of this array. Going over them to process them one after another is no 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 `"born"` 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.

Ex. 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?

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

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");
```

Ex. 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.

```
function startsWith(string, pattern) {
  return string.slice(0, pattern.length) == pattern;
}

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

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.

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:

1. Find the colon in the paragraph.
2. Take the part after this colon.
3. 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.

Ex. 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.

```
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.

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.

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.

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};
```

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);
```

Just like the notation with braces and semicolons 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());
```

Ex. 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.

```
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.

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.

Ex. 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.

```
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", "[ ", " ]"));
```

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.

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.

Ex. 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.

```
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)));
```

Ex. 4.9 Write a function `oldestCat` which, given an object containing cats as its argument, returns the name of the oldest living cat.

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

  for (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?

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");
```

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));
```

Ex. 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.

```
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`.

Ex. 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.

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

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

The previous chapter showed 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.

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 worked 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 things like the `Math` object and other elementary functionality, a rather good reference can be found here. The old documentation from Netscape, which can still be found at Sun's website, can also be helpful, but is outdated and not entirely correct anymore.

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 'watched'. 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);
```

In some browsers, objects have a method `watch`, which can be used to add a watcher to your own properties. Internet Explorer does not support this though, so it is of little use when writing programs that must run on all the 'big' browsers.

```
var winston = {mind: "compliant"};
function watcher(propertyName, from, to) {
  if (to == "compliant")
    print("Doubleplusgood.");
  else
    print("Transmitting information to Thought Police...");
}
winston.watch("mind", watcher);

winston.mind = "rebellious";
```

1. There are a few subtle problems with this approach, which will be discussed and solved in chapter 8. For this chapter, it works well enough.

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.

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.

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.

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 bogus: `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.

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, "'.");
```

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.

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.

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` 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`.

`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;
  }
}
```

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.

Most programmers consider exceptions purely an error-handling mechanism. In essence, though, they are just another way of influencing the control flow of a program. For example, they can be used as a kind of `break` statement in a recursive function. Here is a slightly strange function which determines whether an object, and the objects stored inside it, contain at least seven `true` values:

```
var FoundSeven = {};

function hasSevenTruths(object) {
  var counted = 0;

  function count(object) {
    for (var name in object) {
      if (object[name] === true) {
        counted++;
        if (counted == 7)
          throw FoundSeven;
      }
      else if (typeof object[name] == "object") {
        count(object[name]);
      }
    }
  }

  try {
    count(object);
    return false;
  }
  catch (exception) {
    if (exception != FoundSeven)
      throw exception;
    return true;
  }
}
```

The inner function `count` is recursively called for every object that is part of the argument. When the variable `counted` reaches seven, there is no point in continuing to count, but just returning from the current call to `count` will not necessarily stop the counting, since there might be more calls below it. So what we do is just throw a value, which will cause the control to jump right out of any calls to `count`, and land at the `catch` block.

But just returning `true` in case of an exception is not correct. Something else might be going wrong, so we first check whether the exception is the object `FoundSeven`, created specifically for this purpose. If it is not, this `catch` block does not know how to handle it, so it raises it again.

This is a pattern that is also common when dealing with error conditions — you have to make sure that your `catch` block only handles exceptions that it knows how to handle. 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 use unique values, such as the `FoundSeven` object, or to introduce a new type of objects, as described in chapter 8.

Chapter 6:

Functional Programming

As programs get bigger, they also become more complex and harder to understand. We all think ourselves pretty clever, of course, but we are mere human beings, and even a moderate amount of chaos tends to baffle us. And then it all goes downhill. Working on something you do not really understand is a bit like cutting random wires on those time-activated bombs they always have in movies. If you are lucky, you might get the right one — especially if you are the hero of the movie and strike a suitably dramatic pose — but there is always the possibility of blowing everything up.

Admittedly, in most cases, breaking a program does not cause any large explosions. But when a program, by someone's ignorant tinkering, has degenerated into a ramshackle mass of errors, reshaping it into something sensible is a terrible labour — sometimes you might just as well start over.

Thus, the programmer is always looking for ways to keep the complexity of his programs as low as possible. An important way to do this is to try and make code more abstract. When writing a program, it is easy to get sidetracked into small details at every point. You come across some little issue, and you deal with it, and then proceed to the next little problem, and so on. This makes the code read like a grandmother's tale.

> *Yes, dear, to make pea soup you will need split peas, the dry kind. And you have to soak them at least for a night, or you will have to cook them for hours and hours. I remember one time, when my dull son tried to make pea soup. Would you believe he hadn't soaked the peas? We almost broke our teeth, all of us. Anyway, when you have soaked the peas, and you'll want about a cup of them per person, and pay attention because they will expand a bit while they are soaking, so if you aren't careful they will spill out of whatever you use to hold them, so also use plenty water to soak in, but as I said, about a cup of them, when they are dry, and after they are soaked you cook them in four cups of water per cup of dry peas. Let it simmer for two hours, which means you cover it and keep it barely cooking, and then add some diced onions, sliced celery stalk, and maybe a carrot or two and some ham. Let it all cook for a few minutes more, and it is ready to eat.*

Another way to describe this recipe:

> *Per person: one cup dried split peas, half a chopped onion, half a carrot, a celery stalk, and optionally ham.*
>
> *Soak peas overnight, simmer them for two hours in four cups of water (per person), add vegetables and ham, and cook for ten more minutes.*

This is shorter, but if you don't know how to soak peas you'll surely screw up and put them in too little water. But how to soak peas can be looked up, and that is the trick. If you assume a certain basic knowledge in the audience, you can talk in a language that deals with bigger concepts, and express things in a much shorter and clearer way. This, more or less, is what abstraction is.

How is this far-fetched recipe story relevant to programming? Well, obviously, the recipe is the program. Furthermore, the basic knowledge that the cook is supposed to have corresponds to the functions and other constructs that are available to the programmer. If you remember the introduction of this book, things like `while` make it easier to build loops,

and in chapter 4 we wrote some simple functions in order to make other functions shorter and more straightforward. Such tools, some of them made available by the language itself, others built by the programmer, are used to reduce the amount of uninteresting details in the rest of the program, and thus make that programs easier to work with.

Functional programming, which is the subject of this chapter, produces abstraction through clever ways of combining functions. A programmer armed with a repertoire of fundamental functions and, more importantly, the knowledge on how to use them, is much more effective than one who starts from scratch. Unfortunately, a standard JavaScript environment comes with deplorably few essential functions, so we have to write them ourselves or, which is often preferable, make use of somebody else's code (more on that in chapter 9).

There are other popular approaches to abstraction, most notably object-oriented programming, the subject of chapter 8.

One ugly detail that, if you have any good taste at all, must be starting to bother you is the endlessly repeated `for` loop going over an array: `for (var i = 0; i < something.length; i++) ...`. Can this be abstracted?

The problem is that, whereas most functions just take some values, combine them, and return something, such a loop contains a piece of code that it must execute. It is easy to write a function that goes over an array and prints out every element:

```
function printArray(array) {
  for (var i = 0; i < array.length; i++)
    print(array[i]);
}
```

But what if we want to do something else than print? Since 'doing something' can be represented as a function, and functions are also values, we can pass our action as a function value:

```
function forEach(array, action) {
  for (var i = 0; i < array.length; i++)
    action(array[i]);
}

forEach(["Wampeter", "Foma", "Granfalloon"], print);
```

And by making use of an anonymous function, something just like a `for` loop can be written with less useless details:

```
function sum(numbers) {
  var total = 0;
  forEach(numbers, function (number) {
    total += number;
  });
  return total;
}
show(sum([1, 10, 100]));
```

Note that the variable `total` is visible inside the anonymous function because of the lexical scoping rules. Also note that this version is hardly shorter than the `for` loop and requires a rather clunky `});` at its end — the brace closes the body of the anonymous function, the

parenthesis closes the function call to `forEach`, and the semicolon is needed because this call is a statement.

You do get a variable bound to the current element in the array, `number`, so there is no need to use `numbers[i]` anymore, and when this array is created by evaluating some expression, there is no need to store it in a variable, because it can be passed to `forEach` directly.

The cat-code in chapter 4 contains a piece like this:

```
var paragraphs = mailArchive[mail].split("\n");
for (var i = 0; i < paragraphs.length; i++)
  handleParagraph(paragraphs[i]);
```

This can now be written as...

```
forEach(mailArchive[mail].split("\n"), handleParagraph);
```

On the whole, using more abstract (or 'higher level') constructs results in more information and less noise: The code in `sum` reads '*for each number in numbers add that number to the total*', instead of... '*there is this variable that starts at zero, and it counts upward to the length of the array called numbers, and for every value of this variable we look up the corresponding element in the array and add this to the total*'.

What `forEach` does is take an algorithm, in this case 'going over an array', and abstract it. The 'gaps' in the algorithm, in this case, what to do for each of these elements, are filled by functions which are passed to the algorithm function.

Functions that operate on other functions are called higher-order functions. By operating on functions, they can talk about actions on a whole new level. The `makeAddFunction` function from chapter 3 is also a higher-order function. Instead of taking a function value as an argument, it produces a new function.

Higher-order functions can be used to generalise many algorithms that regular functions can not easily describe. When you have a repertoire of these functions at your disposal, it can help you think about your code in a clearer way: Instead of a messy set of variables and loops, you can decompose algorithms into a combination of a few fundamental algorithms, which are invoked by name, and do not have to be typed out again and again.

Being able to write *what* we want to do instead of *how* we do it means we are working at a higher level of abstraction. In practice, this means shorter, clearer, and more pleasant code.

Another useful type of higher-order function *modifies* the function value it is given:

```
function negate(func) {
  return function(x) {
    return !func(x);
  };
}
var isNotNaN = negate(isNaN);
show(isNotNaN(NaN));
```

The function returned by `negate` feeds the argument it is given to the original function

`func`, and then negates the result. But what if the function you want to negate takes more than one argument? You can get access to any arguments passed to a function with the `arguments` array, but how do you call a function when you do not know how many arguments you have?

Functions have a method called `apply`, which is used for situations like this. It takes two arguments. The role of the first argument will be discussed in chapter 8, for now we just use `null` there. The second argument is an array containing the arguments that the function must be applied to.

```
show(Math.min.apply(null, [5, 6]));

function negate(func) {
  return function() {
    return !func.apply(null, arguments);
  };
}
```

Unfortunately, on the Internet Explorer browser a lot of built-in functions, such as `alert`, are not *really* functions... or something. They report their type as `"object"` when given to the `typeof` operator, and they do not have an `apply` method. Your own functions do not suffer from this, they are always real functions.

Let us look at a few more basic algorithms related to arrays. The `sum` function is really a variant of an algorithm which is usually called `reduce` or `fold`:

```
function reduce(combine, base, array) {
  forEach(array, function (element) {
    base = combine(base, element);
  });
  return base;
}

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

function sum(numbers) {
  return reduce(add, 0, numbers);
}
```

`reduce` combines an array into a single value by repeatedly using a function that combines an element of the array with a base value. This is exactly what `sum` did, so it can be made shorter by using `reduce`... except that addition is an operator and not a function in JavaScript, so we first had to put it into a function.

The reason `reduce` takes the function as its first argument instead of its last, as in `forEach`, is partly that this is tradition — other languages do it like that — and partly that this allows us to use a particular trick, which will be discussed at the end of this chapter. It does mean that, when calling `reduce`, writing the reducing function as an anonymous function looks a bit weirder, because now the other arguments follow after the function, and the resemblance to a normal `for` block is lost entirely.

Ex. 6.1 Write a function `countZeroes`, which takes an array of numbers as its argument and returns the amount of zeroes that occur in it. Use `reduce`.

Then, write the higher-order function `count`, which takes an array and a test function as arguments, and returns the amount of elements in the array for which the test function returned `true`. Re-implement `countZeroes` using this function.

```
function countZeroes(array) {
  function counter(total, element) {
    return total + (element === 0 ? 1 : 0);
  }
  return reduce(counter, 0, array);
}
```

The weird part, with the question mark and the colon, uses a new operator. In chapter 2 we have seen unary and binary operators. This one is ternary — it acts on three values. Its effect resembles that of `if`/`else`, except that, where `if` conditionally executes statements, this one conditionally chooses expressions. The first part, before the question mark, is the condition. If this condition is `true`, the expression after the question mark is chosen, `1` in this case. If it is `false`, the part after the colon, `0` in this case, is chosen.

Use of this operator can make some pieces of code much shorter. When the expressions inside it get very big, or you have to make more decisions inside the conditional parts, just using plain `if` and `else` is usually more readable.

Here is the solution that uses a `count` function, with a function that produces equality-testers included to make the final `countZeroes` function even shorter:

```
function count(test, array) {
  return reduce(function(total, element) {
    return total + (test(element) ? 1 : 0);
  }, 0, array);
}

function equals(x) {
  return function(element) {return x === element;};
}

function countZeroes(array) {
  return count(equals(0), array);
}
```

One other generally useful 'fundamental algorithm' related to arrays is called `map`. It goes over an array, applying a function to every element, just like `forEach`. But instead of discarding the values returned by function, it builds up a new array from these values.

```
function map(func, array) {
  var result = [];
  forEach(array, function (element) {
    result.push(func(element));
  });
  return result;
}

show(map(Math.round, [0.01, 2, 9.89, Math.PI]));
```

Note that the first argument is called `func`, not `function`, this is because `function` is a keyword and thus not a valid variable name.

There once was, living in the deep mountain forests of Transylvania, a recluse. Most of the time, he just wandered around his mountain, talking to trees and laughing with birds. But now and then, when the pouring rain trapped him in his little hut, and the howling wind made him feel unbearably small, the recluse felt an urge to write something, wanted to pour some thoughts out onto paper, where they could maybe grow bigger than he himself was.

After failing miserably at poetry, fiction, and philosophy, the recluse finally decided to write a technical book. In his youth, he had done some computer programming, and he figured that if he could just write a good book about that, fame and recognition would surely follow.

So he wrote. At first he used fragments of tree bark, but that turned out not to be very practical. He went down to the nearest village and bought himself a laptop computer. After a few chapters, he realised he wanted to put the book in HTML format, in order to put it on his web-page...

Are you familiar with HTML? It is the method used to add mark-up to pages on the web, and we will be using it a few times in this book, so it would be nice if you know how it works, at least generally. If you are a good student, you could go search the web for a good introduction to HTML now, and come back here when you have read it. Most of you probably are lousy students, so I will just give a short explanation and hope it is enough.

HTML stands for 'HyperText Mark-up Language'. An HTML document is all text. Because it must be able to express the structure of this text, information about which text is a heading, which text is purple, and so on, a few characters have a special meaning, somewhat like backslashes in JavaScript strings. The 'less than' and 'greater than' characters are used to create 'tags'. A tag gives extra information about the text in the document. It can stand on its own, for example to mark the place where a picture should appear in the page, or it can contain text and other tags, for example when it marks the start and end of a paragraph.

Some tags are compulsory, a whole HTML document must always be contained in between `html` tags. Here is an example of an HTML document:

```
<html>
  <head>
    <title>A quote</title>
  </head>
  <body>
    <h1>A quote</h1>
    <blockquote>
      <p>The connection between the language in which we
      think/program and the problems and solutions we can imagine
      is very close.  For this reason restricting language
      features with the intent of eliminating programmer errors is
      at best dangerous.</p>
      <p>-- Bjarne Stroustrup</p>
    </blockquote>
    <p>Mr. Stroustrup is the inventor of the C++ programming
    language, but quite an insightful person nevertheless.</p>
    <p>Also, here is a picture of an ostrich:</p>
    <img src="img/ostrich.png"/>
  </body>
</html>
```

Elements that contain text or other tags are first opened with `<tagname>`, and afterwards

finished with `</tagname>`. The `html` element always contains two children: `head` and `body`. The first contains information *about* the document, the second contains the actual document.

Most tag names are cryptic abbreviations. `h1` stands for 'heading 1', the biggest kind of heading. There are also `h2` to `h6` for successively smaller headings. `p` means 'paragraph', and `img` stands for 'image'. The `img` element does not contain any text or other tags, but it does have some extra information, `src="img/ostrich.png"`, which is called an 'attribute'. In this case, it contains information about the image file that should be shown here.

Because `<` and `>` have a special meaning in HTML documents, they can not be written directly in the text of the document. If you want to say '`5 < 10`' in an HTML document, you have to write '`5 < 10`', where '`lt`' stands for 'less than'. '`>`' is used for '`>`', and because these codes also give the ampersand character a special meaning, a plain '`&`' is written as '`&`'.

Now, those are only the bare basics of HTML, but they should be enough to make it through this chapter, and later chapters that deal with HTML documents, without getting entirely confused.

The JavaScript console has a function `viewHTML` that can be used to look at HTML documents. I stored the example document above in the variable `stroustrupQuote`, so you can view it by executing the following code:

```
viewHTML(stroustrupQuote);
```

If you have some kind of pop-up blocker installed or integrated in your browser, it will probably interfere with `viewHTML`, which tries to show the HTML document in a new window or tab. Try to configure the blocker to allow pop-ups from this site.

So, picking up the story again, the recluse wanted to have his book in HTML format. At first he just wrote all the tags directly into his manuscript, but typing all those less-than and greater-than signs made his fingers hurt, and he constantly forgot to write `&` when he needed an `&`. This gave him a headache. Next, he tried to write the book in Microsoft Word, and then save it as HTML. But the HTML that came out of that was fifteen times bigger and more complicated than it had to be. And besides, Microsoft Word gave him a headache.

The solution that he eventually came up with was this: He would write the book as plain text, following some simple rules about the way paragraphs were separated and the way headings looked. Then, he would write a program to convert this text into precisely the HTML that he wanted.

The rules are this:

1. Paragraphs are separated by blank lines.
2. A paragraph that starts with a '%' symbol is a header. The more '%' symbols, the smaller the header.
3. Inside paragraphs, pieces of text can be emphasised by putting them between asterisks.
4. Footnotes are written between braces.

After he had struggled painfully with his book for six months, the recluse had still only finished a few paragraphs. At this point, his hut was stuck by lightning, killing him, and forever putting his writing ambitions to rest. From the charred remains of his laptop, I could recover the following file:

```
% The Book of Programming

%% The Two Aspects

Below the surface of the machine, the program moves. Without effort,
it expands and contracts. In great harmony, electrons scatter and
regroup. The forms on the monitor are but ripples on the water. The
essence stays invisibly below.

When the creators built the machine, they put in the processor and the
memory. From these arise the two aspects of the program.

The aspect of the processor is the active substance. It is called
Control. The aspect of the memory is the passive substance. It is
called Data.

Data is made of merely bits, yet it takes complex forms. Control
consists only of simple instructions, yet it performs difficult
tasks. From the small and trivial, the large and complex arise.

The program source is Data. Control arises from it. The Control
proceeds to create new Data. The one is born from the other, the
other is useless without the one. This is the harmonious cycle of
Data and Control.

Of themselves, Data and Control are without structure. The programmers
of old moulded their programs out of this raw substance. Over time,
the amorphous Data has crystallised into data types, and the chaotic
Control was restricted into control structures and functions.

%% Short Sayings

When a student asked Fu-Tzu about the nature of the cycle of Data and
Control, Fu-Tzu replied 'Think of a compiler, compiling itself.'

A student asked 'The programmers of old used only simple machines and
no programming languages, yet they made beautiful programs. Why do we
use complicated machines and programming languages?'. Fu-Tzu replied
'The builders of old used only sticks and clay, yet they made
beautiful huts.'

A hermit spent ten years writing a program. 'My program can compute
the motion of the stars on a 286-computer running MS DOS', he proudly
announced. 'Nobody owns a 286-computer or uses MS DOS anymore.',
Fu-Tzu responded.

Fu-Tzu had written a small program that was full of global state and
dubious shortcuts. Reading it, a student asked 'You warned us against
these techniques, yet I find them in your program. How can this be?'
Fu-Tzu said 'There is no need to fetch a water hose when the house is
not on fire.'{This is not to be read as an encouragement of sloppy
programming, but rather as a warning against neurotic adherence to
rules of thumb.}

%% Wisdom

A student was complaining about digital numbers. 'When I take the root
of two and then square it again, the result is already inaccurate!'.
Overhearing him, Fu-Tzu laughed. 'Here is a sheet of paper. Write down
the precise value of the square root of two for me.'

Fu-Tzu said 'When you cut against the grain of the wood, much strength
is needed. When you program against the grain of a problem, much code
is needed.'

Tzu-li and Tzu-ssu were boasting about the size of their latest
```

```
programs. 'Two-hundred thousand lines', said Tzu-li, 'not counting
comments!'. 'Psah', said Tzu-ssu, 'mine is almost a *million* lines
already.' Fu-Tzu said 'My best program has five hundred lines.'
Hearing this, Tzu-li and Tzu-ssu were enlightened.

A student had been sitting motionless behind his computer for hours,
frowning darkly. He was trying to write a beautiful solution to a
difficult problem, but could not find the right approach. Fu-Tzu hit
him on the back of his head and shouted '*Type something!*' The student
started writing an ugly solution. After he had finished, he suddenly
understood the beautiful solution.

%% Progression

A beginning programmer writes his programs like an ant builds her
hill, one piece at a time, without thought for the bigger structure.
His programs will be like loose sand. They may stand for a while, but
growing too big they fall apart{Referring to the danger of internal
inconsistency and duplicated structure in unorganised code.}.

Realising this problem, the programmer will start to spend a lot of
time thinking about structure. His programs will be rigidly
structured, like rock sculptures. They are solid, but when they must
change, violence must be done to them{Referring to the fact that
structure tends to put restrictions on the evolution of a program.}.

The master programmer knows when to apply structure and when to leave
things in their simple form. His programs are like clay, solid yet
malleable.

%% Language

When a programming language is created, it is given syntax and
semantics. The syntax describes the form of the program, the semantics
describe the function. When the syntax is beautiful and the semantics
are clear, the program will be like a stately tree. When the syntax is
clumsy and the semantics confusing, the program will be like a bramble
bush.

Tzu-ssu was asked to write a program in the language called Java,
which takes a very primitive approach to functions. Every morning, as
he sat down in front of his computer, he started complaining. All day
he cursed, blaming the language for all that went wrong. Fu-Tzu
listened for a while, and then reproached him, saying 'Every language
has its own way. Follow its form, do not try to program as if you
were using another language.'
```

To honour the memory of our good recluse, I would like to finish his HTML-generating program for him. A good approach to this problem goes like this:

1. Split the file into paragraphs by cutting it at every empty line.
2. Remove the '%' characters from header paragraphs and mark them as headers.
3. Process the text of the paragraphs themselves, splitting them into normal parts, emphasised parts, and footnotes.
4. Move all the footnotes to the bottom of the document, leaving numbers[1] in their place.
5. Wrap each piece into the correct HTML tags.
6. Combine everything into a single HTML document.

This approach does not allow footnotes inside emphasised text, or vice versa. This is kind of arbitrary, but helps keep the example code simple. If, at the end of the chapter, you feel like an extra challenge, you can try to revise the program to support 'nested' mark-up.

The whole manuscript, as a string value, is available on this page by calling `recluseFile` function.

Step 1 of the algorithm is trivial. A blank line is what you get when you have two newlines in a row, and if you remember the `split` method that strings have, which we saw in chapter 4, you will realise that this will do the trick:

```
var paragraphs = recluseFile().split("\n\n");
print("Found ", paragraphs.length, " paragraphs.");
```

Ex. 6.2

Write a function `processParagraph` that, when given a paragraph string as its argument, checks whether this paragraph is a header. If it is, it strips of the '%' characters and counts their number. Then, it returns an object with two properties, `content`, which contains the text inside the paragraph, and `type`, which contains the tag that this paragraph must be wrapped in, `"p"` for regular paragraphs, `"h1"` for headers with one '%', and `"hX"` for headers with `X` '%' characters.

Remember that strings have a `charAt` method that can be used to look at a specific character inside them.

```
function processParagraph(paragraph) {
  var header = 0;
  while (paragraph.charAt(0) == "%") {
    paragraph = paragraph.slice(1);
    header++;
  }

  return {type: (header == 0 ? "p" : "h" + header),
          content: paragraph};
}

show(processParagraph(paragraphs[0]));
```

This is where we can try out the `map` function we saw earlier.

```
var paragraphs = map(processParagraph,
                     recluseFile().split("\n\n"));
```

And *bang*, we have an array of nicely categorised paragraph objects. We are getting ahead of ourselves though, we forgot step 3 of the algorithm:

> *Process the text of the paragraphs themselves, splitting them into normal parts, emphasised parts, and footnotes.*

Which can be decomposed into:

1. If the paragraph starts with an asterisk, take off the emphasised part and store it.
2. If the paragraph starts with an opening brace, take off the footnote and store it.
3. Otherwise, take off the part until the first emphasised part or footnote, or until the end of the string, and store it as normal text.
4. If there is anything left in the paragraph, start at 1 again.

Ex. 6.3 Build a function `splitParagraph` which, given a paragraph string, returns an array of paragraph fragments. Think of a good way to represent the fragments.

The method `indexOf`, which searches for a character or sub-string in a string and returns its position, or `-1` if not found, will probably be useful in some way here.

This is a tricky algorithm, and there are many not-quite-correct or way-too-long ways to describe it. If you run into problems, just think about it for a minute. Try to write inner functions that perform the smaller actions that make up the algorithm.

Here is one possible solution:

```
function splitParagraph(text) {
  function indexOrEnd(character) {
    var index = text.indexOf(character);
    return index == -1 ? text.length : index;
  }

  function takeNormal() {
    var end = reduce(Math.min, text.length,
                     map(indexOrEnd, ["*", "{"]));
    var part = text.slice(0, end);
    text = text.slice(end);
    return part;
  }

  function takeUpTo(character) {
    var end = text.indexOf(character, 1);
    if (end == -1)
      throw new Error("Missing closing '" + character + "'");
    var part = text.slice(1, end);
    text = text.slice(end + 1);
    return part;
  }

  var fragments = [];

  while (text != "") {
    if (text.charAt(0) == "*")
      fragments.push({type: "emphasised",
                      content: takeUpTo("*")});
    else if (text.charAt(0) == "{")
      fragments.push({type: "footnote",
                      content: takeUpTo("}")});
    else
      fragments.push({type: "normal",
                      content: takeNormal()});
  }
  return fragments;
}
```

Note the over-eager use of `map` and `reduce` in the `takeNormal` function. This is a chapter about functional programming, so program functionally we will! Can you see how this works? The `map` produces an array of positions where the given characters were found, or the end of the string if they were not found, and the `reduce` takes the minimum of them, which is the next point in the string that we have to look at.

If you'd write that out without mapping and reducing you'd get something like this:

```
var nextAsterisk = text.indexOf("*");
var nextBrace = text.indexOf("{");
var end = text.length;
if (nextAsterisk != -1)
  end = nextAsterisk;
if (nextBrace != -1 && nextBrace < end)
  end = nextBrace;
```

Which is even more hideous. Most of the time, when a decision has to be made based on a series of things, even if there are only two of them, writing it as array operations is nicer than handling every value in a separate `if` statement. (Fortunately, chapter 10 describes an easier way to ask for the first occurrence of 'this or that character' in a string.)

If you wrote a `splitParagraph` that stored fragments in a different way than the solution above, you might want to adjust it, because the functions in the rest of the chapter assume that fragments are objects with `type` and `content` properties.

We can now wire `processParagraph` to also split the text inside the paragraphs, my version can be modified like this:

```
function processParagraph(paragraph) {
  var header = 0;
  while (paragraph.charAt(0) == "%") {
    paragraph = paragraph.slice(1);
    header++;
  }

  return {type: (header == 0 ? "p" : "h" + header),
          content: splitParagraph(paragraph)};
}
```

Mapping that over the array of paragraphs gives us an array of paragraph objects, which in turn contain arrays of fragment objects. The next thing to do is to take out the footnotes, and put references to them in their place. Something like this:

```
function extractFootnotes(paragraphs) {
  var footnotes = [];
  var currentNote = 0;

  function replaceFootnote(fragment) {
    if (fragment.type == "footnote") {
      currentNote++;
      footnotes.push(fragment);
      fragment.number = currentNote;
      return {type: "reference", number: currentNote};
    }
    else {
      return fragment;
    }
  }

  forEach(paragraphs, function(paragraph) {
    paragraph.content = map(replaceFootnote,
                            paragraph.content);
  });

  return footnotes;
}
```

The `replaceFootnote` function is called on every fragment. When it gets a fragment that should stay where it is, it just returns it, but when it gets a footnote, it stores this footnote in the `footnotes` array, and returns a reference to it instead. In the process, every footnote and reference is also numbered.

That gives us enough tools to extract the information we need from the file. All that is left now is generating the correct HTML.

A lot of people think that concatenating strings is a great way to produce HTML. When they need a link to, for example, a site where you can play the game of Go, they will do:

```
var url = "http://www.gokgs.com/";
var text = "Play Go!";
var linkText = "<a href=\"" + url + "\">" + text + "</a>";
print(linkText);
```

(Where `a` is the tag used to create links in HTML documents.) ... Not only is this clumsy, but when the string `text` happens to include an angular bracket or an ampersand, it is also wrong. Weird things will happen on your website, and you will look embarrassingly amateurish. We wouldn't want that to happen. A few simple HTML-generating functions are easy to write. So let us write them.

The secret to successful HTML generation is to treat your HTML document as a data structure instead of a flat piece of text. JavaScript's objects provide a very easy way to model this:

```
var linkObject = {name: "a",
                  attributes: {href: "http://www.gokgs.com/"},
                  content: ["Play Go!"]};
```

Each HTML element contains a `name` property, giving the name of the tag it represents. When it has attributes, it also contains an `attributes` property, which contains an object in which the attributes are stored. When it has content, there is a `content` property, containing an array of other elements contained in this element. Strings play the role of pieces of text in our HTML document, so the array `["Play Go!"]` means that this link has only one element inside it, which is a simple piece of text.

Typing in these objects directly is clumsy, but we don't have to do that. We provide a shortcut function to do this for us:

```
function tag(name, content, attributes) {
  return {name: name, attributes: attributes, content: content};
}
```

Note that, since we allow the `attributes` and `content` of an element to be undefined if they are not applicable, the second and third argument to this function can be left off when they are not needed.

`tag` is still rather primitive, so we write shortcuts for common types of elements, such as links, or the outer structure of a simple document:

```
function link(target, text) {
  return tag("a", [text], {href: target});
}

function htmlDoc(title, bodyContent) {
  return tag("html", [tag("head", [tag("title", [title])]),
                      tag("body", bodyContent)]);
}
```

Ex. 6.4 Looking back at the example HTML document if necessary, write an `image` function which, when given the location of an image file, will create an `img` HTML element.

```
function image(src) {
  return tag("img", [], {src: src});
}
```

When we have created a document, it will have to be reduced to a string. But building this string from the data structures we have been producing is very straightforward. The important thing is to remember to transform the special characters in the text of our document...

```
function escapeHTML(text) {
  var replacements = [[/&/g, "&amp;"], [/"/g, "&quot;"],
                      [/</g, "&lt;"], [/>/g, "&gt;"]];
  forEach(replacements, function(replace) {
    text = text.replace(replace[0], replace[1]);
  });
  return text;
}
```

The `replace` method of strings creates a new string in which all occurrences of the pattern in the first argument are replaced by the second argument, so `"Borobudur".replace(/r/g, "k")` gives `"Bokobuduk"`. Don't worry about the pattern syntax here — we'll get to that in chapter 10. The `escapeHTML` function puts the different replacements that have to be made into an array, so that it can loop over them and apply them to the argument one by one.

Double quotes are also replaced, because we will also be using this function for the text inside the attributes of HTML tags. Those will be surrounded by double quotes, and thus must not have any double quotes inside of them.

Calling replace four times means the computer has to go over the whole string four times to check and replace its content. This is not very efficient. If we cared enough, we could write a more complex version of this function, something that resembles the `splitParagraph` function we saw earlier, to go over it only once. For now, we are too lazy for this. Again, chapter 10 shows a much better way to do this.

To turn an HTML element object into a string, we can use a recursive function like this:

```
function renderHTML(element) {
  var pieces = [];

  function renderAttributes(attributes) {
    var result = [];
    if (attributes) {
      for (var name in attributes)
        result.push(" " + name + "=\"" +
                    escapeHTML(attributes[name]) + "\"");
    }
    return result.join("");
  }

  function render(element) {
    // Text node
    if (typeof element == "string") {
      pieces.push(escapeHTML(element));
    }
    // Empty tag
    else if (!element.content || element.content.length == 0) {
      pieces.push("<" + element.name +
                  renderAttributes(element.attributes) + "/>");
    }
    // Tag with content
    else {
      pieces.push("<" + element.name +
                  renderAttributes(element.attributes) + ">");
      forEach(element.content, render);
      pieces.push("</" + element.name + ">");
    }
  }

  render(element);
  return pieces.join("");
}
```

Note the `in` loop that extracts the properties from a JavaScript object in order to make HTML tag attributes out of them. Also note that in two places, arrays are being used to accumulate strings, which are then joined into a single result string. Why didn't I just start with an empty string and then add the content to it with the `+=` operator?

It turns out that creating new strings, especially big strings, is quite a lot of work. Remember that JavaScript string values never change. If you concatenate something to them, a new string is created, the old ones stay intact. If we build up a big string by concatenating lots of little strings, new strings have to be created at every step, only to be thrown away when the next piece is concatenated to them. If, on the other hand, we store all the little strings in an array and then join them, only *one* big string has to be created.

So, let us try out this HTML generating system...

```
print(renderHTML(link("http://www.nedroid.com", "Drawings!")));
```

That seems to work.

```
var body = [tag("h1", ["The Test"]),
            tag("p", ["Here is a paragraph, and an image..."]),
            image("img/sheep.png")];
var doc = htmlDoc("The Test", body);
viewHTML(renderHTML(doc));
```

Now, I should probably warn you that this approach is not perfect. What it actually renders is XML, which is similar to HTML, but more structured. In simple cases, such as the above, this does not cause any problems. However, there are some things, which are correct XML, but not proper HTML, and these might confuse a browser that is trying to show the documents we create. For example, if you have an empty `script` tag (used to put JavaScript into a page) in your document, browsers will not realise that it is empty and think that everything after it is JavaScript. (In this case, the problem can be fixed by putting a single space inside of the tag, so that it is no longer empty, and gets a proper closing tag.)

Ex. 6.5

Write a function `renderFragment`, and use that to implement another function `renderParagraph`, which takes a paragraph object (with the footnotes already filtered out), and produces the correct HTML element (which might be paragraph or a header, depending on the `type` property of the paragraph object).

This function might come in useful for rendering the footnote references:

```
function footnote(number) {
  return tag("sup", [link("#footnote" + number,
                          String(number))]);
}
```

A `sup` tag will show its content as 'superscript', which means it will be smaller and a little higher than other text. The target of the link will be something like `"#footnote1"`. Links that contain a '#' character refer to 'anchors' within a page, and in this case we will use them to make it so that clicking on the footnote link will take the reader to the bottom of the page, where the footnotes live.

The tag to render emphasised fragments with is `em`, and normal text can be rendered without any extra tags.

```
function renderParagraph(paragraph) {
  return tag(paragraph.type, map(renderFragment,
                                 paragraph.content));
}

function renderFragment(fragment) {
  if (fragment.type == "reference")
    return footnote(fragment.number);
  else if (fragment.type == "emphasised")
    return tag("em", [fragment.content]);
  else if (fragment.type == "normal")
    return fragment.content;
}
```

We are almost finished. The only thing that we do not have a rendering function for yet are the footnotes. To make the `"#footnote1"` links work, an anchor must be included with every footnote. In HTML, an anchor is specified with an `a` element, which is also used for links. In this case, it needs a `name` attribute, instead of an `href`.

```
function renderFootnote(footnote) {
  var anchor = tag("a", [], {name: "footnote" + footnote.number});
  var number = "[" + footnote.number + "] ";
  return tag("p", [tag("small", [anchor, number,
                                 footnote.content])]);
}
```

Here, then, is the function which, when given a file in the correct format and a document title, returns an HTML document:

```
function renderFile(file, title) {
  var paragraphs = map(processParagraph, file.split("\n\n"));
  var footnotes = map(renderFootnote,
                      extractFootnotes(paragraphs));
  var body = map(renderParagraph, paragraphs).concat(footnotes);
  return renderHTML(htmlDoc(title, body));
}

viewHTML(renderFile(recluseFile(), "The Book of Programming"));
```

The `concat` method of an array can be used to concatenate another array to it, similar to what the `+` operator does with strings.

In the chapters after this one, elementary higher-order functions like `map` and `reduce` will always be available and will be used by code examples. Now and then, a new useful tool is added to this. In chapter 9, we develop a more structured approach to this set of 'basic' functions.

When using higher-order functions, it is often annoying that operators are not functions in JavaScript. We have needed `add` or `equals` functions at several points. Rewriting these every time, you will agree, is a pain. From now on, we will assume the existence of an object called `op`, which contains these functions:

```
var op = {
  "+": function(a, b){return a + b;},
  "==": function(a, b){return a == b;},
  "===": function(a, b){return a === b;},
  "!": function(a){return !a;}
  /* and so on */
};
```

So we can write `reduce(op["+"], 0, [1, 2, 3, 4, 5])` to sum an array. But what if we need something like `equals` or `makeAddFunction`, in which one of the arguments already has a value? In that case we are back to writing a new function again.

For cases like that, something called 'partial application' is useful. You want to create a new function that already knows some of its arguments, and treats any additional arguments it is passed as coming after these fixed arguments. This can be done by making creative use of the `apply` method of a function:

```
function asArray(quasiArray, start) {
  var result = [];
  for (var i = (start || 0); i < quasiArray.length; i++)
    result.push(quasiArray[i]);
  return result;
}

function partial(func) {
  var fixedArgs = asArray(arguments, 1);
  return function(){
    return func.apply(null, fixedArgs.concat(asArray(arguments)));
  };
}
```

We want to allow binding multiple arguments at the same time, so the `asArray` function is necessary to make normal arrays out of the `arguments` objects. It copies their content into a real array, so that the `concat` method can be used on it. It also takes an optional second argument, which can be used to leave out some arguments at the start.

Also note that it is necessary to store the `arguments` of the outer function (`partial`) into a variable with another name, because otherwise the inner function can not see them — it has its own `arguments` variable, which shadows the one of the outer function.

Now `equals(10)` can be written as `partial(op["=="], 10)`.

```
show(map(partial(op["+"], 1), [0, 2, 4, 6, 8, 10]));
```

The reason `map` takes its function argument before its array argument is that it is often useful to partially apply map by giving it a function. This 'lifts' the function from operating on a single value to operating on an array of values. For example, if you have an array of arrays of numbers, and you want to square them all, you do this:

```
function square(x) {return x * x;}

show(map(partial(map, square), [[10, 100], [12, 16], [0, 1]]));
```

One last trick that can be useful when you want to combine functions is function composition. At the start of this chapter I showed a function `negate`, which applies the boolean *not* operator to the result of calling a function:

```
function negate(func) {
  return function() {
    return !func.apply(null, arguments);
  };
}
```

This is a special case of a general pattern: call function A, and then apply function B to the result. Compositions is a common concept in mathematics. It can be caught in a higher-order function like this:

```
function compose(func1, func2) {
  return function() {
    return func1(func2.apply(null, arguments));
  };
}

var isUndefined = partial(op["==="], undefined);
var isDefined = compose(op["!"], isUndefined);
show(isDefined(Math.PI));
show(isDefined(Math.PIE));
```

Here we are defining new functions without using the `function` keyword at all. This can be useful when you need to create a simple function to give to, for example, `map` or `reduce`. However, when a function becomes more complex than these examples, it is usually shorter (not to mention more efficient) to just write it out with `function`.

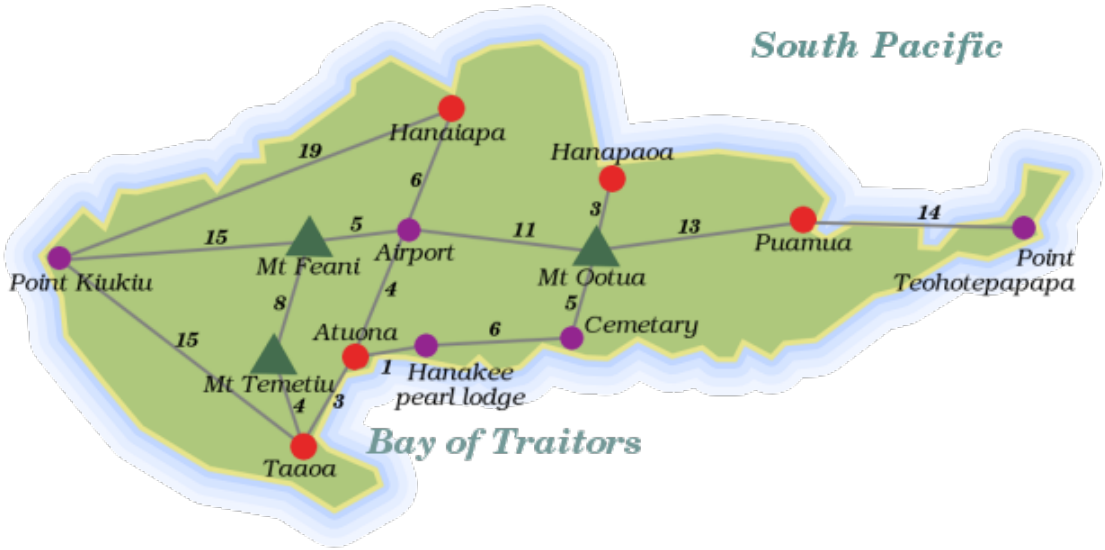
1. Like this...

Chapter 7:

Searching

This chapter does not introduce any new JavaScript-specific concepts. Instead, we will go through the solution to two problems, discussing some interesting algorithms and techniques along the way. If this does not sound interesting to you, it is safe to skip to the next chapter.

Let me introduce our first problem. Take a look at this map. It shows Hiva Oa, a small tropical island in the Pacific Ocean.



The grey lines are roads, and the numbers next to them are the lengths of these roads. Imagine we need a program that finds the shortest route between two points on Hiva Oa. How could we approach that? Think about this for a moment.

No really. Don't just steamroll on to the next paragraph. Try to seriously think of some ways you could do this, and consider the issues you would come up against. When reading a technical book, it is way too easy to just zoom over the text, nod solemnly, and promptly forget what you have read. If you make a sincere effort to solve a problem, it becomes *your* problem, and its solution will be more meaningful.

The first aspect of this problem is, again, representing our data. The information in the picture does not mean much to our computer. We could try writing a program that looks at the map and extracts the information in it... but that can get complicated. If we had twenty-thousand maps to interpret, this would be a good idea, in this case we will do the interpretation ourself and transcribe the map into a more computer-friendly format.

What does our program need to know? It has to be able to look up which locations are connected, and how long the roads between them are. The places and roads on the island form a graph, as mathematicians call it. There are many ways to store graphs. A simple possibility is to just store an array of road objects, each of which contains properties naming its two endpoints and its length...

```
var roads = [{point1: "Point Kiukiu", point2: "Hanaiapa", length: 19},
             {point1: "Point Kiukiu", point2: "Mt Feani", length: 15}
             /* and so on */];
```

However, it turns out that the program, as it is working out a route, will very often need to get a list of all the roads that start at a certain location, like a person standing on a crossroads will look at a signpost and read "Hanaiapa: 19km, Mount Feani: 15km". It would be nice if this was easy (and quick) to do.

With the representation given above, we have to sift through the whole list of roads, picking out the relevant ones, every time we want this signpost list. A better approach would be to store this list directly. For example, use an object that associates place-names with signpost lists:

```
var roads = {"Point Kiukiu": [{to: "Hanaiapa", distance: 19},
                              {to: "Mt Feani", distance: 15},
                              {to: "Taaoa", distance: 15}],
             "Taaoa": [/* et cetera */]};
```

When we have this object, getting the roads that leave from Point Kiukiu is just a matter of looking at `roads["Point Kiukiu"]`.

However, this new representation does contain duplicate information: The road between A and B is listed both under A and under B. The first representation was already a lot of work to type in, this one is even worse.

Fortunately, we have at our command the computer's talent for repetitive work. We can specify the roads once, and have the correct data structure be generated by the computer. First, initialise an empty object called `roads`, and write a function `makeRoad`:

```
var roads = {};
function makeRoad(from, to, length) {
  function addRoad(from, to) {
    if (!(from in roads))
      roads[from] = [];
    roads[from].push({to: to, distance: length});
  }
  addRoad(from, to);
  addRoad(to, from);
}
```

Nice, huh? Notice how the inner function, `addRoad`, uses the same names (`from`, `to`) for its parameters as the outer function. These will not interfere: inside `addRoad` they refer to `addRoad`'s parameters, and outside it they refer to `makeRoad`'s parameters.

The `if` statement in `addRoad` makes sure that there is an array of destinations associated with the location named by `from`, if there isn't already one it puts in an empty array. This way, the next line can assume there is such an array and safely push the new road onto it.

Now the map information looks like this:

```
makeRoad("Point Kiukiu", "Hanaiapa", 19);
makeRoad("Point Kiukiu", "Mt Feani", 15);
makeRoad("Point Kiukiu", "Taaoa", 15);
// ...
```

Ex. 7.1

In the above description, the string `"Point Kiukiu"` still occurs three times in a row. We could make our description even more succinct by allowing multiple roads to be specified in one line.

Write a function `makeRoads` that takes any uneven number of arguments. The first argument is always the starting point of the roads, and every pair of arguments after that gives an ending point and a distance.

Do not duplicate the functionality of `makeRoad`, but have `makeRoads` call `makeRoad` to do the actual road-making.

```
function makeRoads(start) {
  for (var i = 1; i < arguments.length; i += 2)
    makeRoad(start, arguments[i], arguments[i + 1]);
}
```

This function uses one named parameter, `start`, and gets the other parameters from the `arguments` (quasi-) array. `i` starts at `1` because it has to skip this first parameter. `i += 2` is short for `i = i + 2`, as you might recall.

```
var roads = {};
makeRoads("Point Kiukiu", "Hanaiapa", 19,
          "Mt Feani", 15, "Taaoa", 15);
makeRoads("Airport", "Hanaiapa", 6, "Mt Feani", 5,
          "Atuona", 4, "Mt Ootua", 11);
makeRoads("Mt Temetiu", "Mt Feani", 8, "Taaoa", 4);
makeRoads("Atuona", "Taaoa", 3, "Hanakee pearl lodge", 1);
makeRoads("Cemetery", "Hanakee pearl lodge", 6, "Mt Ootua", 5);
makeRoads("Hanapaoa", "Mt Ootua", 3);
makeRoads("Puamua", "Mt Ootua", 13, "Point Teohotepapapa", 14);

show(roads["Airport"]);
```

We managed to considerably shorten our description of the road-information by defining some convenient operations. You could say we expressed the information more succinctly by expanding our vocabulary. Defining a 'little language' like this is often a very powerful technique — when, at any time, you find yourself writing repetitive or redundant code, stop and try to come up with a vocabulary that makes it shorter and denser.

Redundant code is not only a bore to write, it is also error-prone, people pay less attention when doing something that doesn't require them to think. On top of that, repetitive code is hard to change, because structure that is repeated a hundred times has to be changed a hundred times when it turns out to be incorrect or suboptimal.

If you ran all the pieces of code above, you should now have a variable named `roads` that contains all the roads on the island. When we need the roads starting from a certain place, we could just do `roads[place]`. But then, when someone makes a typo in a place name, which is not unlikely with these names, he will get `undefined` instead of the array he expects, and strange errors will follow. Instead, we will use a function that retrieves the road arrays, and yells at us when we give it an unknown place name:

```
function roadsFrom(place) {
  var found = roads[place];
  if (found == undefined)
    throw new Error("No place named '" + place + "' found.");
  else
    return found;
}

show(roadsFrom("Puamua"));
```

Here is a first stab at a path-finding algorithm, the gambler's method:

```
function gamblerPath(from, to) {
  function randomInteger(below) {
    return Math.floor(Math.random() * below);
  }
  function randomDirection(from) {
    var options = roadsFrom(from);
    return options[randomInteger(options.length)].to;
  }

  var path = [];
  while (true) {
    path.push(from);
    if (from == to)
      break;
    from = randomDirection(from);
  }
  return path;
}

show(gamblerPath("Hanaiapa", "Mt Feani"));
```

At every split in the road, the gambler rolls his dice to decide which road he shall take. If the dice sends him back the way he came, so be it. Sooner or later, he will arrive at his destination, since all places on the island are connected by roads.

The most confusing line is probably the one containing `Math.random`. This function returns a pseudo-random[1] number between 0 and 1. Try calling it a few times from the console, it will (most likely) give you a different number every time. The function `randomInteger` multiplies this number by the argument it is given, and rounds the result down with `Math.floor`. Thus, for example, `randomInteger(3)` will produce the number `0`, `1`, or `2`.

The gambler's method is the way to go for those who abhor structure and planning, who desperately search for adventure. We set out to write a program that could find the *shortest* route between places though, so something else will be needed.

A very straightforward approach to solving such a problem is called 'generate and test'. It goes like this:

1. Generate all possible routes.
2. In this set, find the shortest one that actually connects the start point to the end point.

Step two is not hard. Step one is a little problematic. If you allow routes with circles in them, there is an infinite amount of routes. Of course, routes with circles in them are unlikely to be the shortest route to anywhere, and routes that do not start at the start

point do not have to be considered either. For a small graph like Hiva Oa, it should be possible to generate all non-cyclic (circle-free) routes starting from a certain point.

But first, we will need some new tools. The first is a function named `member`, which is used to determine whether an element is found within an array. The route will be kept as an array of names, and when arriving at a new place, the algorithm calls `member` to check whether we have been at that place already. It could look like this:

```
function member(array, value) {
  var found = false;
  forEach(array, function(element) {
    if (element === value)
      found = true;
  });
  return found;
}

print(member([6, 7, "Bordeaux"], 7));
```

However, this will go over the whole array, even if the value is found immediately at the first position. What wastefulness. When using a `for` loop, you can use the `break` statement to jump out of it, but in a `forEach` construct this will not work, because the body of the loop is a function, and `break` statements do not jump out of functions. One solution could be to adjust `forEach` to recognise a certain kind of exceptions as signalling a break.

```
var Break = {toString: function() {return "Break";}};

function forEach(array, action) {
  try {
    for (var i = 0; i < array.length; i++)
      action(array[i]);
  }
  catch (exception) {
    if (exception != Break)
      throw exception;
  }
}
```

Now, if the `action` function throws `Break`, `forEach` will absorb the exception and stop looping. The object stored in the variable `Break` is used purely as a thing to compare with. The only reason I gave it a `toString` property is that this might be useful to figure out what kind of strange value you are dealing with if you somehow end up with a `Break` exception outside of a `forEach`.

Having a way to break out of `forEach` loops can be very useful, but in the case of the `member` function the result is still rather ugly, because you need to specifically store the result and later return it. We could add yet another kind of exception, `Return`, which can be given a `value` property, and have `forEach` return this value when such an exception is thrown, but this would be terribly ad-hoc and messy. What we really need is a whole new higher-order function, called `any` (or sometimes `some`). It looks like this:

```
function any(test, array) {
  for (var i = 0; i < array.length; i++) {
    var found = test(array[i]);
    if (found)
      return found;
  }
  return false;
}

function member(array, value) {
  return any(partial(op["==="], value), array);
}

print(member(["Fear", "Loathing"], "Denial"));
```

`any` goes over the elements in an array, from left to right, and applies the test function to them. The first time this returns a true-ish value, it returns that value. If no true-ish value is found, `false` is returned. Calling `any(test, array)` is more or less equivalent to doing `test(array[0]) || test(array[1]) || ...` etcetera.

Just like `&&` is the companion of `||`, `any` has a companion called `every`:

```
function every(test, array) {
  for (var i = 0; i < array.length; i++) {
    var found = test(array[i]);
    if (!found)
      return found;
  }
  return true;
}

show(every(partial(op["!="], 0), [1, 2, -1]));
```

Another function we will need is `flatten`. This function takes an array of arrays, and puts the elements of the arrays together in one big array.

```
  function flatten(arrays) {
    var result = [];
    forEach(arrays, function (array) {
      forEach(array, function (element){result.push(element);});
    });
    return result;
  }
```

The same could have been done using the `concat` method and some kind of `reduce`, but this would be less efficient. Just like repeatedly concatenating strings together is slower than putting them into an array and then calling `join`, repeatedly concatenating arrays produces a lot of unnecessary intermediary array values.

Ex. 7.2

Before starting to generate routes, we need one more higher-order function. This one is called `filter`. Like `map`, it takes a function and an array as arguments, and produces a new array, but instead of putting the results of calling the function in the new array, it produces an array with only those values from the old array for which the given function returns a true-like value. Write this function.

```
function filter(test, array) {
  var result = [];
  forEach(array, function (element) {
    if (test(element))
      result.push(element);
  });
  return result;
}

show(filter(partial(op[">"], 5), [0, 4, 8, 12]));
```

(If the result of that application of `filter` surprises you, remember that the argument given to `partial` is used as the *first* argument of the function, so it ends up to the left of the `>`.)

Imagine what an algorithm to generate routes would look like — it starts at the starting location, and starts to generate a route for every road leaving there. At the end of each of these roads it continues to generate more routes. It doesn't run along one road, it branches out. Because of this, recursion is a natural way to model it.

```
function possibleRoutes(from, to) {
  function findRoutes(route) {
    function notVisited(road) {
      return !member(route.places, road.to);
    }
    function continueRoute(road) {
      return findRoutes({places: route.places.concat([road.to]),
                         length: route.length + road.distance});
    }

    var end = route.places[route.places.length - 1];
    if (end == to)
      return [route];
    else
      return flatten(map(continueRoute, filter(notVisited,
                                               roadsFrom(end))));
  }
  return findRoutes({places: [from], length: 0});
}

show(possibleRoutes("Point Teohotepapapa", "Point Kiukiu").length);
show(possibleRoutes("Hanapaoa", "Mt Ootua"));
```

The function returns an array of route objects, each of which contains an array of places that the route passes, and a length. `findRoutes` recursively continues a route, returning an array with every possible extension of that route. When the end of a route is the place where we want to go, it just returns that route, since continuing past that place would be pointless. If it is another place, we must go on. The `flatten`/`map`/`filter` line is probably the hardest to read. This is what it says: 'Take all the roads going from the current location, discard the ones that go to places that this route has already visited. Continue each of these roads, which will give an array of finished routes for each of them, then put all these routes into a single big array that we return.'

That line does a lot. This is why good abstractions help: They allow you to say complicated things without typing screenfulls of code.

Doesn't this recurse forever, seeing how it keeps calling itself (via `continueRoute`)? No, at some point, all outgoing roads will go to places that a route has already passed, and the

result of `filter` will be an empty array. Mapping over an empty array produces an empty array, and flattening that still gives an empty array. So calling `findRoutes` on a dead end produces an empty array, meaning 'there are no ways to continue this route'.

Notice that places are appended to routes by using `concat`, not `push`. The `concat` method creates a new array, while `push` modifies the existing array. Because the function might branch off several routes from a single partial route, we must not modify the array that represents the original route, because it must be used several times.

Ex. 7.3 Now that we have all possible routes, let us try to find the shortest one. Write a function `shortestRoute` that, like `possibleRoutes`, takes the names of a starting and ending location as arguments. It returns a single route object, of the type that `possibleRoutes` produces.

```
function shortestRoute(from, to) {
  var currentShortest = null;
  forEach(possibleRoutes(from, to), function(route) {
    if (!currentShortest || currentShortest.length > route.length)
      currentShortest = route;
  });
  return currentShortest;
}
```

The tricky thing in 'minimising' or 'maximising' algorithms is to not screw up when given an empty array. In this case, we happen to know that there is at least one road between every two places, so we could just ignore it. But that would be a bit lame. What if the road from Puamua to Mount Ootua, which is steep and muddy, is washed away by a rainstorm? It would be a shame if this caused our function to break as well, so it takes care to return `null` when no routes are found.

Then, the very functional, abstract-everything-we-can approach:

```
function minimise(func, array) {
  var minScore = null;
  var found = null;
  forEach(array, function(element) {
    var score = func(element);
    if (minScore == null || score < minScore) {
      minScore = score;
      found = element;
    }
  });
  return found;
}

function getProperty(propName) {
  return function(object) {
    return object[propName];
  };
}

function shortestRoute(from, to) {
  return minimise(getProperty("length"), possibleRoutes(from, to));
}
```

Unfortunately, it is three times longer than the other version. In programs where you need to minimise several things it might be a good idea to write the generic algorithm like this, so you can re-use it. In most cases the first version is probably good enough.

> Note the `getProperty` function though, it is often useful when doing functional programming with objects.

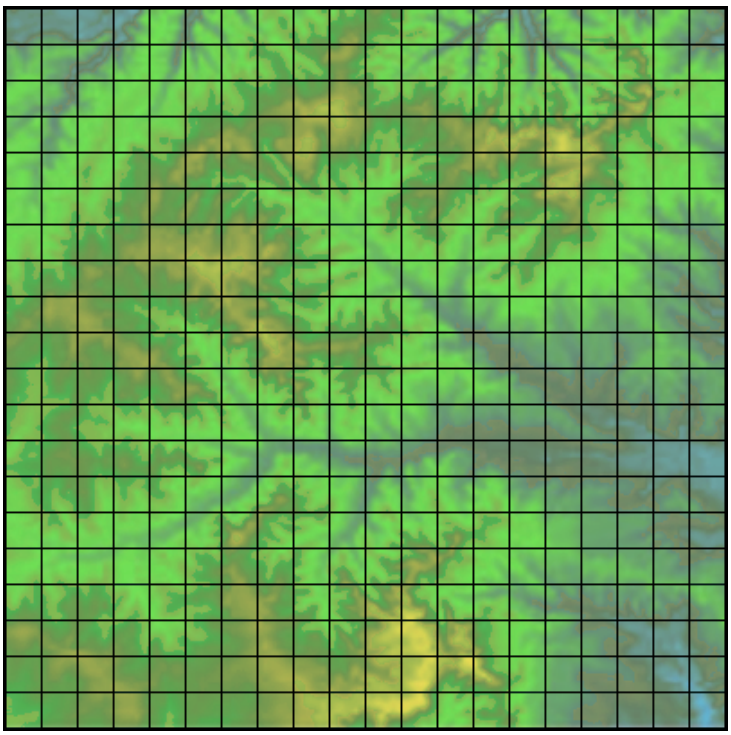
Let us see what route our algorithm comes up with between Point Kiukiu and Point Teohotepapapa...

```
show(shortestRoute("Point Kiukiu", "Point Teohotepapapa").places);
```

On a small island like Hiva Oa, it is not too much work to generate all possible routes. If you try to do that on a reasonably detailed map of, say, Belgium, it is going to take an absurdly long time, not to mention an absurd amount of memory. Still, you have probably seen those online route-planners. These give you a more or less optimal route through a gigantic maze of roads in just a few seconds. How do they do it?

If you are paying attention, you may have noticed that it is not necessary to generate all routes all the way to the end. If we start comparing routes *while* we are building them, we can avoid building this big set of routes, and, as soon as we have found a single route to our destination, we can stop extending routes that are already longer than that route.

To try this out, we will use a 20 by 20 grid as our map:



What you see here is an elevation map of a mountain landscape. The yellow spots are the peaks, and the blue spots the valleys. The area is divided into squares with a size of a hundred meters. We have at our disposal a function `heightAt`, which can give us the height, in meters, of any square on that map, where squares are represented by objects with `x` and `y` properties.

```
print(heightAt({x: 0, y: 0}));
print(heightAt({x: 11, y: 18}));
```

We want to cross this landscape, on foot, from the top left to the bottom right. A grid can be approached like a graph. Every square is a node, which is connected to the squares around it.

We do not like wasting energy, so we would prefer to take the easiest route possible. Going uphill is heavier than going downhill, and going downhill is heavier than going level[2]. This function calculates the amount of 'weighted meters', between two adjacent squares, which represents how tired you get from walking (or climbing) between them. Going uphill is counted as twice as heavy as going downhill.

```
function weightedDistance(pointA, pointB) {
  var heightDifference = heightAt(pointB) - heightAt(pointA);
  var climbFactor = (heightDifference < 0 ? 1 : 2);
  var flatDistance = (pointA.x == pointB.x || pointA.y == pointB.y ? 100 : 141);
  return flatDistance + climbFactor * Math.abs(heightDifference);
}
```

Note the `flatDistance` calculation. If the two points are on the same row or column, they are right next to each other, and the distance between them is a hundred meters. Otherwise, they are assumed to diagonal from each other, and the diagonal distance between two squares of this size is a hundred times the square root of two, which is approximately `141`. One is not allowed to call this function for squares that are further than one step apart. (It could double-check this... but it is too lazy.)

Points on the map are represented by objects containing `x` and `y` properties. These three functions are useful when working with such objects:

```
function point(x, y) {
  return {x: x, y: y};
}

function addPoints(a, b) {
  return point(a.x + b.x, a.y + b.y);
}

function samePoint(a, b) {
  return a.x == b.x && a.y == b.y;
}

show(samePoint(addPoints(point(10, 10), point(4, -2)),
               point(14, 8)));
```

Ex. 7.4 If we are going to find routes through this map, we will again need a function to create 'signposts', lists of directions that can be taken from a given point. Write a function `possibleDirections`, which takes a point object as argument and returns an array of nearby points. We can only move to adjacent points, both straight and diagonally, so squares have a maximum of eight neighbours. Take care not to return squares that lie outside of the map. For all we know the edge of the map might be the edge of the world.

```
function possibleDirections(from) {
  var mapSize = 20;
  function insideMap(point) {
    return point.x >= 0 && point.x < mapSize &&
           point.y >= 0 && point.y < mapSize;
  }

  var directions = [point(-1, 0), point(1, 0), point(0, -1),
                    point(0, 1), point(-1, -1), point(-1, 1),
                    point(1, 1), point(1, -1)];
  return filter(insideMap, map(partial(addPoints, from),
                               directions));
}

show(possibleDirections(point(0, 0)));
```

I created a variable `mapSize`, for the sole purpose of not having to write `20` two times. If, at some other time, we want to use this same function for another map, it would be clumsy if the code was full of `20`s, which all have to be changed. We could even go as far as making the `mapSize` an argument to `possibleDirections`, so we can use the function for different maps without changing it. I judged that that was not necessary in this case though, such things can always be changed when the need arises.

Then why didn't I also add a variable to hold the `0`, which also occurs two times? I assumed that maps always start at `0`, so this one is unlikely to ever change, and using a variable for it only adds noise.

To find a route on this map without having our browser cut off the program because it takes too long to finish, we have to stop our amateurish attempts and implement a serious algorithm. A lot of work has gone into problems like this in the past, and many solutions have been designed (some brilliant, others useless). A very popular and efficient one is called A* (pronounced A-star). We will spend the rest of the chapter implementing an A* route-finding function for our map.

Before I get to the algorithm itself, let me tell you a bit more about the problem it solves. The trouble with searching routes through graphs is that, in big graphs, there are an awful lot of them. Our Hiva Oa path-finder showed that, when the graph is small, all we needed to do was to make sure our paths didn't revisit points they had already passed. On our new map, this is not enough anymore.

The fundamental problem is that there is too much room for going in the wrong direction. Unless we somehow manage to steer our exploration of paths towards the goal, a choice we make for continuing a given path is more likely to go in the wrong direction than in the right direction. If you keep generating paths like that, you end up with an enormous amount of paths, and even if one of them accidentally reaches the end point, you do not know whether that is the shortest path.

So what you want to do is explore directions that are likely to get you to the end point first. On a grid like our map, you can get a rough estimate of how good a path is by checking how long it is and how close its end is to the end point. By adding path length and an estimate of the distance it still has to go, you can get a rough idea of which paths are promising. If you extend promising paths first, you are less likely to waste time on useless ones.

But that still is not enough. If our map was of a perfectly flat plane, the path that looked

promising would almost always be the best one, and we could use the above method to walk right to our goal. But we have valleys and hillsides blocking our paths, so it is hard to tell in advance which direction will be the most efficient path. Because of this, we still end up having to explore way too many paths.

To correct this, we can make clever use of the fact that we are constantly exploring the most promising path first. Once we have determined that path A is the best way to get to point X, we can remember that. When, later on, path B also gets to point X, we know that it is not the best route, so we do not have to explore it further. This can prevent our program from building a lot of pointless paths.

The algorithm, then, goes something like this...

There are two pieces of data to keep track of. The first one is called the open list, it contains the partial routes that must still be explored. Each route has a score, which is calculated by adding its length to its estimated distance from the goal. This estimate must always be optimistic, it should never overestimate the distance. The second is a set of nodes that we have seen, together with the shortest partial route that got us there. This one we will call the reached list. We start by adding a route that contains only the starting node to the open list, and recording it in the reached list.

Then, as long as there are any nodes in the open list, we take out the one that has the lowest (best) score, and find the ways in which it can be continued (by calling `possibleDirections`). For each of the nodes this returns, we create a new route by appending it to our original route, and adjusting the length of the route using `weightedDistance`. The endpoint of each of these new routes is then looked up in the reached list.

If the node is not in the reached list yet, it means we have not seen it before, and we add the new route to the open list and record it in the reached list. If we *had* seen it before, we compare the score of the new route to the score of the route in the reached list. If the new route is shorter, we replace the existing route with the new one. Otherwise, we discard the new route, since we have already seen a better way to get to that point.

We continue doing this until the route we fetch from the open list ends at the goal node, in which case we have found our route, or until the open list is empty, in which case we have found out that there is no route. In our case the map contains no unsurmountable obstacles, so there is always a route.

How do we know that the first full route that we get from the open list is also the shortest one? This is a result of the fact that we only look at a route when it has the lowest score. The score of a route is its actual length plus an *optimistic* estimate of the remaining length. This means that if a route has the lowest score in the open list, it is always the best route to its current endpoint — it is impossible for another route to later find a better way to that point, because if it were better, its score would have been lower.

Try not to get frustrated when the fine points of why this works are still eluding you. When thinking about algorithms like this, having seen 'something like it' before helps a lot, it gives you a point of reference to compare the approach to. Beginning programmers have to do without such a point of reference, which makes it rather easy to get lost. Just realise that this is advanced stuff, globally read over the rest of the chapter, and come back to it later when you feel like a challenge.

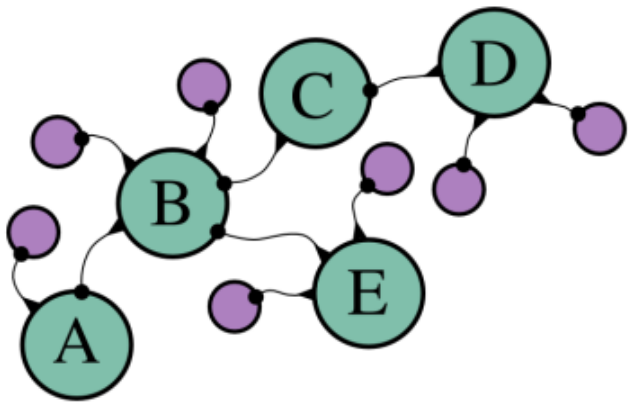
I am afraid that, for one aspect of the algorithm, I'm going to have to invoke magic again. The open list needs to be able to hold a large amount of routes, and to quickly find the route with the lowest score among them. Storing them in a normal array, and searching through this array every time, is way too slow, so I give you a data structure called a binary heap. You create them with `new`, just like `Date` objects, giving them a function that is used to 'score' its elements as argument. The resulting object has the methods `push` and `pop`, just like an array, but `pop` always gives you the element with the lowest score, instead of the one that was `push`ed last.

```
function identity(x) {
  return x;
}

var heap = new BinaryHeap(identity);
forEach([2, 4, 5, 1, 6, 3], function(number) {
  heap.push(number);
});
while (heap.size() > 0)
  show(heap.pop());
```

Appendix 2 discusses the implementation of this data structure, which is quite interesting. After you have read chapter 8, you might want to take a look at it.

The need to squeeze out as much efficiency as we can has another effect. The Hiva Oa algorithm used arrays of locations to store routes, and copied them with the `concat` method when it extended them. This time, we can not afford to copy arrays, since we will be exploring lots and lots of routes. Instead, we use a 'chain' of objects to store a route. Every object in the chain has some properties, such as a point on the map, and the length of the route so far, and it also has a property that points at the previous object in the chain. Something like this:



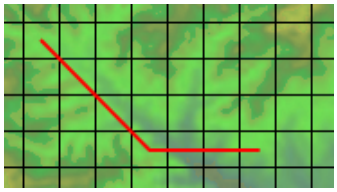
Where the cyan circles are the relevant objects, and the lines represent properties — the end with the dot points at the value of the property. Object `A` is the start of a route here. Object `B` is used to build a new route, which continues from `A`. It has a property, which we will call `from`, pointing at the route it is based on. When we need to reconstruct a route later, we can follow these properties to find all the points that the route passed. Note that object `B` is part of two routes, one that ends in `D` and one that ends in `E`. When there are a lot of routes, this can save us much storage space — every new route only needs one new object for itself, the rest is shared with other routes that started the same way.

Ex. 7.5 Write a function `estimatedDistance` that gives an optimistic estimate of the distance

between two points. It does not have to look at the height data, but can assume a flat map. Remember that we are only travelling straight and diagonally, and that we are counting the diagonal distance between two squares as `141`.

```
function estimatedDistance(pointA, pointB) {
  var dx = Math.abs(pointA.x - pointB.x),
      dy = Math.abs(pointA.y - pointB.y);
  if (dx > dy)
    return (dx - dy) * 100 + dy * 141;
  else
    return (dy - dx) * 100 + dx * 141;
}
```

The strange formulae are used to decompose the path into a straight and a diagonal part. If you have a path like this...



... the path is `8` squares wide and `4` high, so you get `8 - 4 = 4` straight moves, and `4` diagonal ones.

If you wrote a function that just computes the straight 'Pythagorean' distance between the points, that would also work. What we need is an optimistic estimate, and assuming you can go straight to the goal is certainly optimistic. However, the closer the estimate is to the real distance, the less useless paths our program has to try out.

Ex. 7.6 We will use a binary heap for the open list. What would be a good data structure for the reached list? This one will be used to look up routes, given a pair of `x`, `y` coordinates. Preferably in a way that is fast. Write three functions, `makeReachedList`, `storeReached`, and `findReached`. The first one creates your data structure, the second one, given a reached list, a point, and a route, stores a route in it, and the last one, given a reached list and point, retrieves a route or returns `undefined` to indicate that no route was found for that point.

One reasonable idea would be to use an object with objects in it. One of the coordinates in the points, say `x`, is used as a property name for the outer object, and the other, `y`, for the inner object. This does require some bookkeeping to handle the fact that, sometimes, the inner object we are looking for is not there (yet).

```
function makeReachedList() {
  return {};
}

function storeReached(list, point, route) {
  var inner = list[point.x];
  if (inner == undefined) {
    inner = {};
    list[point.x] = inner;
  }
  inner[point.y] = route;
}

function findReached(list, point) {
  var inner = list[point.x];
  if (inner == undefined)
    return undefined;
  else
    return inner[point.y];
}
```

Another possibility is to merge the `x` and `y` of the point into a single property name, and use that to store routes in a single object.

```
function pointID(point) {
  return point.x + "-" + point.y;
}

function makeReachedList() {
  return {};
}

function storeReached(list, point, route) {
  list[pointID(point)] = route;
}

function findReached(list, point) {
  return list[pointID(point)];
}
```

Defining a type of data structure by providing a set of functions to create and manipulate such structures is a useful technique. It makes it possible to 'isolate' the code that makes use of the structure from the details of the structure itself. Note that, no matter which of the above two implementations is used, code that needs a reached list works in exactly the same way. It doesn't care what kind of objects are used, as long as it gets the results it expected.

This will be discussed in much more detail in chapter 8, where we will learn to make object types like `BinaryHeap`, which are created using `new` and have methods to manipulate them.

Here we finally have the actual path-finding function:

```
function findRoute(from, to) {
  var open = new BinaryHeap(routeScore);
  var reached = makeReachedList();

  function routeScore(route) {
    if (route.score == undefined)
      route.score = estimatedDistance(route.point, to) +
                    route.length;
    return route.score;
  }
  function addOpenRoute(route) {
    open.push(route);
    storeReached(reached, route.point, route);
  }
  addOpenRoute({point: from, length: 0});

  while (open.size() > 0) {
    var route = open.pop();
    if (samePoint(route.point, to))
      return route;

    forEach(possibleDirections(route.point), function(direction) {
      var known = findReached(reached, direction);
      var newLength = route.length +
                      weightedDistance(route.point, direction);
      if (!known || known.length > newLength){
        if (known)
          open.remove(known);
        addOpenRoute({point: direction,
                      from: route,
                      length: newLength});
      }
    });
  }
  return null;
}
```

First, it creates the data structures it needs, one open list and one reached list. `routeScore` is the scoring function given to the binary heap. Note how it stores its result in the route object, to prevent having to re-calculate it multiple times.

`addOpenRoute` is a convenience function that adds a new route to both the open list and the reached list. It is immediately used to add the start of the route. Note that route objects always have the properties `point`, which holds the point at the end of the route, and `length`, which holds the current length of the route. Routes which are more than one square long also have a `from` property, which points at their predecessors.

The `while` loop, as was described in the algorithm, keeps taking the lowest-scoring route from the open list and checks whether this gets us to the goal point. If it does not, we must continue by expanding it. This is what the `forEach` takes care of. It looks up this new point in the reached list. If it is not found there, or the node found has a longer length that the new route, a new route object is created and added to the open list and reached list, and the existing route (if any) is removed from the open list.

What if the route in `known` is not on the open list? It has to be, because routes are only removed from the open list when they have been found to be the most optimal route to their endpoint. If we try to remove a value from a binary heap that is not on it, it will throw an exception, so if my reasoning is wrong, we will probably see an exception when running the function.

When code gets complex enough to make you doubt certain things about it, it is a good

idea to add some checks that raise exceptions when something goes wrong. That way, you know that there are no weird things happening 'silently', and when you break something, you immediately see what you broke.

Note that this algorithm does not use recursion, but still manages to explore all those branches. The open list more or less takes over the role that the function call stack played in the recursive solution to the Hiva Oa problem — it keeps track of the paths that still have to be explored. Every recursive algorithm can be rewritten in a non-recursive way by using a data structure to store the 'things that must still be done'.

Well, let us try our path-finder:

```
var route = findRoute(point(0, 0), point(19, 19));
```

If you ran all the code above, and did not introduce any errors, that call, though it might take a few seconds to run, should give us a route object. This object is rather hard to read. That can be helped by using the `showRoute` function which, if your console is big enough, will show a route on a map.

```
showRoute(route);
```

You can also pass multiple routes to `showRoute`, which can be useful when you are, for example, trying to plan a scenic route, which must include the beautiful viewpoint at `11`, `17`.

```
showRoute(findRoute(point(0, 0), point(11, 17)),
          findRoute(point(11, 17), point(19, 19)));
```

Variations on the theme of searching an optimal route through a graph can be applied to many problems, many of which are not at all related to finding a physical path. For example, a program that needs to solve a puzzle of fitting a number of blocks into a limited space could do this by exploring the various 'paths' it gets by trying to put a certain block in a certain place. The paths that ends up with insufficient room for the last blocks are dead ends, and the path that manages to fit in all blocks is the solution.

1. Computers are deterministic machines: They always react in the same way to the input they receive, so they can not produce truly random values. Therefore, we have to make do with series of numbers that look random, but are in fact the result of some complicated deterministic computation.
2. No really, it is.

Chapter 8:

Object-oriented Programming

In the early nineties, a thing called object-oriented programming stirred up the software industry. Most of the ideas behind were not really new at the time, but they had finally gained enough momentum to start rolling, to become fashionable. Books were being written, courses given, programming languages developed. All of a sudden, everybody was extolling the virtues of object-orientation, enthusiastically applying it to every problem, convincing themselves they had finally found the *right way to write programs*.

These things happen a lot. When a process is hard and confusing, people are always on the lookout for a magic solution. When something looking like such a solution presents itself, they are prepared to become devoted followers. For many programmers, even today, object-orientation (or their view of it) is the gospel. When a program is not 'truly object-oriented', whatever that means, it is considered decidedly inferior.

Few fads have managed to stay popular for as long as this one, though. Object-orientation's longevity can largely be explained by the fact that the ideas at its core are very solid and useful. In this chapter, we will discuss these ideas, along with JavaScript's (rather eccentric) take on them. The above paragraphs are by no means meant to discredit these ideas. What I want to do is warn the reader against developing an unhealthy attachment to them.

As the name suggests, object-oriented programming is related to objects. So far, we have used objects as loose aggregations of values, adding and altering their properties whenever we saw fit. In an object-oriented approach, objects are viewed as little worlds of their own, and the outside world may touch them only through a limited and well-defined interface, a number of specific methods and properties. The 'reached list' we used at the end of chapter 7 is an example of this: We used only three functions, `makeReachedList`, `storeReached`, and `findReached` to interact with it. These three functions form an interface for such objects.

The `Date`, `Error`, and `BinaryHeap` objects we have seen also work like this. Instead of providing regular functions for working with the objects, they provide a way to create such objects, using the `new` keyword, and a number of methods and properties that provide the rest of the interface.

One way to give an object methods is to simply attach function values to it.

```
var rabbit = {};
rabbit.speak = function(line) {
  print("The rabbit says '", line, "'");
};

rabbit.speak("Well, now you're asking me.");
```

In most cases, the method will need to know *who* it should act on. For example, if there are different rabbits, the `speak` method must indicate which rabbit is speaking. For this purpose, there is a special variable called `this`, which is always present when a function is called, and which points at the relevant object when the function is called as a method. A function is called as a method when it is looked up as a property, and immediately called,

as in `object.method()`.

```
function speak(line) {
  print("The ", this.adjective, " rabbit says '", line, "'");
}
var whiteRabbit = {adjective: "white", speak: speak};
var fatRabbit = {adjective: "fat", speak: speak};

whiteRabbit.speak("Oh my ears and whiskers, how late it's getting!");
fatRabbit.speak("I could sure use a carrot right now.");
```

I can now clarify the mysterious first argument to the `apply` method, for which we always used `null` in chapter 6. This argument can be used to specify the object that the function must be applied to. For non-method functions, this is irrelevant, hence the `null`.

```
speak.apply(fatRabbit, ["Yum."]);
```

Functions also have a `call` method, which is similar to `apply`, but you can give the arguments for the function separately instead of as an array:

```
speak.call(fatRabbit, "Burp.");
```

The `new` keyword provides a convenient way of creating new objects. When a function is called with the word `new` in front of it, its `this` variable will point at a *new* object, which it will automatically return (unless it explicitly returns something else). Functions used to create new objects like this are called constructors. Here is a constructor for rabbits:

```
function Rabbit(adjective) {
  this.adjective = adjective;
  this.speak = function(line) {
    print("The ", this.adjective, " rabbit says '", line, "'");
  };
}

var killerRabbit = new Rabbit("killer");
killerRabbit.speak("GRAAAAAAAAAH!");
```

It is a convention, among JavaScript programmers, to start the names of constructors with a capital letter. This makes it easy to distinguish them from other functions.

Why is the `new` keyword even necessary? After all, we could have simply written this:

```
function makeRabbit(adjective) {
  return {
    adjective: adjective,
    speak: function(line) {/*etc*/}
  };
}

var blackRabbit = makeRabbit("black");
```

But that is not entirely the same. `new` does a few things behind the scenes. For one thing, our `killerRabbit` has a property called `constructor`, which points at the `Rabbit` function that created it. `blackRabbit` also has such a property, but it points at the `Object` function.

```
show(killerRabbit.constructor);
show(blackRabbit.constructor);
```

Where did the `constructor` property come from? It is part of the prototype of a rabbit. Prototypes are a powerful, if somewhat confusing, part of the way JavaScript objects work. Every object is based on a prototype, which gives it a set of inherent properties. The simple objects we have used so far are based on the most basic prototype, which is associated with the `Object` constructor. In fact, typing `{}` is equivalent to typing `new Object()`.

```
var simpleObject = {};
show(simpleObject.constructor);
show(simpleObject.toString);
```

`toString` is a method that is part of the `Object` prototype. This means that all simple objects have a `toString` method, which converts them to a string. Our rabbit objects are based on the prototype associated with the `Rabbit` constructor. You can use a constructor's `prototype` property to get access to, well, their prototype:

```
show(Rabbit.prototype);
show(Rabbit.prototype.constructor);
```

Every function automatically gets a `prototype` property, whose `constructor` property points back at the function. Because the rabbit prototype is itself an object, it is based on the `Object` prototype, and shares its `toString` method.

```
show(killerRabbit.toString == simpleObject.toString);
```

Even though objects seem to share the properties of their prototype, this sharing is one-way. The properties of the prototype influence the object based on it, but the properties of this object never change the prototype.

The precise rules are this: When looking up the value of a property, JavaScript first looks at the properties that the object *itself* has. If there is a property that has the name we are looking for, that is the value we get. If there is no such property, it continues searching the prototype of the object, and then the prototype of the prototype, and so on. If no property is found, the value `undefined` is given. On the other hand, when *setting* the value of a property, JavaScript never goes to the prototype, but always sets the property in the object itself.

```
Rabbit.prototype.teeth = "small";
show(killerRabbit.teeth);
killerRabbit.teeth = "long, sharp, and bloody";
show(killerRabbit.teeth);
show(Rabbit.prototype.teeth);
```

This does mean that the prototype can be be used at any time to add new properties and methods to all objects based on it. For example, it might become necessary for our rabbits to dance.

```
Rabbit.prototype.dance = function() {
  print("The ", this.adjective, " rabbit dances a jig.");
};

killerRabbit.dance();
```

And, as you might have guessed, the prototypical rabbit is the perfect place for values that all rabbits have in common, such as the `speak` method. Here is a new approach to the `Rabbit` constructor:

```
function Rabbit(adjective) {
  this.adjective = adjective;
}
Rabbit.prototype.speak = function(line) {
  print("The ", this.adjective, " rabbit says '", line, "'");
};

var hazelRabbit = new Rabbit("hazel");
hazelRabbit.speak("Good Frith!");
```

The fact that all objects have a prototype and receive some properties from this prototype can be tricky. It means that using an object to store a set of things, such as the cats from chapter 4, can go wrong. If, for example, we wondered whether there is a cat called `"constructor"`, we would have checked it like this:

```
var noCatsAtAll = {};
if ("constructor" in noCatsAtAll)
  print("Yes, there definitely is a cat called 'constructor'.");
```

This is problematic. A related problem is that it can often be practical to extend the prototypes of standard constructors such as `Object` and `Array` with new useful functions. For example, we could give all objects a method called `properties`, which returns an array with the names of the (non-hidden) properties that the object has:

```
Object.prototype.properties = function() {
  var result = [];
  for (var property in this)
    result.push(property);
  return result;
};

var test = {x: 10, y: 3};
show(test.properties());
```

And that immediately shows the problem. Now that the `Object` prototype has a property called `properties`, looping over the properties of any object, using `for` and `in`, will also give us that shared property, which is generally not what we want. We are interested only in the properties that the object itself has.

Fortunately, there is a way to find out whether a property belongs to the object itself or to one of its prototypes. Unfortunately, it does make looping over the properties of an object a bit clumsier. Every object has a method called `hasOwnProperty`, which tells us whether the object has a property with a given name. Using this, we could rewrite our `properties` method like this:

```
Object.prototype.properties = function() {
  var result = [];
  for (var property in this) {
    if (this.hasOwnProperty(property))
      result.push(property);
  }
  return result;
};

var test = {"Fat Igor": true, "Fireball": true};
show(test.properties());
```

And of course, we can abstract that into a higher-order function. Note that the `action` function is called with both the name of the property and the value it has in the object.

```
function forEachIn(object, action) {
  for (var property in object) {
    if (object.hasOwnProperty(property))
      action(property, object[property]);
  }
}

var chimera = {head: "lion", body: "goat", tail: "snake"};
forEachIn(chimera, function(name, value) {
  print("The ", name, " of a ", value, ".");
});
```

But, what if we find a cat named `hasOwnProperty`? (You never know.) It will be stored in the object, and the next time we want to go over the collection of cats, calling `object.hasOwnProperty` will fail, because that property no longer points at a function value. This can be solved by doing something even uglier:

```
function forEachIn(object, action) {
  for (var property in object) {
    if (Object.prototype.hasOwnProperty.call(object, property))
      action(property, object[property]);
  }
}

var test = {name: "Mordecai", hasOwnProperty: "Uh-oh"};
forEachIn(test, function(name, value) {
  print("Property ", name, " = ", value);
});
```

Here, instead of using the method found in the object itself, we get the method from the `Object` prototype, and then use `call` to apply it to the right object. Unless someone actually messes with the method in `Object.prototype` (don't do that), this should work correctly.

`hasOwnProperty` can also be used in those situations where we have been using the `in` operator to see whether an object has a specific property. There is one more catch, however. We saw in chapter 4 that some properties, such as `toString`, are 'hidden', and do not show up when going over properties with `for`/`in`. It turns out that browsers in the Gecko family (Firefox, most importantly) give every object a hidden property named `__proto__`, which points to the prototype of that object. `hasOwnProperty` will return `true` for this one, even though the program did not explicitly add it. Having access to the prototype of an object can be very convenient, but making it a property like that was not a very good idea. Still, Firefox is a widely used browser, so when you write a program for the web

you have to be careful with this. There is a method `propertyIsEnumerable`, which returns `false` for hidden properties, and which can be used to filter out strange things like `__proto__`. An expression such as this one can be used to reliably work around this:

```
var object = {foo: "bar"};
show(Object.prototype.hasOwnProperty.call(object, "foo") &&
     Object.prototype.propertyIsEnumerable.call(object, "foo"));
```

Nice and simple, no? This is one of the not-so-well-designed aspects of JavaScript. Objects play both the role of 'values with methods', for which prototypes work great, and 'sets of properties', for which prototypes only get in the way.

Writing the above expression every time you need to check whether a property is present in an object is unworkable. We could put it into a function, but an even better approach is to write a constructor and a prototype specifically for situations like this, where we want to approach an object as just a set of properties. Because you can use it to look things up by name, we will call it a `Dictionary`.

```
function Dictionary(startValues) {
  this.values = startValues || {};
}
Dictionary.prototype.store = function(name, value) {
  this.values[name] = value;
};
Dictionary.prototype.lookup = function(name) {
  return this.values[name];
};
Dictionary.prototype.contains = function(name) {
  return Object.prototype.hasOwnProperty.call(this.values, name) &&
    Object.prototype.propertyIsEnumerable.call(this.values, name);
};
Dictionary.prototype.each = function(action) {
  forEachIn(this.values, action);
};

var colours = new Dictionary({Grover: "blue",
                              Elmo: "orange",
                              Bert: "yellow"});
show(colours.contains("Grover"));
show(colours.contains("constructor"));
colours.each(function(name, colour) {
  print(name, " is ", colour);
});
```

Now the whole mess related to approaching objects as plain sets of properties has been 'encapsulated' in a convenient interface: one constructor and four methods. Note that the `values` property of a `Dictionary` object is not part of this interface, it is an internal detail, and when you are using `Dictionary` objects you do not need to directly use it.

Whenever you write an interface, it is a good idea to add a comment with a quick sketch of what it does and how it should be used. This way, when someone, possibly yourself three months after you wrote it, wants to work with the interface, they can quickly see how to use it, and do not have to study the whole program.

Most of the time, when you are designing an interface, you will soon find some limitations and problems in whatever you came up with, and change it. To prevent wasting your time, it is advisable to document your interfaces only *after* they have been used in a few real situations and proven themselves to be practical. — Of course, this might make it

tempting to forget about documentation altogether. Personally, I treat writing documentation as a 'finishing touch' to add to a system. When it feels ready, it is time to write something about it, and to see if it sounds as good in English (or whatever language) as it does in JavaScript (or whatever programming language).

The distinction between the external interface of an object and its internal details is important for two reasons. Firstly, having a small, clearly described interface makes an object easier to use. You only have to keep the interface in mind, and do not have to worry about the rest unless you are changing the object itself.

Secondly, it often turns out to be necessary or practical to change something about the internal implementation of an object type[1], to make it more efficient, for example, or to fix some problem. When outside code is accessing every single property and detail in the object, you can not change any of them without also updating a lot of other code. If outside code only uses a small interface, you can do what you want, as long as you do not change the interface.

Some people go very far in this. They will, for example, never include properties in the interface of object, only methods — if their object type has a length, it will be accessible with the `getLength` method, not the `length` property. This way, if they ever want to change their object in such a way that it no longer has a `length` property, for example because it now has some internal array whose length it must return, they can update the function without changing the interface.

My own take is that in most cases this is not worth it. Adding a `getLength` method which only contains `return this.length;` mostly just adds meaningless code, and, in most situations, I consider meaningless code a bigger problem than the risk of having to occasionally change the interface to my objects.

Adding new methods to existing prototypes can be very convenient. Especially the `Array` and `String` prototypes in JavaScript could use a few more basic methods. We could, for example, replace `forEach` and `map` with methods on arrays, and make the `startsWith` function we wrote in chapter 4 a method on strings.

However, if your program has to run on the same web-page as another program (either written by you or by someone else) which uses `for`/`in` naively — the way we have been using it so far — then adding things to prototypes, especially the `Object` and `Array` prototype, will definitely break something, because these loops will suddenly start seeing those new properties. For this reason, some people prefer not to touch these prototypes at all. Of course, if you are careful, and you do not expect your code to have to coexist with badly-written code, adding methods to standard prototypes is a perfectly good technique.

In this chapter we are going to build a virtual terrarium, a tank with insects moving around in it. There will be some objects involved (this is, after all, the chapter on object-oriented programming). We will take a rather simple approach, and make the terrarium a two-dimensional grid, like the second map in chapter 7. On this grid there are a number of bugs. When the terrarium is active, all the bugs get a chance to take an action, such as moving, every half second.

Thus, we chop both time and space into units with a fixed size — squares for space, half seconds for time. This usually makes things easier to model in a program, but of course

has the drawback of being wildly inaccurate. Fortunately, this terrarium-simulator is not required to be accurate in any way, so we can get away with it.

A terrarium can be defined with a 'plan', which is an array of strings. We could have used a single string, but because JavaScript strings must stay on a single line it would have been a lot harder to type.

```
var thePlan =
  ["############################",
   "#       #    #      o      ##",
   "#                           #",
   "#           #####           #",
   "##          #   #    ##     #",
   "###           ##     #      #",
   "#           ###      #      #",
   "#   ####                    #",
   "#   ##       o              #",
   "# o  #         o       ### #",
   "#    #                     #",
   "############################"];
```

The `"#"` characters are used to represent the walls of the terrarium (and the ornamental rocks lying in it), the `"o"`s represent bugs, and the spaces are, as you might have guessed, empty space.

Such a plan-array can be used to create a terrarium-object. This object keeps track of the shape and content of the terrarium, and lets the bugs inside move. It has four methods: Firstly `toString`, which converts the terrarium back to a string similar to the plan it was based on, so that you can see what is going on inside it. Then there is `step`, which allows all the bugs in the terrarium to move one step, if they so desire. And finally, there are `start` and `stop`, which control whether the terrarium is 'running'. When it is running, `step` is automatically called every half second, so the bugs keep moving.

Ex. 8.1

The points on the grid will be represented by objects again. In chapter 7 we used three functions, `point`, `addPoints`, and `samePoint` to work with points. This time, we will use a constructor and two methods. Write the constructor `Point`, which takes two arguments, the x and y coordinates of the point, and produces an object with `x` and `y` properties. Give the prototype of this constructor a method `add`, which takes another point as argument and returns a *new* point whose `x` and `y` are the sum of the `x` and `y` of the two given points. Also add a method `isEqualTo`, which takes a point and returns a boolean indicating whether the `this` point refers to the same coordinates as the given point.

Apart from the two methods, the `x` and `y` properties are also part of the interface of this type of objects: Code which uses point objects may freely retrieve and modify `x` and `y`.

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
Point.prototype.add = function(other) {
  return new Point(this.x + other.x, this.y + other.y);
};
Point.prototype.isEqualTo = function(other) {
  return this.x == other.x && this.y == other.y;
};

show((new Point(3, 1)).add(new Point(2, 4)));
```

Make sure your version of `add` leaves the `this` point intact and produces a new point object. A method which changes the current point instead would be similar to the `+=` operator, whereas this one is like the `+` operator.

When writing objects to implement a certain program, it is not always very clear which functionality goes where. Some things are best written as methods of your objects, other things are better expressed as separate functions, and some things are best implemented by adding a new type of object. To keep things clear and organised, it is important to keep the amount of methods and responsibilities that an object type has as small as possible. When an object does too much, it becomes a big mess of functionality, and a formidable source of confusion.

I said above that the terrarium object will be responsible for storing its contents and for letting the bugs inside it move. Firstly, note that it *lets* them move, it doesn't *make* them move. The bugs themselves will also be objects, and these objects are responsible for deciding what they want to do. The terrarium merely provides the infrastructure that asks them what to do every half second, and if they decide to move, it makes sure this happens.

Storing the grid on which the content of the terrarium is kept can get quite complex. It has to define some kind of representation, ways to access this representation, a way to initialise the grid from a 'plan' array, a way to write the content of the grid to a string for the `toString` method, and the movement of the bugs on the grid. It would be nice if part of this could be moved into another object, so that the terrarium object itself doesn't get too big and complex.

Whenever you find yourself about to mix data representation and problem-specific code in one object, it is a good idea to try and put the data representation code into a separate type of object. In this case, we need to represent a grid of values, so I wrote a `Grid` type, which supports the operations that the terrarium will need.

To store the values on the grid, there are two options. One can use an array of arrays, like this:

```
var grid = [["0,0", "1,0", "2,0"],
            ["0,1", "1,1", "2,1"]];
show(grid[1][2]);
```

Or the values can all be put into a single array. In this case, the element at `x`,`y` can be found by getting the element at position `x + y * width` in the array, where `width` is the width of the grid.

```
var grid = ["0,0", "1,0", "2,0",
            "0,1", "1,1", "2,1"];
show(grid[2 + 1 * 3]);
```

I chose the second representation, because it makes it much easier to initialise the array. `new Array(x)` produces a new array of length `x`, filled with `undefined` values.

```
function Grid(width, height) {
  this.width = width;
  this.height = height;
  this.cells = new Array(width * height);
}
Grid.prototype.valueAt = function(point) {
  return this.cells[point.y * this.width + point.x];
};
Grid.prototype.setValueAt = function(point, value) {
  this.cells[point.y * this.width + point.x] = value;
};
Grid.prototype.isInside = function(point) {
  return point.x >= 0 && point.y >= 0 &&
         point.x < this.width && point.y < this.height;
};
Grid.prototype.moveValue = function(from, to) {
  this.setValueAt(to, this.valueAt(from));
  this.setValueAt(from, undefined);
};
```

Ex. 8.2

We will also need to go over all the elements of the grid, to find the bugs we need to move, or to convert the whole thing to a string. To make this easy, we can use a higher-order function that takes an action as its argument. Add the method `each` to the prototype of `Grid`, which takes a function of two arguments as its argument. It calls this function for every point on the grid, giving it the point object for that point as its first argument, and the value that is on the grid at that point as second argument.

Go over the points starting at `0,0`, one row at a time, so that `1,0` is handled before `0,1`. This will make it easier to write the `toString` function of the terrarium later. (Hint: Put a `for` loop for the `x` coordinate inside a loop for the `y` coordinate.)

It is advisable not to muck about in the `cells` property of the grid object directly, but use `valueAt` to get at the values. This way, if we decide (for some reason) to use a different method for storing the values, we only have to rewrite `valueAt` and `setValueAt`, and the other methods can stay untouched.

```
Grid.prototype.each = function(action) {
  for (var y = 0; y < this.height; y++) {
    for (var x = 0; x < this.width; x++) {
      var point = new Point(x, y);
      action(point, this.valueAt(point));
    }
  }
};
```

Finally, to test the grid:

```
var testGrid = new Grid(3, 2);
testGrid.setValueAt(new Point(1, 0), "#");
testGrid.setValueAt(new Point(1, 1), "o");
testGrid.each(function(point, value) {
  print(point.x, ",", point.y, ": ", value);
});
```

Before we can start to write a `Terrarium` constructor, we will have to get a bit more specific about these 'bug objects' that will be living inside it. Earlier, I mentioned that the terrarium will ask the bugs what action they want to take. This will work as follows: Each bug object has an `act` method which, when called, returns an 'action'. An action is an object with a `type` property, which names the type of action the bug wants to take, for example `"move"`. For most actions, the action also contains extra information, such as the direction the bug wants to go.

Bugs are terribly myopic, they can only see the squares directly around them on the grid. But these they can use to base their action on. When the `act` method is called, it is given an object with information about the surroundings of the bug in question. For each of the eight directions, it contains a property. The property indicating what is above of the bug is called `"n"`, for North, the one indicating what is above and to the left `"ne"`, for North-East, and so on. To look up the direction these names refer to, the following dictionary object is useful:

```
var directions = new Dictionary(
  {"n":  new Point( 0, -1),
   "ne": new Point( 1, -1),
   "e":  new Point( 1,  0),
   "se": new Point( 1,  1),
   "s":  new Point( 0,  1),
   "sw": new Point(-1,  1),
   "w":  new Point(-1,  0),
   "nw": new Point(-1, -1)});

show(new Point(4, 4).add(directions.lookup("se")));
```

When a bug decides to move, he indicates in which direction he wants to go by giving the resulting action object a `direction` property that names one of these directions. We can make a simple, stupid bug that always just goes south, 'towards the light', like this:

```
function StupidBug() {};
StupidBug.prototype.act = function(surroundings) {
  return {type: "move", direction: "s"};
};
```

Now we can start on the `Terrarium` object type itself. First, its constructor, which takes a plan (an array of strings) as argument, and initialises its grid.

```
var wall = {};

function Terrarium(plan) {
  var grid = new Grid(plan[0].length, plan.length);
  for (var y = 0; y < plan.length; y++) {
    var line = plan[y];
    for (var x = 0; x < line.length; x++) {
      grid.setValueAt(new Point(x, y),
                      elementFromCharacter(line.charAt(x)));
    }
  }
  this.grid = grid;
}

function elementFromCharacter(character) {
  if (character == " ")
    return undefined;
  else if (character == "#")
    return wall;
  else if (character == "o")
    return new StupidBug();
}
```

`wall` is an object that is used to mark the location of walls on the grid. Like a real wall, it doesn't do much, it just sits there and takes up space.

The most straightforward method of a terrarium object is `toString`, which transforms a terrarium into a string. To make this easier, we mark both the `wall` and the prototype of the `StupidBug` with a property `character`, which holds the character that represents them.

```
wall.character = "#";
StupidBug.prototype.character = "o";

function characterFromElement(element) {
  if (element == undefined)
    return " ";
  else
    return element.character;
}

show(characterFromElement(wall));
```

Ex. 8.3 Now we can use the `each` method of the `Grid` object to build up a string. But to make the result readable, it would be nice to have a newline at the end of every row. The `x` coordinate of the positions on the grid can be used to determine when the end of a line is reached. Add a method `toString`, which takes no arguments and returns a string which, when given to `print`, shows a nice two-dimensional view of the terrarium.

```
Terrarium.prototype.toString = function() {
  var characters = [];
  var endOfLine = this.grid.width - 1;
  this.grid.each(function(point, value) {
    characters.push(characterFromElement(value));
    if (point.x == endOfLine)
      characters.push("\n");
  });
  return characters.join("");
};
```

And to try it out...

```
var terrarium = new Terrarium(thePlan);
print(terrarium.toString());
```

It is possible that, when trying to solve the above exercise, you have tried to access `this.grid` inside the function that you pass as an argument to the grid's `each` method. This will not work. Calling a function always results in a new `this` being defined inside that function, even when it is not used as a method. Thus, any `this` variable outside of the function will not be visible.

Sometimes it is straightforward to work around this by storing the information you need in a variable, like `endOfLine`, which *is* visible in the inner function. If you need access to the whole `this` object, you can store that in a variable too. The name `self` (or `that`) is often used for such a variable.

But all these extra variables can get messy. Another good solution is to use a function similar to `partial` from chapter 6. Instead of adding arguments to a function, this one adds a `this` object, using the first argument to the function's `apply` method:

```
function bind(func, object) {
  return function(){
    return func.apply(object, arguments);
  };
}

var testArray = [];
var pushTest = bind(testArray.push, testArray);
pushTest("A");
pushTest("B");
show(testArray);
```

This way, you can `bind` an inner function to `this`, and it will have the same `this` as the outer function.

Ex. 8.4 In the expression `bind(testArray.push, testArray)` the name `testArray` still occurs twice. Can you design a function `method`, which allows you to bind an object to one of its methods *without* naming the object twice?

Is possible to give the name of the method as a string. This way, the `method` function can look up the correct function value for itself.

```
function method(object, name) {
  return function() {
    object[name].apply(object, arguments);
  };
}

var pushTest = method(testArray, "push");
```

We will need `bind` (or `method`) when implementing the `step` method of a terrarium. This method has to go over all the bugs on the grid, ask them for an action, and execute the given action. You might be tempted to use `each` on the grid, and just handle the bugs we

come across. But then, when a bug moves South or East, we will come across it again in the same turn, and allow it to move again.

Instead, we first gather all the bugs into an array, and then process them. This method gathers bugs, or other things that have an `act` method, and stores them in objects that also contain their current position:

```
Terrarium.prototype.listActingCreatures = function() {
  var found = [];
  this.grid.each(function(point, value) {
    if (value != undefined && value.act)
      found.push({object: value, point: point});
  });
  return found;
};
```

Ex. 8.5

When asking a bug to act, we must pass it an object with information about its current surroundings. This object will use the direction names we saw earlier (`"n"`, `"ne"`, etcetera) as property names. Each property holds a string of one character, as returned by `characterFromElement`, indicating what the bug can see in that direction.

Add a method `listSurroundings` to the `Terrarium` prototype. It takes one argument, the point at which the bug is currently standing, and returns an object with information about the surroundings of that point. When the point is at the edge of the grid, use `"#"` for the directions that go outside of the grid, so the bug will not try to move there.

Hint: Do not write out all the directions, use the `each` method on the `directions` dictionary.

```
Terrarium.prototype.listSurroundings = function(center) {
  var result = {};
  var grid = this.grid;
  directions.each(function(name, direction) {
    var place = center.add(direction);
    if (grid.isInside(place))
      result[name] = characterFromElement(grid.valueAt(place));
    else
      result[name] = "#";
  });
  return result;
};
```

Note the use of the `grid` variable to work around the `this` problem.

Both above methods are not part of the external interface of a `Terrarium` object, they are internal details. Some languages provide ways to explicitly declare certain methods and properties 'private', and make it an error to use them from outside the object. JavaScript does not, so you will have to rely on comments to describe the interface to an object. Sometimes it can be useful to use some kind of naming scheme to distinguish between external and internal properties, for example by prefixing all internal ones with an underscore ('_'). This will make accidental uses of properties that are not part of an object's interface easier to spot.

Next is one more internal method, the one that will ask a bug for an action and carry it out. It takes an object with `object` and `point` properties, as returned by

`listActingCreatures`, as argument. For now, it only knows about the `"move"` action:

```
Terrarium.prototype.processCreature = function(creature) {
  var surroundings = this.listSurroundings(creature.point);
  var action = creature.object.act(surroundings);
  if (action.type == "move" && directions.contains(action.direction)) {
    var to = creature.point.add(directions.lookup(action.direction));
    if (this.grid.isInside(to) && this.grid.valueAt(to) == undefined)
      this.grid.moveValue(creature.point, to);
  }
  else {
    throw new Error("Unsupported action: " + action.type);
  }
};
```

Note that it checks whether the chosen direction is inside of the grid and empty, and ignores it otherwise. This way, the bugs can ask for any action they like — the action will only be carried out if it is actually possible. This acts as a layer of insulation between the bugs and the terrarium, and allows us to be less precise when writing the bugs' `act` methods — for example the `StupidBug` just always travels South, regardless of any walls that might stand in its way.

These three internal methods then finally allow us to write the `step` method, which gives all bugs a chance to do something (all elements with an `act` method — we could also give the `wall` object one if we so desired, and make the walls walk).

```
Terrarium.prototype.step = function() {
  forEach(this.listActingCreatures(),
          bind(this.processCreature, this));
};
```

Now, let us make a terrarium and see whether the bugs move...

```
var terrarium = new Terrarium(thePlan);
print(terrarium);
terrarium.step();
print(terrarium);
```

Wait, how come the above calls `print(terrarium)` and ends up displaying the output of our `toString` method? `print` turns its arguments to strings using the `String` function. Objects are turned to strings by calling their `toString` method, so giving your own object types a meaningful `toString` is a good way to make them readable when printed out.

```
Point.prototype.toString = function() {
  return "(" + this.x + "," + this.y + ")";
};
print(new Point(5, 5));
```

As promised, `Terrarium` objects also get `start` and `stop` methods to start or stop their simulation. For this, we will use two functions provided by the browser, called `setInterval` and `clearInterval`. The first is used to cause its first argument (a function, or a string containing JavaScript code) to be executed periodically. Its second argument gives the amount of milliseconds (1/1000 second) between invocations. It returns a value that can

be given to `clearInterval` to stop its effect.

```
var annoy = setInterval(function() {print("What?");}, 400);
```

And...

```
clearInterval(annoy);
```

There are similar functions for one-shot time-based actions. `setTimeout` causes a function or string to be executed after a given amount of milliseconds, and `clearTimeout` cancels such an action.

```
Terrarium.prototype.start = function() {
  if (!this.running)
    this.running = setInterval(bind(this.step, this), 500);
};

Terrarium.prototype.stop = function() {
  if (this.running) {
    clearInterval(this.running);
    this.running = null;
  }
};
```

Now we have a terrarium with some simple-minded bugs, and we can run it. But to see what is going on, we have to repeatedly do `print(terrarium)`, or we won't see what is going on. That is not very practical. It would be nicer if it would print automatically. It would also look better if, instead of printing a thousand terraria below each other, we could update a single printout of the terrarium. For that second problem, this page conveniently provides a function called `inPlacePrinter`. It returns a function like `print` which, instead of adding to the output, replaces its previous output.

```
var printHere = inPlacePrinter();
printHere("Now you see it.");
setTimeout(partial(printHere, "Now you don't."), 1000);
```

To cause the terrarium to be re-printed every time it changes, we can modify the `step` method as follows:

```
Terrarium.prototype.step = function() {
  forEach(this.listActingCreatures(),
          bind(this.processCreature, this));
  if (this.onStep)
    this.onStep();
};
```

Now, when an `onStep` property has been added to a terrarium, it will be called on every step.

```
var terrarium = new Terrarium(thePlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Note the use of `partial` — it produces an in-place printer applied to the terrarium. Such a

printer only takes one argument, so after partially applying it there are no arguments left, and it becomes a function of zero arguments. That is exactly what we need for the `onStep` property.

Don't forget to stop the terrarium when it is no longer interesting (which should be pretty soon), so that it does not keep wasting your computer's resources:

```
terrarium.stop();
```

But who wants a terrarium with just one kind of bug, and a stupid bug at that? Not me. It would be nice if we could add different kinds of bugs. Fortunately, all we have to is to make the `elementFromCharacter` function more general. Right now it contains three cases which are typed in directly, or 'hard-coded':

```
function elementFromCharacter(character) {
  if (character == " ")
    return undefined;
  else if (character == "#")
    return wall;
  else if (character == "o")
    return new StupidBug();
}
```

The first two cases we can leave intact, but the last one is way too specific. A better approach would be to store the characters and the corresponding bug-constructors in a dictionary, and look for them there:

```
var creatureTypes = new Dictionary();
creatureTypes.register = function(constructor) {
  this.store(constructor.prototype.character, constructor);
};

function elementFromCharacter(character) {
  if (character == " ")
    return undefined;
  else if (character == "#")
    return wall;
  else if (creatureTypes.contains(character))
    return new (creatureTypes.lookup(character))();
  else
    throw new Error("Unknown character: " + character);
}
```

Note how the `register` method is added to `creatureTypes` — this is a dictionary object, but there is no reason why it shouldn't support an additional method. This method looks up the character associated with a constructor, and stores it in the dictionary. It should only be called on constructors whose prototype does actually have a `character` property.

`elementFromCharacter` now looks up the character it is given in `creatureTypes`, and raises an exception when it comes across an unknown character.

Here is a new bug type, and the call to register its character in `creatureTypes`:

```
function BouncingBug() {
  this.direction = "ne";
}
BouncingBug.prototype.act = function(surroundings) {
  if (surroundings[this.direction] != " ")
    this.direction = (this.direction == "ne" ? "sw" : "ne");
  return {type: "move", direction: this.direction};
};
BouncingBug.prototype.character = "%";

creatureTypes.register(BouncingBug);
```

Can you figure out what it does?

Ex. 8.6 Create a bug type called `DrunkBug` which tries to move in a random direction every turn, never mind whether there is a wall there. Remember the `Math.random` trick from chapter 7.

To pick a random direction, we will need an array of direction names. We could of course just type `["n", "ne", ...]`, but that duplicates information, and duplicated information makes me nervous. We could also use the `each` method in `directions` to build the array, which is better already.

But there is clearly a generality to be discovered here. Getting a list of the property names in a dictionary sounds like a useful tool to have, so we add it to the `Dictionary` prototype.

```
Dictionary.prototype.names = function() {
  var names = [];
  this.each(function(name, value) {names.push(name);});
  return names;
};

show(directions.names());
```

A real neurotic programmer would immediately restore symmetry by also adding a `values` method, which returns a list of the values stored in the dictionary. But I guess that can wait until we need it.

Here is a way to take a random element from an array:

```
function randomElement(array) {
  if (array.length == 0)
    throw new Error("The array is empty.");
  return array[Math.floor(Math.random() * array.length)];
}

show(randomElement(["heads", "tails"]));
```

And the bug itself:

```
function DrunkBug() {};
DrunkBug.prototype.act = function(surroundings) {
  return {type: "move",
          direction: randomElement(directions.names())};
};
DrunkBug.prototype.character = "~";

creatureTypes.register(DrunkBug);
```

So, let us test out our new bugs:

```
var newPlan =
  ["############################",
   "#                      #####",
   "#    ##                 ####",
   "#   ####     ~ ~          ##",
   "#    ##       ~            #",
   "#                          #",
   "#                 ###      #",
   "#                #####     #",
   "#                 ###      #",
   "# %         ###        %   #",
   "#         #######          #",
   "############################"];

var terrarium = new Terrarium(newPlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Notice the bouncing bugs bouncing off the drunk ones? Pure drama. Anyway, when you are done watching this fascinating show, shut it down:

```
terrarium.stop();
```

We now have two kinds of objects that both have an `act` method and a `character` property. Because they share these traits, the terrarium can approach them in the same way. This allows us to have all kinds of bugs, without changing anything about the terrarium code. This technique is called polymorphism, and it is arguably the most powerful aspect of object-oriented programming.

The basic idea of polymorphism is that when a piece of code is written to work with objects that have a certain interface, any kind of object that happens to support this interface can be plugged into the code, and it will just work. We already saw simple examples of this, like the `toString` method on objects. All objects that have a meaningful `toString` method can be given to `print` and other functions that need to convert values to strings, and the correct string will be produced, no matter how their `toString` method chooses to build this string.

Similarly, `forEach` works on both real arrays and the pseudo-arrays found in the `arguments` variable, because all it needs is a `length` property and properties called `0`, `1`, and so on, for the elements of the array.

To make life in the terrarium more life-like, we will add to it the concepts of food and reproduction. Each living thing in the terrarium gets a new property, `energy`, which is reduced by performing actions, and increased by eating things. When it has enough energy, a thing can reproduce[2], generating a new creature of the same kind.

If there are only bugs, wasting energy by moving around and eating each other, a terrarium will soon succumb to the forces of entropy, run out of energy, and become a lifeless wasteland. To prevent this from happening (too quickly, at least), we add lichen to the terrarium. Lichen do not move, they just use photo-synthesis to gather energy, and reproduce.

To make this work, we will need a terrarium with a different `processCreature` method. We

could just replace the method of to the `Terrarium` prototype, but we have become very attached to the simulation of the bouncing and drunk bugs, and we would hate to break our old terrarium.

What we can do is create a new constructor, `LifeLikeTerrarium`, whose prototype is based on the `Terrarium` prototype, but which has a different `processCreature` method.

There are a few ways to do this. We could go over the properties of `Terrarium.prototype`, and add them one by one to `LifeLikeTerrarium.prototype`. This is easy to do, and in some cases it is the best solution, but in this case there is a cleaner way. If we make the old prototype object the prototype of the new prototype object (you may have to re-read that a few times), it will automatically have all its properties.

Unfortunately, JavaScript does not have a straightforward way to create an object whose prototype is a certain other object. It is possible to write a function that does this, though, by using the following trick:

```
function clone(object) {
  function OneShotConstructor(){}
  OneShotConstructor.prototype = object;
  return new OneShotConstructor();
}
```

This function uses an empty one-shot constructor, whose prototype is the given object. When using `new` on this constructor, it will create a new object based on the given object.

```
function LifeLikeTerrarium(plan) {
  Terrarium.call(this, plan);
}
LifeLikeTerrarium.prototype = clone(Terrarium.prototype);
LifeLikeTerrarium.prototype.constructor = LifeLikeTerrarium;
```

The new constructor doesn't need to do anything different from the old one, so it just calls the old one on the `this` object. We also have to restore the `constructor` property in the new prototype, or it would claim its constructor is `Terrarium` (which, of course, is only really a problem when we make use of this property, which we don't).

It is now possible to replace some of the methods of the `LifeLikeTerrarium` object, or add new ones. We have based a new object type on an old one, which saved us the work of re-writing all the methods which are the same in `Terrarium` and `LifeLikeTerrarium`. This technique is called 'inheritance'. The new type inherits the properties of the old type. In most cases, this means the new type will still support the interface of the old type, though it might also support a few methods that the old type does not have. This way, objects of the new type can be (polymorphically) used in all the places where objects of the old type could be used.

In most programming languages with explicit support for object-oriented programming, inheritance is a very straightforward thing. In JavaScript, the language doesn't really specify a simple way to do it. Because of this, JavaScript programmers have invented many different approaches to inheritance. Unfortunately, none of them is quite perfect. Fortunately, such a broad range of approaches allows a programmer to choose the most suitable one for the problem he is solving, and allows certain tricks that would be utterly impossible in other languages.

At the end of this chapter, I will show a few other ways to do inheritance, and the issues they have.

Here is the new `processCreature` method. It is big.

```
LifeLikeTerrarium.prototype.processCreature = function(creature) {
  var surroundings = this.listSurroundings(creature.point);
  var action = creature.object.act(surroundings);

  var target = undefined;
  var valueAtTarget = undefined;
  if (action.direction && directions.contains(action.direction)) {
    var direction = directions.lookup(action.direction);
    var maybe = creature.point.add(direction);
    if (this.grid.isInside(maybe)) {
      target = maybe;
      valueAtTarget = this.grid.valueAt(target);
    }
  }

  if (action.type == "move") {
    if (target && !valueAtTarget) {
      this.grid.moveValue(creature.point, target);
      creature.point = target;
      creature.object.energy -= 1;
    }
  }
  else if (action.type == "eat") {
    if (valueAtTarget && valueAtTarget.energy) {
      this.grid.setValueAt(target, undefined);
      creature.object.energy += valueAtTarget.energy;
    }
  }
  else if (action.type == "photosynthese") {
    creature.object.energy += 1;
  }
  else if (action.type == "reproduce") {
    if (target && !valueAtTarget) {
      var species = characterFromElement(creature.object);
      var baby = elementFromCharacter(species);
      creature.object.energy -= baby.energy * 2;
      if (creature.object.energy > 0)
        this.grid.setValueAt(target, baby);
    }
  }
  else if (action.type == "wait") {
    creature.object.energy -= 0.2;
  }
  else {
    throw new Error("Unsupported action: " + action.type);
  }

  if (creature.object.energy <= 0)
    this.grid.setValueAt(creature.point, undefined);
};
```

The function still starts by asking the creature for an action. Then, if the action has a `direction` property, it immediately computes which point on the grid this direction points to and which value is currently sitting there. Three of the five supported actions need to know this, and the code would be even uglier if they all computed it separately. If there is no `direction` property, or an invalid one, it leaves the variables `target` and `valueAtTarget` undefined.

After this, it goes over all the actions. Some actions require additional checking before they are executed, this is done with a separate `if` so that if a creature, for example, tries to walk through a wall, we do not generate an `"Unsupported action"` exception.

Note that, in the `"reproduce"` action, the parent creature loses twice the energy that the newborn creature gets (childbearing is not easy), and the new creature is only placed on the grid if the parent had enough energy to produce it.

After the action has been performed, we check whether the creature is out of energy. If it is, it dies, and we remove it.

Lichen is not a very complex organism. We will use the character `"*"` to represent it. Make sure you have defined the `randomElement` function from exercise 8.6, because it is used again here.

```
function Lichen() {
  this.energy = 5;
}
Lichen.prototype.act = function(surroundings) {
  var emptySpace = findDirections(surroundings, " ");
  if (this.energy >= 13 && emptySpace.length > 0)
    return {type: "reproduce", direction: randomElement(emptySpace)};
  else if (this.energy < 20)
    return {type: "photosynthese"};
  else
    return {type: "wait"};
};
Lichen.prototype.character = "*";

creatureTypes.register(Lichen);

function findDirections(surroundings, wanted) {
  var found = [];
  directions.each(function(name) {
    if (surroundings[name] == wanted)
      found.push(name);
  });
  return found;
}
```

Lichen do not grow bigger than 20 energy, or they would get *huge* when they are surrounded by other lichen and have no room to reproduce.

Ex. 8.7

Create a `LichenEater` creature. It starts with an energy of `10`, and behaves in the following way:

- When it has an energy of 30 or more, and there is room near it, it reproduces.
- Otherwise, if there are lichen nearby, it eats a random one.
- Otherwise, if there is space to move, it moves into a random nearby empty square.
- Otherwise, it waits.

Use `findDirections` and `randomElement` to check the surroundings and to pick directions. Give the lichen-eater `"c"` as its character (pac-man).

```
function LichenEater() {
  this.energy = 10;
}
LichenEater.prototype.act = function(surroundings) {
  var emptySpace = findDirections(surroundings, " ");
  var lichen = findDirections(surroundings, "*");

  if (this.energy >= 30 && emptySpace.length > 0)
    return {type: "reproduce", direction: randomElement(emptySpace)};
  else if (lichen.length > 0)
    return {type: "eat", direction: randomElement(lichen)};
  else if (emptySpace.length > 0)
    return {type: "move", direction: randomElement(emptySpace)};
  else
    return {type: "wait"};
};
LichenEater.prototype.character = "c";

creatureTypes.register(LichenEater);
```

And try it out.

```
var lichenPlan =
  ["############################",
   "#                      ######",
   "#    ***                **##",
   "#   *####*          **  c  *##",
   "#    ***     c     ##**    *#",
   "#       c          ##***   *#",
   "#                  ####    *#",
   "#   c        #*             *#",
   "#*           #**       c    *#",
   "#***         ##**    c     **#",
   "#*****     ##***       *###",
   "############################"];

var terrarium = new LifeLikeTerrarium(lichenPlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Most likely, you will see the lichen quickly over-grow a large part of the terrarium, after which the abundance of food makes the eaters so numerous that they wipe out all the lichen, and thus themselves. Ah, tragedy of nature.

```
terrarium.stop();
```

Having the inhabitants of your terrarium go extinct after a few minutes is kind of depressing. To deal with this, we have to teach our lichen-eaters about long-term sustainable farming. By making them only eat if they see at least two lichen nearby, no matter how hungry they are, they will never exterminate the lichen. This requires some discipline, but the result is a biotope that does not destroy itself. Here is a new `act` method — the only change is that it now only eats when `lichen.length` is at least two.

```
LichenEater.prototype.act = function(surroundings) {
  var emptySpace = findDirections(surroundings, " ");
  var lichen = findDirections(surroundings, "*");

  if (this.energy >= 30 && emptySpace.length > 0)
    return {type: "reproduce", direction: randomElement(emptySpace)};
  else if (lichen.length > 1)
    return {type: "eat", direction: randomElement(lichen)};
  else if (emptySpace.length > 0)
    return {type: "move", direction: randomElement(emptySpace)};
  else
    return {type: "wait"};
};
```

Run the above `lichenPlan` terrarium again, and see how it goes. Unless you are very lucky, the lichen-eaters will probably still go extinct after a while, because, in a time of mass starvation, they crawl aimlessly back and forth through empty space, instead of finding the lichen that is sitting just around the corner.

Ex. 8.8 Find a way to modify the `LichenEater` to be more likely to survive. Do not cheat — `this.energy += 100` is cheating. If you rewrite the constructor, do not forget to re-register it in the `creatureTypes` dictionary, or the terrarium will continue to use the old constructor.

One approach would be to reduce the randomness of its movement. By always picking a random direction, it will often move back and forth without getting anywhere. By remembering the last direction it went, and preferring that direction, the eater will waste less time, and find food faster.

```
function CleverLichenEater() {
  this.energy = 10;
  this.direction = "ne";
}
CleverLichenEater.prototype.act = function(surroundings) {
  var emptySpace = findDirections(surroundings, " ");
  var lichen = findDirections(surroundings, "*");

  if (this.energy >= 30 && emptySpace.length > 0) {
    return {type: "reproduce",
            direction: randomElement(emptySpace)};
  }
  else if (lichen.length > 1) {
    return {type: "eat",
            direction: randomElement(lichen)};
  }
  else if (emptySpace.length > 0) {
    if (surroundings[this.direction] != " ")
      this.direction = randomElement(emptySpace);
    return {type: "move",
            direction: this.direction};
  }
  else {
    return {type: "wait"};
  }
};
CleverLichenEater.prototype.character = "c";

creatureTypes.register(CleverLichenEater);
```

Try it out using the previous terrarium-plan.

Ex. 8.9 A one-link food chain is still a bit rudimentary. Can you write a new creature, `LichenEaterEater` (character `"@"`), which survives by eating lichen-eaters? Try to find a way to make it fit in the ecosystem without dying out too quickly. Modify the `lichenPlan` array to include a few of these, and try them out.

You are on your own here. I failed to find a really good way to prevent these creatures from either going extinct right away or gobbling up all lichen-eaters and then going extinct. The trick of only eating when it spots two pieces of food doesn't work very well for them, because their food moves around so much it is rare to find two in one place. What does seem to help is making the eater-eater really fat (high energy), so that it can survive times when lichen-eaters are scarce, and only reproduces slowly, which prevents it from exterminating its food source too quickly.

The lichen and eaters go through a periodic movement — sometimes lichen are abundant, which causes a lot of eaters to be born, which causes the lichen to become scarce, which causes the eaters to starve, which causes the lichen to become abundant, and so on. You could try to make the lichen-eater-eaters 'hibernate' (use the `"wait"` action for a while), when they fail to find food for a few turns. If you choose the right amount of turns for this hibernation, or have them wake up automatically when they smell lots of food, this could be a good strategy.

That concludes our discussion of terraria. The rest of the chapter is devoted to a more in-depth look at inheritance, and the problems related to inheritance in JavaScript.

First, some theory. Students of object-oriented programming can often be heard having lengthy, subtle discussions about correct and incorrect uses of inheritance. It is important to bear in mind that inheritance, in the end, is just a trick that allows lazy[3] programmers to write less code. Thus, the question of whether inheritance is being used correctly boils down to the question of whether the resulting code works correctly and avoids useless repetitions. Still, the principles used by these students provide a good way to start thinking about inheritance.

Inheritance is the creation of a new type of objects, the 'sub-type', based on an existing type, the 'super-type'. The sub-type starts with all the properties and methods of the super-type, it inherits them, and then modifies a few of these, and optionally adds new ones. Inheritance is best used when the thing modelled by the sub-type can be said to *be* an object of the super-type.

Thus, a `Piano` type could be a sub-type of an `Instrument` type, because a piano *is* an instrument. Because a piano has a whole array of keys, one might be tempted to make `Piano` a sub-type of `Array`, but a piano *is* no array, and implementing it like that is bound to lead to all kinds of silliness. For example, a piano also has pedals. Why would `piano[0]` give me the first key, and not the first pedal? The situation is, of course, that a piano *has* keys, so it would be better to give it a property `keys`, and possibly another property `pedals`, both holding arrays.

It is possible for a sub-type to be the super-type of yet another sub-type. Some problems are best solved by building a complex family tree of types. You have to take care not to get too inheritance-happy, though. Overuse of inheritance is a great way to make a program into a big ugly mess.

The working of the `new` keyword and the `prototype` property of constructors suggest a

certain way of using objects. For simple objects, such as the terrarium-creatures, this way works rather well. Unfortunately, when a program starts to make serious use of inheritance, this approach to objects quickly becomes clumsy. Adding some functions to take care of common operations can make things a little smoother. Many people define, for example, `inherit` and `method` methods on objects.

```
Object.prototype.inherit = function(baseConstructor) {
  this.prototype = clone(baseConstructor.prototype);
  this.prototype.constructor = this;
};
Object.prototype.method = function(name, func) {
  this.prototype[name] = func;
};

function StrangeArray(){}
StrangeArray.inherit(Array);
StrangeArray.method("push", function(value) {
  Array.prototype.push.call(this, value);
  Array.prototype.push.call(this, value);
});

var strange = new StrangeArray();
strange.push(4);
show(strange);
```

If you search the web for the words 'JavaScript' and 'inheritance', you will come across scores of different variations on this, some of them quite a lot more complex and clever than the above.

Note how the `push` method written here uses the `push` method from the prototype of its parent type. This is something that is done often when using inheritance — a method in the sub-type internally uses a method of the super-type, but extends it somehow.

The biggest problem with this basic approach is the duality between constructors and prototypes. Constructors take a very central role, they are the things that give an object type its name, and when you need to get at a prototype, you have to go to the constructor and take its `prototype` property.

Not only does this lead to a *lot* of typing (`"prototype"` is 9 letters), it is also confusing. We had to write an empty, useless constructor for `StrangeArray` in the example above. Quite a few times, I have found myself accidentally adding methods to a constructor instead of its prototype, or trying to call `Array.slice` when I really meant `Array.prototype.slice`. As far as I am concerned, the prototype itself is the most important aspect of an object type, and the constructor is just an extension of that, a special kind of method.

With a few simple helper methods added to `Object.prototype`, it is possible to create an alternative approach to objects and inheritance. In this approach, a type is represented by its prototype, and we will use capitalised variables to store these prototypes. When it needs to do any 'constructing' work, this is done by a method called `construct`. We add a method called `create` to the `Object` prototype, which is used in place of the `new` keyword. It clones the object, and calls its `construct` method, if there is such a method, giving it the arguments that were passed to `create`.

```
Object.prototype.create = function() {
  var object = clone(this);
  if (typeof object.construct == "function")
    object.construct.apply(object, arguments);
  return object;
};
```

Inheritance can be done by cloning a prototype object and adding or replacing some of its properties. We also provide a convenient shorthand for this, an `extend` method, which clones the object it is applied to and adds to this clone the properties in the object that it is given as an argument.

```
Object.prototype.extend = function(properties) {
  var result = clone(this);
  forEachIn(properties, function(name, value) {
    result[name] = value;
  });
  return result;
};
```

In a case where it is not safe to mess with the `Object` prototype, these can of course be implemented as regular (non-method) functions.

An example. If you are old enough, you may at one time have played a 'text adventure' game, where you move through a virtual world by typing commands, and get textual descriptions of the things around you and the actions you perform. Now those were games!

We could write the prototype for an item in such a game like this.

```
var Item = {
  construct: function(name) {
    this.name = name;
  },
  inspect: function() {
    print("it is ", this.name, ".");
  },
  kick: function() {
    print("klunk!");
  },
  take: function() {
    print("you can not lift ", this.name, ".");
  }
};

var lantern = Item.create("the brass lantern");
lantern.kick();
```

Inherit from it like this...

```
var DetailedItem = Item.extend({
  construct: function(name, details) {
    Item.construct.call(this, name);
    this.details = details;
  },
  inspect: function() {
    print("you see ", this.name, ", ", this.details, ".");
  }
});

var giantSloth = DetailedItem.create(
  "the giant sloth",
  "it is quietly hanging from a tree, munching leaves");
giantSloth.inspect();
```

Leaving out the compulsory `prototype` part makes things like calling `Item.construct` from `DetailedItem`'s constructor slightly simpler. Note that it would be a bad idea to just do `this.name = name` in `DetailedItem.construct`. This duplicates a line. Sure, duplicating the line is shorter than calling the `Item.construct` function, but if we end up adding something to this constructor later, we have to add it in two places.

Most of the time, a sub-type's constructor should start by calling the constructor of the super-type. This way, it starts with a valid object of the super-type, which it can then extend. In this new approach to prototypes, types that need no constructor can leave it out. They will automatically inherit the constructor of their super-type.

```
var SmallItem = Item.extend({
  kick: function() {
    print(this.name, " flies across the room.");
  },
  take: function() {
    // (imagine some code that moves the item to your pocket here)
    print("you take ", this.name, ".");
  }
});

var pencil = SmallItem.create("the red pencil");
pencil.take();
```

Even though `SmallItem` does not define its own constructor, creating it with a `name` argument works, because it inherited the constructor from the `Item` prototype.

JavaScript has an operator called `instanceof`, which can be used to determine whether an object is based on a certain prototype. You give it the object on the left hand side, and a constructor on the right hand side, and it returns a boolean, `true` if the constructor's `prototype` property is the direct or indirect prototype of the object, and `false` otherwise.

When you are not using regular constructors, using this operator becomes rather clumsy — it expects a constructor function as its second argument, but we only have prototypes. A trick similar to the `clone` function can be used to get around it: We use a 'fake constructor', and apply `instanceof` to it.

```
Object.prototype.hasPrototype = function(prototype) {
  function DummyConstructor() {}
  DummyConstructor.prototype = prototype;
  return this instanceof DummyConstructor;
};

show(pencil.hasPrototype(Item));
show(pencil.hasPrototype(DetailedItem));
```

Next, we want to make a small item that has a detailed description. It seems like this item would have to inherit both from `DetailedItem` and `SmallItem`. JavaScript does not allow an object to have multiple prototypes, and even if it did, the problem would not be quite that easy to solve. For example, if `SmallItem` would, for some reason, also define an `inspect` method, which `inspect` method should the new prototype use?

Deriving an object type from more than one parent type is called multiple inheritance. Some languages chicken out and forbid it altogether, others define complicated schemes for making it work in a well-defined and practical way. It is possible to implement a decent multiple-inheritance framework in JavaScript. In fact there are, as usual, multiple good approaches to this. But they all are too complex to be discussed here. Instead, I will show a very simple approach which suffices in most cases.

A mix-in is a specific kind of prototype which can be 'mixed into' other prototypes. `SmallItem` can be seen as such a prototype. By copying its `kick` and `take` methods into another prototype, we mix smallness into this prototype.

```
function mixInto(object, mixIn) {
  forEachIn(mixIn, function(name, value) {
    object[name] = value;
  });
};

var SmallDetailedItem = clone(DetailedItem);
mixInto(SmallDetailedItem, SmallItem);

var deadMouse = SmallDetailedItem.create(
  "Fred the mouse",
  "he is dead");
deadMouse.inspect();
deadMouse.kick();
```

Remember that `forEachIn` only goes over the object's *own* properties, so it will copy `kick` and `take`, but not the constructor that `SmallItem` inherited from `Item`.

Mixing prototypes gets more complex when the mix-in has a constructor, or when some of its methods 'clash' with methods in the prototype that it is mixed into. Sometimes, it is workable to do a 'manual mix-in'. Say we have a prototype `Monster`, which has its own constructor, and we want to mix that with `DetailedItem`.

```
var Monster = Item.extend({
  construct: function(name, dangerous) {
    Item.construct.call(this, name);
    this.dangerous = dangerous;
  },
  kick: function() {
    if (this.dangerous)
      print(this.name, " bites your head off.");
    else
      print(this.name, " runs away, weeping.");
  }
});

var DetailedMonster = DetailedItem.extend({
  construct: function(name, description, dangerous) {
    DetailedItem.construct.call(this, name, description);
    Monster.construct.call(this, name, dangerous);
  },
  kick: Monster.kick
});

var giantSloth = DetailedMonster.create(
  "the giant sloth",
  "it is quietly hanging from a tree, munching leaves",
  true);
giantSloth.kick();
```

But note that this causes `Item` constructor to be called twice when creating a `DetailedMonster` — once through the `DetailedItem` constructor, and once through the `Monster` constructor. In this case there is not much harm done, but there are situations where this would cause a problem.

But don't let those complications discourage you from making use of inheritance. Multiple inheritance, though extremely useful in some situations, can be safely ignored most of the time. This is why languages like Java get away with forbidding multiple inheritance. And if, at some point, you find that you really need it, you can search the web, do some research, and figure out an approach that works for your situation.

Now that I think about it, JavaScript would probably be a great environment for building a text adventure. The ability to change the behaviour of objects at will, which is what prototypical inheritance gives us, is very well suited for this. If you have an object `hedgehog`, which has the unique habit of rolling up when it is kicked, you can just change its `kick` method.

Unfortunately, the text adventure went the way of the vinyl record and, while once very popular, is nowadays only played by a small population of enthusiasts.

1. Such types are usually called 'classes' in other programming languages.
2. To keep things reasonably simple, the creatures in our terrarium reproduce asexually, all by themselves.
3. Laziness, for a programmer, is not necessarily a sin. The kind of people who will industriously do the same thing over and over again tend to make great assembly-line workers and lousy programmers.

Chapter 9:

Modularity

This chapter deals with the process of organising programs. In small programs, organisation rarely becomes a problem. As a program grows, however, it can reach a size where its structure and interpretation become hard to keep track of. Easily enough, such a program starts to look like a bowl of spaghetti, an amorphous mass in which everything seems to be connected to everything else.

When structuring a program, we do two things. We separate it into smaller parts, called modules, each of which has a specific role, and we specify the relations between these parts.

In chapter 8, while developing a terrarium, we made use of a number of functions described in chapter 6. The chapter also defined a few new concepts that had nothing in particular to do with terraria, such as `clone` and the `Dictionary` type. All these things were haphazardly added to the environment. One way to split this program into modules would be:

- A module `FunctionalTools`, which contains the functions from chapter 6, and depends on nothing.
- Then `ObjectTools`, which contains things like `clone` and `create`, and depends on `FunctionalTools`.
- `Dictionary`, containing the dictionary type, and depending on `FunctionalTools`.
- And finally the `Terrarium` module, which depends on `ObjectTools` and `Dictionary`.

When a module depends on another module, it uses functions or variables from that module, and will only work when this module is loaded.

It is a good idea to make sure dependencies never form a circle. Not only do circular dependencies create a practical problem (if module `A` and `B` depend on each other, which one should be loaded first?), it also makes the relation between the modules less straightforward, and can result in a modularised version of the spaghetti I mentioned earlier.

Most modern programming languages have some kind of module system built in. Not JavaScript. Once again, we have to invent something ourselves. The most obvious way to start is to put every module in a different file. This makes it clear which code belongs to which module.

Browsers load JavaScript files when they find a `<script>` tag with an `src` attribute in the HTML of the web-page. The extension `.js` is usually used for files containing JavaScript code. On the console, a shortcut for loading files is provided by the `load` function.

```
load("FunctionalTools.js");
```

In some cases, giving load commands in the wrong order will result in errors. If a module tries to create a `Dictionary` object, but the `Dictionary` module has not been loaded yet, it will be unable to find the constructor, and will fail.

One would imagine this to be easy to solve. Just put some calls to `load` at the top of the

file for a module, to load all the modules it depends on. Unfortunately, because of the way browsers work, calling `load` does not immediately cause the given file to be loaded. The file will be loaded *after* the current file has finished executing. Which is too late, usually.

In most cases, the practical solution is to just manage dependencies by hand: Put the `script` tags in your HTML documents in the right order.

There are two ways to (partially) automate dependency management. The first is to keep a separate file with information about the dependencies between modules. This can be loaded first, and used to determine the order in which to load the files. The second way is to not use a `script` tag (`load` internally creates and adds such a tag), but to fetch the content of the file directly (see chapter 14), and then use the `eval` function to execute it. This makes script loading instantaneous, and thus easier to deal with.

`eval`, short for 'evaluate', is an interesting function. You give it a string value, and it will execute the content of the string as JavaScript code.

```
eval("print(\"I am a string inside a string!\");");
```

You can imagine that `eval` can be used to do some interesting things. Code can build new code, and run it. In most cases, however, problems that can be solved with creative uses of `eval` can also be solved with creative uses of anonymous functions, and the latter is less likely to cause strange problems.

When `eval` is called inside a function, all new variables will become local to that function. Thus, when a variation of the `load` would use `eval` internally, loading the `Dictionary` module would create a `Dictionary` constructor inside of the `load` function, which would be lost as soon as the function returned. There are ways to work around this, but they are rather clumsy.

Let us quickly go over the first variant of dependency management. It requires a special file for dependency information, which could look something like this:

```
var dependencies =
  {"ObjectTools.js": ["FunctionalTools.js"],
   "Dictionary.js":  ["ObjectTools.js"],
   "TestModule.js":  ["FunctionalTools.js", "Dictionary.js"]};
```

The `dependencies` object contains a property for each file that depends on other files. The values of the properties are arrays of file names. Note that we could not use a `Dictionary` object here, because we can not be sure that the `Dictionary` module has been loaded yet. Because all the properties in this object will end in `".js"`, they are unlikely to interfere with hidden properties like `__proto__` or `hasOwnProperty`, and a regular object will work fine.

The dependency manager must do two things. Firstly it must make sure that files are loaded in the correct order, by loading a file's dependencies before the file itself. And secondly, it must make sure that no file is loaded twice. Loading the same file twice might cause problems, and is definitely a waste of time.

```
var loadedFiles = {};

function require(file) {
  if (dependencies[file]) {
    var files = dependencies[file];
    for (var i = 0; i < files.length; i++)
      require(files[i]);
  }
  if (!loadedFiles[file]) {
    loadedFiles[file] = true;
    load(file);
  }
}
```

The `require` function can now be used to load a file and all its dependencies. Note how it recursively calls itself to take care of dependencies (and possible dependencies of that dependency).

```
require("TestModule.js");
```

```
test();
```

Building a program as a set of nice, small modules often means the program will use a lot of different files. When programming for the web, having lots of small JavaScript files on a page tends to make the page slower to load. This does not have to be a problem though. You can write and test your program as a number of small files, and put them all into a single big file when 'publishing' the program to the web.

Just like an object type, a module has an interface. In simple collection-of-functions modules such as `FunctionalTools`, the interface usually consists of all the functions that are defined in the module. In other cases, the interface of the module is only a small part of the functions defined inside it. For example, our manuscript-to-HTML system from chapter 6 only needs an interface of a single function, `renderFile`. (The sub-system for building HTML would be a separate module.)

For modules which only define a single type of object, such as `Dictionary`, the object's interface is the same as the module's interface.

In JavaScript, 'top-level' variables all live together in a single place. In browsers, this place is an object that can be found under the name `window`. The name is somewhat odd, `environment` or `top` would have made more sense, but since browsers associate a JavaScript environment with a window (or 'frame'), someone decided that `window` was a logical name.

```
show(window);
show(window.print == print);
show(window.window.window.window.window);
```

As the third line shows, the name `window` is merely a property of this environment object, pointing at itself.

When much code is loaded into an environment, it will use many top-level variable names. Once there is more code than you can really keep track of, it becomes very easy to accidentally use a name that was already used for something else. This will break the code that used the original value. The proliferation of top-level variables is called name-space pollution, and it can be a rather severe problem in JavaScript — the language will not warn you when you redefine an existing variable.

There is no way to get rid of this problem entirely, but it can be greatly reduced by taking care to cause as little pollution as possible. For one thing, modules should not use top-level variables for values that are not part of their external interface.

Not being able to define any internal functions and variables at all in your modules is, of course, not very practical. Fortunately, there is a trick to get around this. We write all the code for the module inside a function, and then finally add the variables that are part of the module's interface to the `window` object. Because they were created in the same parent function, all the functions of the module can see each other, but code outside of the module can not.

```
function buildMonthNameModule() {
  var names = ["January", "February", "March", "April",
               "May", "June", "July", "August", "September",
               "October", "November", "December"];
  function getMonthName(number) {
    return names[number];
  }
  function getMonthNumber(name) {
    for (var number = 0; number < names.length; number++) {
      if (names[number] == name)
        return number;
    }
  }

  window.getMonthName = getMonthName;
  window.getMonthNumber = getMonthNumber;
}
buildMonthNameModule();

show(getMonthName(11));
```

This builds a very simple module for translating between month names and their number (as used by `Date`, where January is `0`). But note that `buildMonthNameModule` is still a top-level variable that is not part of the module's interface. Also, we have to repeat the names of the interface functions three times. Ugh.

The first problem can be solved by making the module function anonymous, and calling it directly. To do this, we have to add a pair of parentheses around the function value, or JavaScript will think it is a normal function definition, which can not be called directly.

The second problem can be solved with a helper function, `provide`, which can be given an object containing the values that must be exported into the `window` object.

```
function provide(values) {
  forEachIn(values, function(name, value) {
    window[name] = value;
  });
}
```

Using this, we can write a module like this:

```
(function() {
  var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"];
  provide({
    getDayName: function(number) {
      return names[number];
    },
    getDayNumber: function(name) {
      for (var number = 0; number < names.length; number++) {
        if (names[number] == name)
          return number;
      }
    }
  });
})();

show(getDayNumber("Wednesday"));
```

I do not recommend writing modules like this right from the start. While you are still working on a piece of code, it is easier to just use the simple approach we have used so far, and put everything at top level. That way, you can inspect the module's internal values in your browser, and test them out. Once a module is more or less finished, it is not difficult to wrap it in a function.

There are cases where a module will export so many variables that it is a bad idea to put them all into the top-level environment. In cases like this, you can do what the standard `Math` object does, and represent the module as a single object whose properties are the functions and values it exports. For example...

```
var HTML = {
  tag: function(name, content, properties) {
    return {name: name, properties: properties, content: content};
  },
  link: function(target, text) {
    return HTML.tag("a", [text], {href: target});
  }
  /* ... many more HTML-producing functions ... */
};
```

When you need the content of such a module so often that it becomes cumbersome to constantly type `HTML`, you can always move it into the top-level environment using `provide`.

```
provide(HTML);
show(link("http://docs.sun.com/source/816-6408-10/object.htm",
          "This is how objects work."));
```

You can even combine the function and object approaches, by putting the internal variables of the module inside a function, and having this function return an object containing its external interface.

When adding methods to standard prototypes, such as those of `Array` and `Object` a similar problem to name-space pollution occurs. If two modules decide to add a `map` method to `Array.prototype`, you might have a problem. If these two versions of `map` have the precise

same effect, things will continue to work, but only by sheer luck.

Designing an interface for a module or an object type is one of the subtler aspects of programming. On the one hand, you do not want to expose too many details. They will only get in the way when using the module. On the other hand, you do not want to be *too* simple and general, because that might make it impossible to use the module in complex or specialised situations.

Sometimes the solution is to provide two interfaces, a detailed 'low-level' one for complicated things, and a simple 'high-level' one for straightforward situations. The second one can usually be built very easily using the tools provided by the first one.

In other cases, you just have to find the right idea around which to base your interface. Compare this to the various approaches to inheritance we saw in chapter 8. By making prototypes the central concept, rather than constructors, we managed to make some things considerably more straightforward.

The best way to learn to the value of good interface design is, unfortunately, to use bad interfaces. Once you get fed up with them, you'll figure out a way to improve them, and learn a lot in the process. Try not to assume that a lousy interface is 'just the way it is'. Fix it, or wrap it in a new interface that is better (we will see an example of this in chapter 12).

There are functions which require a lot of arguments. Sometimes this means they are just badly designed, and can easily be remedied by splitting them into a few more modest functions. But in other cases, there is no way around it. Typically, some of these arguments have a sensible 'default' value. We could, for example, write yet another extended version of `range`.

```
function range(start, end, stepSize, length) {
  if (stepSize == undefined)
    stepSize = 1;
  if (end == undefined)
    end = start + stepSize * (length - 1);

  var result = [];
  for (; start <= end; start += stepSize)
    result.push(start);
  return result;
}

show(range(0, undefined, 4, 5));
```

It can get hard to remember which argument goes where, not to mention the annoyance of having to pass `undefined` as a second argument when a `length` argument is used. We can make passing arguments to this function more comprehensive by wrapping them in an object.

```
function defaultTo(object, values) {
  forEachIn(values, function(name, value) {
    if (!object.hasOwnProperty(name))
      object[name] = value;
  });
}

function range(args) {
  defaultTo(args, {start: 0, stepSize: 1});
  if (args.end == undefined)
    args.end = args.start + args.stepSize * (args.length - 1);

  var result = [];
  for (; args.start <= args.end; args.start += args.stepSize)
    result.push(args.start);
  return result;
}

show(range({stepSize: 4, length: 5}));
```

The `defaultTo` function is useful for adding default values to an object. It copies the properties of its second argument into its first argument, skipping those that already have a value.

A module or group of modules that can be useful in more than one program is usually called a library. For many programming languages, there is a huge set of quality libraries available. This means programmers do not have to start from scratch all the time, which can make them a lot more productive. For JavaScript, unfortunately, the amount of available libraries is not very large.

But recently this seems to be improving. There are a number of good libraries with 'basic' tools, things like `map` and `clone`. Other languages tend to provide such obviously useful things as built-in standard features, but with JavaScript you'll have to either build a collection of them for yourself or use a library. Using a library is recommended: It is less work, and the code in a library has usually been tested more thoroughly than the things you wrote yourself.

Covering these basics, there are (among others) the 'lightweight' libraries prototype, mootools, jQuery, and MochiKit. There are also some larger 'frameworks' available, which do a lot more than just provide a set of basic tools. YUI (by Yahoo), and Dojo seem to be the most popular ones in that genre. All of these can be downloaded and used free of charge. My personal favourite is MochiKit, but this is mostly a matter of taste. When you get serious about JavaScript programming, it is a good idea to quickly glance through the documentation of each of these, to get a general idea about the way they work and the things they provide.

The fact that a basic toolkit is almost indispensable for any non-trivial JavaScript programs, combined with the fact that there are so many different toolkits, causes a bit of a dilemma for library writers. You either have to make your library depend on one of the toolkits, or write the basic tools yourself and include them with the library. The first option makes the library hard to use for people who are using a different toolkit, and the second option adds a lot of non-essential code to the library. This dilemma might be one of the reasons why there are relatively few good, widely used JavaScript libraries. It is possible that, in the future, ECMAScript 4 and changes in browsers will make toolkits less necessary, and thus (partially) solve this problem.

Chapter 10:

Regular Expressions

At various points in the previous chapters, we had to look for patterns in string values. In chapter 4 we extracted date values from strings by writing out the precise positions at which the numbers that were part of the date could be found. Later, in chapter 6, we saw some particularly ugly pieces of code for finding certain types of characters in a string, for example the characters that had to be escaped in HTML output.

Regular expressions are a language for describing patterns in strings. They form a small, separate language, which is embedded inside JavaScript (and in various other programming languages, in one way or another). It is not a very readable language — big regular expressions tend to be completely unreadable. But, it is a useful tool, and can really simplify string-processing programs.

Just like strings get written between quotes, regular expression patterns get written between slashes (`/`). This means that slashes inside the expression have to be escaped.

```
var slash = /\//;
show("AC/DC".search(slash));
```

The `search` method resembles `indexOf`, but it searches for a regular expression instead of a string. Patterns specified by regular expressions can do a few things that strings can not do. For a start, they allow some of their elements to match more than a single character. In chapter 6, when extracting mark-up from a document, we needed to find the first asterisk or opening brace in a string. That could be done like this:

```
var asteriskOrBrace = /[\{\*]/;
var story =
  "We noticed the *giant sloth*, hanging from a giant branch.";
show(story.search(asteriskOrBrace));
```

The `[` and `]` characters have a special meaning inside a regular expression. They can enclose a set of characters, and they mean 'any of these characters'. Most non-alphanumeric characters have some special meaning inside a regular expression, so it is a good idea to always escape them with a backslash[1] when you use them to refer to the actual characters.

There are a few shortcuts for sets of characters that are needed often. The dot (`.`) can be used to mean 'any character that is not a newline', an escaped 'd' (`\d`) means 'any digit', an escaped 'w' (`\w`) matches any alphanumeric character (including underscores, for some reason), and an escaped 's' (`\s`) matches any white-space (tab, newline, space) character.

```
var digitSurroundedBySpace = /\s\d\s/;
show("1a 2 3d".search(digitSurroundedBySpace));
```

The escaped 'd', 'w', and 's' can be replaced by their capital letter to mean their opposite. For example, `\s` matches any character that is *not* white-space. When using `[` and `]`, a pattern can be inverted by starting with a `^` character:

```
var notABC = /[^ABC]/;
show("ABCBACCBBADABC".search(notABC));
```

As you can see, the way regular expressions use characters to express patterns makes them A) very short, and B) very hard to read.

Ex. 10.1 Write a regular expression that matches a date in the format `"XX/XX/XXXX"`, where the `X`s are digits. Test it against the string `"born 15/11/2003 (mother Spot): White Fang"`.

```
var datePattern = /\d\d\/\d\d\/\d\d\d\d/;
show("born 15/11/2003 (mother Spot): White Fang".search(datePattern));
```

Sometimes you need to make sure a pattern starts at the beginning of a string, or ends at its end. For this, the special characters `^` and `$` can be used. The first matches the start of the string, the second the end.

```
show(/a+/.test("blah"));
show(/^a+$/.test("blah"));
```

The first regular expression matches any string that contains an `a` character, the second only those strings that consist entirely of `a` characters.

Note that regular expressions are objects, and have methods. Their `test` method returns a boolean indicating whether the given string matches the expression.

The code `\b` matches a 'word boundary', which can be punctuation, white-space, or the start or end of the string.

```
show(/cat/.test("concatenate"));
show(/\bcat\b/.test("concatenate"));
```

Parts of a pattern can be allowed to be repeated a number of times. Putting an asterisk (`*`) after an element allows it to be repeated any number of times, including zero. A plus (`+`) does the same, but requires the pattern to occur at least one time. A question mark (`?`) makes an element 'optional' — it can occur zero or one times.

```
var parenthesizedText = /\(.*\)/;
show("Its (the sloth's) claws were gigantic!".search(parenthesizedText));
```

When necessary, braces can be used to be more precise about the amount of times an element may occur. A number between braces (`{4}`) gives the exact amount of times it must occur. Two numbers with a comma between them (`{3,10}`) indicate that the pattern must occur at least as often as the first number, and at most as often as the second one. Similarly, `{2,}` means two or more occurrences, while `{,4}` means four or less.

```
var datePattern = /\d{1,2}\/\d\d?\/\d{4}/;
show("born 15/11/2003 (mother Spot): White Fang".search(datePattern));
```

The pieces `/\d{1,2}/` and `/\d\d?/` both express 'one or two digits'.

Ex. 10.2 Write a pattern that matches e-mail addresses. For simplicity, assume that the parts before and after the `@` can contain only alphanumeric characters and the characters `.` and `-` (dot and dash), while the last part of the address, the country code after the last dot, may only contain alphanumeric characters, and must be two or three characters long.

```
var mailAddress = /\b[\w\.-]+@[\w\.-]+\.\w{2,3}\b/;

show(mailAddress.test("kenny@test.net"));
show(mailAddress.test("I mailt kenny@tets.nets, but it didn wrok!"));
show(mailAddress.test("the_giant_sloth@gmail.com"));
```

The `\b`s at the start and end of the pattern make sure that the second string does not match.

Part of a regular expression can be grouped together with parentheses. This allows us to use `*` and such on more than one character. For example:

```
var cartoonCrying = /boo(hoo+)+/i;
show("Then, he exclaimed 'Boohoooohoohooo'".search(cartoonCrying));
```

Where did the `i` at the end of that regular expression come from? After the closing slash, 'options' may be added to a regular expression. An `i`, here, means the expression is case-insensitive, which allows the lower-case B in the pattern to match the upper-case one in the string.

A pipe character (`|`) is used to allow a pattern to make a choice between two elements. For example:

```
var holyCow = /(sacred|holy) (cow|bovine|bull|taurus)/i;
show(holyCow.test("Sacred bovine!"));
```

Often, looking for a pattern is just a first step in extracting something from a string. In previous chapters, this extraction was done by calling a string's `indexOf` and `slice` methods a lot. Now that we are aware of the existence of regular expressions, we can use the `match` method instead. When a string is matched against a regular expression, the result will be `null` if the match failed, or an array of matched strings if it succeeded.

```
show("No".match(/Yes/));
show("... yes".match(/yes/));
show("Giant Ape".match(/giant (\w+)/i));
```

The first element in the returned array is always the part of the string that matched the pattern. As the last example shows, when there are parenthesized parts in the pattern, the parts they match are also added to the array. Often, this makes extracting pieces of string very easy.

```
var parenthesized = prompt("Tell me something", "").match(/\((.*)\)/);
if (parenthesized != null)
  print("You parenthesized '", parenthesized[1], "'");
```

Ex. 10.3 Re-write the function `extractDate` that we wrote in chapter 4. When given a string, this function looks for something that follows the date format we saw earlier. If it can find such

a date, it puts the values into a `Date` object. Otherwise, it throws an exception. Make it accept dates in which the day or month are written with only one digit.

```
function extractDate(string) {
  var found = string.match(/(\d\d?)\/(\d\d?)\/(\d{4})/);
  if (found == null)
    throw new Error("No date found in '" + string + "'.");
  return new Date(Number(found[3]), Number(found[2]) - 1,
                  Number(found[1]));
}

show(extractDate("born 5/2/2007 (mother Noog): Long-ear Johnson"));
```

This version is slightly longer than the previous one, but it has the advantage of actually checking what it is doing, and shouting out when it is given nonsensical input. This was a lot harder without regular expressions — it would have taken a lot of calls to `indexOf` to find out whether the numbers had one or two digits, and whether the dashes were in the right places.

The `replace` method of string values, which we saw in chapter 6, can be given a regular expression as its first argument.

```
print("Borobudur".replace(/[ou]/g, "a"));
```

Notice the `g` character after the regular expression. It stands for 'global', and means that every part of the string that matches the pattern should be replaced. When this `g` is omitted, only the first `"o"` would be replaced.

Sometimes it is necessary to keep parts of the replaced strings. For example, we have a big string containing the names of people, one name per line, in the format "Lastname, Firstname". We want to swap these names, and remove the comma, to get a simple "Firstname Lastname" format.

```
var names = "Picasso, Pablo\nGauguin, Paul\nVan Gogh, Vincent";
print(names.replace(/([\w ]+), ([\w ]+)/g, "$2 $1"));
```

The `$1` and `$2` the replacement string refer to the parenthesized parts in the pattern. `$1` is replaced by the text that matched against the first pair of parentheses, `$2` by the second, and so on, up to `$9`.

If you have more than 9 parentheses parts in your pattern, this will no longer work. But there is one more way to replace pieces of a string, which can also be useful in some other tricky situations. When the second argument given to the `replace` method is a function value instead of a string, this function is called every time a match is found, and the matched text is replaced by whatever the function returns. The arguments given to the function are the matched elements, similar to the values found in the arrays returned by `match`: The first one is the whole match, and after that comes one argument for every parenthesized part of the pattern.

```
function eatOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) {
    unit = unit.slice(0, unit.length - 1);
  }
  else if (amount == 0) {
    unit = unit + "s";
    amount = "no";
  }
  return amount + " " + unit;
}

var stock = "1 lemon, 2 cabbages, and 101 eggs";
stock = stock.replace(/(\d+) (\w+)/g, eatOne);

print(stock);
```

Ex. 10.4 That last trick can be used to make the HTML-escaper from chapter 6 more efficient. You may remember that it looked like this:

```
function escapeHTML(text) {
  var replacements = [["&", "&amp;"], ["\"", "&quot;"],
                      ["<", "&lt;"], [">", "&gt;"]];
  forEach(replacements, function(replace) {
    text = text.replace(replace[0], replace[1]);
  });
  return text;
}
```

Write a new function `escapeHTML`, which does the same thing, but only calls `replace` once.

```
function escapeHTML(text) {
  var replacements = {"<": "&lt;", ">": "&gt;",
                      "&": "&amp;", "\"": "&quot;"};
  return text.replace(/[<>&"]/g, function(character) {
    return replacements[character];
  });
}

print(escapeHTML("The 'pre-formatted' tag is written \"<pre>\"."));
```

The `replacements` object is a quick way to associate each character with its escaped version. Using it like this is safe (i.e. no `Dictionary` object is needed), because the only properties that will be used are those matched by the `/[<>&"]/` expression.

There are cases where the pattern you need to match against is not known while you are writing the code. Say we are writing a (very simple-minded) obscenity filter for a message board. We only want to allow messages that do not contain obscene words. The administrator of the board can specify a list of words that he or she considers unacceptable.

The most efficient way to check a piece of text for a set of words is to use a regular expression. If we have our word list as an array, we can build the regular expression like this:

```
var badWords = ["ape", "monkey", "simian", "gorilla", "evolution"];
var pattern = new RegExp(badWords.join("|"), "i");
function isAcceptable(text) {
  return !pattern.test(text);
}

show(isAcceptable("Mmmm, grapes."));
show(isAcceptable("No more of that monkeybusiness, now."));
```

We could add `\b` patterns around the words, so that the thing about grapes would not be classified as unacceptable. That would also make the second one acceptable, though, which is probably not correct. Obscenity filters are hard to get right (and usually way too annoying to be a good idea).

The first argument to the `RegExp` constructor is a string containing the pattern, the second argument can be used to add case-insensitivity or globalness. When building a string to hold the pattern, you have to be careful with backslashes. Because, normally, backslashes are removed when a string is interpreted, any backslashes that must end up in the regular expression itself have to be escaped:

```
var digits = new RegExp("\\d+");
show(digits.test("101"));
```

The most important thing to know about regular expressions is that they exist, and can greatly enhance the power of your string-mangling code. They are so cryptic that you'll probably have to look up the details on them the first ten times you want to make use of them. Persevere, and you will soon be off-handedly writing expressions that look like occult gibberish

(Comic by Randall Munroe.)

1. In this case, the backslashes were not really necessary, because the characters occur between `[` and `]`, but it is easier to just escape them anyway, so you won't have to think about it.

Chapter 11:

Web programming: A crash course

You are probably reading this in a web browser, so you are likely to be at least a little familiar with the World Wide Web. This chapter contains a quick, superficial introduction to the various elements that make the web work, and the way they relate to JavaScript. The three after this one are more practical, and show some of the ways JavaScript can be used to inspect and change a web-page.

The Internet is, basically, just a computer network spanning most of the world. Computer networks make it possible for computers to send each other messages. The techniques that underlie networking are an interesting subject, but not the subject of this book. All you have to know is that, typically, one computer, which we will call the server, is waiting for other computers to start talking to it. Once another computer, the client, opens communications with this server, they will exchange whatever it is that needs to be exchanged using some specific language, a protocol.

The Internet is used to carry messages for *many* different protocols. There are protocols for chatting, protocols for file sharing, protocols used by malicious software to control the computer of the poor schmuck who installed it, and so on. The protocol that is of interest to us is that used by the World Wide Web. It is called HTTP, which stands for Hyper Text Transfer Protocol, and is used to retrieve web-pages and the files associated with them.

In HTTP communication, the server is the computer on which the web-page is stored. The client is the computer, such as yours, which asks the server for a page, so that it can display it. Asking for a page like this is called an 'HTTP request'.

Web-pages and other files that are accessible though the Internet are identified by URLs, which is an abbreviation of Universal Resource Locators. A URL looks like this:

```
http://acc6.its.brooklyn.cuny.edu/~phalsall/texts/taote-v3.html
```

It is composed of three parts. The start, `http://`, indicates that this URL uses the HTTP protocol. There are some other protocols, such as FTP (File Transfer Protocol), which also make use of URLs. The next part, `acc6.its.brooklyn.cuny.edu`, names the server on which this page can be found. The end of the URL, `/~phalsal/texts/taote-v3.html`, names a specific file on this server.

Most of the time, the World Wide Web is accessed using a browser. After typing a URL or clicking a link, the browser makes the appropriate HTTP request to the appropriate server. If all goes well, the server responds by sending a file back to the browser, who shows it to the user in one way or another.

When, as in the example, the retrieved file is an HTML document, it will be displayed as a web-page. We briefly discussed HTML in chapter 6, where we saw that it could refer to image files. In chapter 9, we found that HTML pages can also contain `<script>` tags to load files of JavaScript code. When showing an HTML document, a browser will fetch all these extra files from their servers, so it can add them to the document.

Although a URL is supposed to point at a file, it is possible for a web-server to do something more complicated than just looking up a file and sending it to the client. — It can process this file in some way first, or maybe there is no file at all, but only a program that, given an URL, has some way of generating the relevant document for it.

Programs that transform or generate documents on a server are a popular way to make web-pages less static. When a file is just a file, it is always the same, but when there is a program that builds it every time it is requested, it could be made to look different for each person, based on things like whether this person has logged in or specified certain preferences. This can also make managing the content of web-pages much easier — instead of adding a new HTML file whenever something new is put on a website, a new document is added to some central storage, and the program knows where to find it and how to show it to clients.

This kind of web programming is called server-side programming. It affects the document before it is sent to the user. In some cases, it is also practical to have a program that runs *after* the page has been sent, when the user is looking at it. This is called client-side programming, because the program runs on the client computer. Client-side web programming is what JavaScript was invented for.

Running programs client-side has an inherent problem. You can never really know in advance what kinds of programs the page you are visiting is going to run. If it can send information from your computer to others, damage something, or infiltrate your system, surfing the web would be a rather hazardous activity.

To solve this dilemma, browsers severely limit the things a JavaScript program may do. It is not allowed to look at your files, or to modify anything not related to the web-page it came with. Isolating a programming environment like this is called sand-boxing. Allowing the programs enough room to be useful, and at the same time restricting them enough to prevent them from doing harm is not an easy thing to do. Every few months, some JavaScript programmer comes up with a new way to circumvent the limitations and do something harmful or privacy-invading. The people responsible for the browsers respond by modifying their programs to make this trick impossible, and all is well again — until the next problem is discovered.

One of the first JavaScript tricks that became widely used is the `open` method of the `window` object. It takes a URL as an argument, and will open a new window showing that URL.

```
var perry = window.open("http://www.pbfcomics.com");
```

Unless you turned off pop-up blocking in chapter 6, there's a chance that this new window is blocked. There is a good reason pop-up blockers exist. Web-programmers, especially those trying to get people to pay attention to advertisements, have abused the poor `window.open` method so much that by now, most users hate it with a passion. It has its place though, and in this book we will be using it to show some example pages. As a general rule, your scripts should not open any new windows unless the user asked for them.

Note that, because `open` (just like `setTimeout` and company) is a method on the `window` object, the `window.` part can be left off. When a function is called 'normally', it is called as a method on the top-level object, which is what `window` is. Personally, I think `open` sounds a bit generic, so I'll usually type `window.open`, which makes it clear that it is a window that is being opened.

The value returned by `window.open` is a new window. This is the global object for the script running in that window, and contains all the standard things like the `Object` constructor and the `Math` object. But if you try to look at them, most browsers will (probably) not let you...

```
show(perry.Math);
```

This is part of the sand-boxing that I mentioned earlier. Pages opened by your browser might show information that is meant only for you, for example on sites where you logged in, and thus it would be bad if any random script could go and read them. The exception to this rule is pages opened on the same domain: When a script running on a page from `eloquentjavascript.net` opens another page on that same domain, it can do everything it wants to this page.

An opened window can be closed with its `close` method. If you didn't already close it yourself...

```
perry.close();
```

Other kinds of sub-documents, such as frames (documents-within-a-document), are also windows from the perspective of a JavaScript program, and have their own JavaScript environment. In fact, the environment that you have access to in the console belongs to a small invisible frame hidden somewhere on this page — this way, it is slightly harder for you to accidentally mess up the whole page.

Every window object has a `document` property, which contains an object representing the document shown in that window. This object contains, for example, a property `location`, with information about the URL of the document.

```
show(document.location.href);
```

Setting `document.location.href` to a new URL can be used to make the browser load another document. Another application of the `document` object is its `write` method. This method, when given a string argument, writes some HTML to the document. When it is used on a fully loaded document, it will replace the whole document by the given HTML, which is usually not what you want. The idea is to have a script call it while the document is being loaded, in which case the written HTML will be inserted into the document at the place of the `script` tag that triggered it. This is a simple way to add some dynamic elements to a page. For example, here is a trivially simple document showing the current time.

```
print(timeWriter);
var time = viewHTML(timeWriter);
```

```
time.close();
```

Often, the techniques shown in chapter 12 provide a cleaner and more versatile way to modify the document, but occasionally, `document.write` is the nicest, simplest way to do something.

Another popular application of JavaScript in web pages centers around forms. In case you

are not quite sure what the role of 'forms' is, let me give a quick summary.

A basic HTTP request is a simple request for a file. When this file is not really a passive file, but a server-side program, it can become useful to include information other than a filename in the request. For this purpose, HTTP requests are allowed to contain additional 'parameters'. Here is an example:

```
http://www.google.com/search?q=aztec%20empire
```

After the filename (`/search`), the URL continues with a question mark, after which the parameters follow. This request has one parameter, called `q` (for 'query', presumably), whose value is `aztec empire`. The `%20` part corresponds to a space. There are a number of characters that can not occur in these values, such as spaces, ampersands, or question marks. These are 'escaped' by replacing them with a `%` followed by their numerical value[1], which serves the same purpose as the backslashes used in strings and regular expressions, but is even more unreadable.

JavaScript provides functions `encodeURIComponent` and `decodeURIComponent` to add these codes to strings and remove them again.

```
var encoded = encodeURIComponent("aztec empire");
show(encoded);
show(decodeURIComponent(encoded));
```

When a request contains more than one parameter, they are separated by ampersands, as in...

```
http://www.google.com/search?q=aztec%20empire&lang=nl
```

A form, basically, is a way to make it easy for browser-users to create such parameterised URLs. It contains a number of fields, such as input boxes for text, checkboxes that can be 'checked' and 'unchecked', or thingies that allow you to choose from a given set of values. It also usually contains a 'submit' button and, invisible to the user, an 'action' URL to which it should be sent. When the submit button is clicked, or enter is pressed, the information that was entered in the fields is added to this action URL as parameters, and the browser will request this URL.

Here is the HTML for a simple form:

```
<form name="userinfo" method="get" action="info.html">
  <p>Please give us your information, so that we can send
  you spam.</p>
  <p>Name: <input type="text" name="name"/></p>
  <p>E-Mail: <input type="text" name="email"/></p>
  <p>Sex: <select name="sex">
            <option>Male</option>
            <option>Female</option>
            <option>Other</option>
          </select></p>
  <p><input name="send" type="submit" value="Send!"/></p>
</form>
```

The name of the form can be used to access it with JavaScript, as we shall see in a moment. The names of the fields determine the names of the HTTP parameters that are used to store their values. Sending this form might produce a URL like this:

```
http://planetspam.com/info.html?name=Ted&email=ted@zork.com&sex=Male
```

There are quite some other tags and properties that can be used in forms, but in this book we will stick with simple ones, so that we can concentrate on JavaScript.

The `method="get"` property of the example form shown above indicates that this form should encode the values it is given as URL parameters, as shown before. There is an alternative method for sending parameters, which is called `post`. An HTTP request using the `post` method contains, in addition to a URL, a block of data. A form using the `post` method puts the values of its parameters in this data block instead of in the URL.

When sending big chunks of data, the `get` method will result in URLs that are a mile wide, so `post` is usually more convenient. But the difference between the two methods is not just a question of convenience. Traditionally, `get` requests are used for requests that just ask the server for some document, while `post` requests are used to take an action that changes something on the server. For example, getting a list of recent messages on an Internet forum would be a `get` request, while adding a new message would be a `post` request. There is a good reason why most pages follow this distinction — programs that automatically explore the web, such as those used by search engines, will generally only make `get` requests. If changes to a site can be made by `get` requests, these well-meaning 'crawlers' could do all kinds of damage.

When the browser is displaying a page containing a form, JavaScript programs can inspect and modify the values that are entered in the form's fields. This opens up possibilities for all kinds of tricks, such as checking values before they are sent to the server, or automatically filling in certain fields.

The form shown above can be found in the file `example_getinfo.html`. Open it.

```
var form = window.open("example_getinfo.html");
```

When a URL does not contain a server name, is called a relative URL. Relative URLs are interpreted by the browser to refer to files on the same server as the current document. Unless they start with a slash, the path (or directory) of the current document is also retained, and the given path is appended to it.

We will be adding a validity check to the form, so that it only submits if the name field is not left empty and the e-mail field contains something that looks like a valid e-mail address. Because we no longer want the form to submit immediately when the 'Send!' button is pressed, its `type` property has been changed from `"submit"` to `"button"`, which turns it into a regular button with no effect. — Chapter 13 will show a *much* better way of doing this, but for now, we use the naive method.

To be able to work with the newly opened window (if you closed it, re-open it first), we 'attach' the console to it, like this:

```
attach(form);
```

After doing this, the code run from the console will be run in the given window. To verify that we are indeed working with the correct window, we can look at the document's `location` and `title` properties.

```
print(document.location.href);
print(document.title);
```

Because we have entered a new environment, previously defined variables, such as `form`, are no longer present.

```
show(form);
```

To get back to our starting environment, we can use the `detach` function (without arguments). But first, we have to add that validation system to the form.

Every HTML tag shown in a document has a JavaScript object associated with it. These objects can be used to inspect and manipulate almost every aspect of the document. In this chapter, we will work with the objects for forms and form fields, chapter 12 talks about these objects in more detail.

The `document` object has a property named `forms`, which contains links to all the forms in the document, by name. Our form has a property `name="userinfo"`, so it can be found under the property `userinfo`.

```
var userForm = document.forms.userinfo;
print(userForm.method);
print(userForm.action);
```

In this case, the properties `method` and `action` that were given to the HTML `form` tag are also present as properties of the JavaScript object. This is often the case, but not always: Some HTML properties are spelled differently in JavaScript, others are not present at all. Chapter 12 will show a way to get at all properties.

The object for the `form` tag has a property `elements`, which refers to an object containing the fields of the form, by name.

```
var nameField = userForm.elements.name;
nameField.value = "Eugène";
```

Text-input objects have a `value` property, which can be used to read and change their content. If you look at the form window after running the above code, you'll see that the name has been filled in.

Ex. 11.1 Being able to read the values of the form fields makes it possible to write a function `validInfo`, which takes a form object as its argument and returns a boolean value: `true` when the `name` field is not empty and the `email` field contains something that looks like an e-mail address, `false` otherwise. Write this function.

```
function validInfo(form) {
  return form.elements.name.value != "" &&
    /^.+@.+\.\w{2,3}$/.test(form.elements.email.value);
}

show(validInfo(document.forms.userinfo));
```

You did think to use a regular expression for the e-mail check, didn't you?

All we have to do now is determine what happens when people click the 'Send!' button. At the moment, it does not do anything at all. This can be remedied by setting its `onclick` property.

```
userForm.elements.send.onclick = function() {
  alert("Click.");
};
```

Just like the actions given to `setInterval` and `setTimeout` (chapter 8), the value stored in an `onclick` (or similar) property can be either a function or a string of JavaScript code. In this case, we give it a function that opens an alert window. Try clicking it.

Ex. 11.2 Finish the form validator by giving the button's `onclick` property a new value — a function that checks the form, submits when it is valid, or pops up a warning message when it is not. It will be useful to know that form objects have a `submit` method that takes no parameters and submits the form.

```
userForm.elements.send.onclick = function() {
  if (validInfo(userForm))
    userForm.submit();
  else
    alert("Give us a name and a valid e-mail address!");
};
```

Another trick related to form inputs, as well as other things that can be 'selected', such as buttons and links, is the `focus` method. When you know for sure that a user will want to start typing in a certain text field as soon as he enters the page, you can have your script start by placing the cursor in it, so he won't have to click it or select it in some other way.

```
userForm.elements.name.focus();
```

Because the form sits in another window, it may not be obvious that something was selected, depending on the browser you are using. Some pages also automatically make the cursor jump to the next field when it looks like you finished filling in one field — for example, when you type a zip code. This should not be overdone — it makes the page behave in a way the user does not expect. If he is used to pressing tab to move the cursor manually, or mistyped the last character and wants to remove it, such magic cursor-jumping is very annoying.

```
detach();
```

Test the validator. When you enter valid information and click the button, the form should submit. If the console was still attached to it, this will cause it to detach itself, because the page reloads and the JavaScript environment is replaced by a new one.

If you haven't closed the form window yet, this will close it.

```
form.close();
```

The above may look easy, but let me assure you, client-side web programming is no walk

in the park. It can, at times, be a very painful ordeal. Why? Because programs that are supposed to run on the client computer generally have to work for all popular browsers. Each of these browsers tends to work slightly different. To make things worse, each of them contains a unique set of problems. Do not assume that a program is bug-free just because it was made by a multi-billion dollar company. So it is up to us, the web-programmer, to rigorously test our programs, figure out what goes wrong, and find ways to work around it.

Some of you might think "I will just report any problems/bugs I find to the browser manufacturers, and they will certainly solve fix them immediately". These people are in for a major disappointment. The most recent version of Internet Explorer, the browser that is still used by some seventy percent of web-surfers (and that every web-developer likes to rag on) still contains bugs that have been known for over five years. Serious bugs, too.

But do not let that discourage you. With the right kind of obsessive-compulsive mindset, such problems provide wonderful challenges. And for those of us who do not like wasting our time, being careful and avoiding the obscure corners of the browser's functionality will generally prevent you from running into too much trouble.

Bugs aside, the by-design differences in interface between browsers still make for an interesting challenge. The current situation looks something like this: On the one hand, there are all the 'small' browsers: Firefox, Safari, and Opera are the most important ones, but there are more. These browsers all make a reasonable effort to adhere to a set of standards that have been developed, or are being developed, by the W3C, an organisation that tries to make the Web a less confusing place by defining standard interfaces for things like this. On the other hand, there is Internet Explorer, Microsoft's browser, which rose to dominance in a time when many of these standards did not really exist yet, and hasn't made much effort to adjust itself to what other people are doing.

In some areas, such as the way the content of an HTML document can be approached from JavaScript (chapter 12), the standards are based on the method that Internet Explorer invented, and things work more or less the same on all browsers. In other areas, such as the way events (mouse-clicks, key-presses, and such) are handled (chapter 13), Internet Explorer works radically different from other browsers.

For a long time, owing partially to the cluelessness of the average JavaScript developer, and partially to the fact that browser incompatibilities were much worse when browsers like Internet Explorer version 4 or 5 and old versions of Netscape were still common, the usual way to deal with such differences was to detect which browser the user was running, and litter code with alternate solutions for each browser — if this is Internet Explorer, do this, if this is Netscape, do that, and if this is other browser that we didn't think of, just hope for the best. You can imagine how hideous, confusing, and long such programs were.

Many sites would also just refuse to load when opened in a browser that was 'not supported'. This caused a few of the minor browsers to swallow their pride and pretend they were Internet Explorer, just so they would be allowed to load such pages. The properties of the `navigator` object contain information about the browser that a page was loaded in, but because of such lying this information is not particularly reliable. See what yours says[2]:

```
forEachIn(navigator, function(name, value) {
  print(name, " = ", value);
});
```

A better approach is to try and 'isolate' our programs from differences in browsers. If you need, for example, to find out more about an event, such as the clicks we handled by setting the `onclick` property of our send button, you have to look at the top-level object called `event` on Internet Explorer, but you have to use the first argument passed to the event-handling function on other browsers. To handle this, and a number of other differences related to events, one can write a helper function for attaching events to things, which takes care of all the plumbing and allows the event-handling functions themselves to be the same for all browsers. In chapter 13 we will write such a function.

(Note: The browser quirks mentioned in the following chapters refer to the state of affairs in early 2007, and might no longer be accurate on some points.)

These chapters will only give a somewhat superficial introduction to the subject of browser interfaces. They are not the main subject of this book, and they are complex enough to fill a thick book on their own. When you understand the basics of these interfaces (and understand something about HTML), it is not too hard to look for specific information online. The interface documentation for the Firefox and Internet Explorer browsers are a good place to start.

The information in the next chapters will not deal with the quirks of 'previous-generation' browsers. They deal with Internet Explorer 6, Firefox 1.5, Opera 9, Safari 3, or any more recent versions of the same browsers. Most of it will also probably be relevant to modern but obscure browsers such as Konqueror, but this has not been extensively checked. Fortunately, these previous-generation browsers have pretty much died out, and are hardly used anymore.

There is, however, a group of web-users that will still use a browser without JavaScript. A large part of this group consists of people using a regular graphical browser, but with JavaScript disabled for security reasons. Then there are people using textual browsers, or browsers for blind people. When working on a 'serious' site, it is often a good idea to start with a plain HTML system that works, and then add non-essential tricks and conveniences with JavaScript.

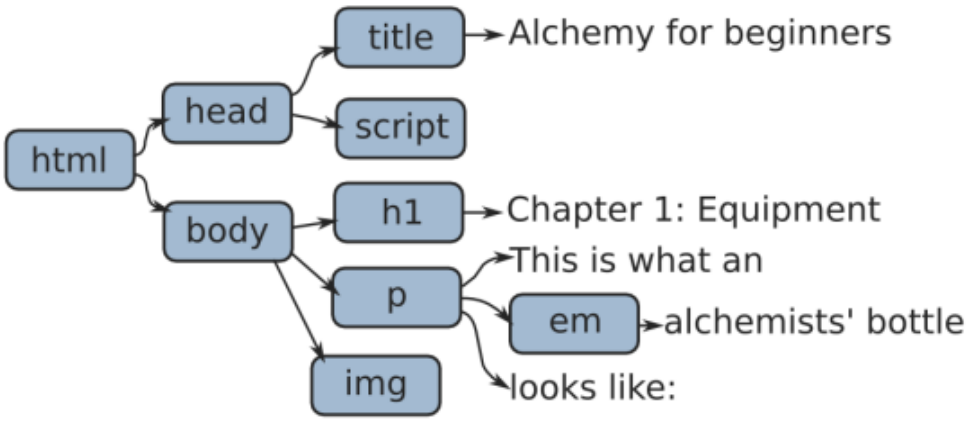
1. The value a character gets is decided by the ASCII standard, which assigns the numbers 0 to 127 to a set of letters and symbols used by the Latin alphabet. This standard is a precursor of the Unicode standard mentioned in chapter 2.
2. Some browsers seem to hide the properties of the `navigator` object, in which case this will print nothing.

Chapter 12:

The Document-Object Model

In chapter 11 we saw JavaScript objects referring to `form` and `input` tags from the HTML document. Such objects are part of a structure called the Document-Object Model (DOM). Every tag of the document is represented in this model, and can be looked up and interacted with.

HTML documents have what is called a hierarchical structure. Each element (or tag) except the top `<html>` tag is contained in another element, its parent. This element can in turn contain child elements. You can visualise this as a kind of family tree:



The document-object model is based on such a view of the document. Note that the tree contains two types of elements: Nodes, which are shown as blue boxes, and pieces of simple text. The pieces of text, as we will see, work somewhat different than the other elements. For one thing, they never have children.

Open the file `example_alchemy.html`, which contains the document shown in the picture, and attach the console to it.

```
attach(window.open("example_alchemy.html"));
```

The object for the root of the document tree, the `html` node, can be reached through the `documentElement` property of the `document` object. Most of the time, we need access to the `body` part of the document instead, which is at `document.body`.

The links between these nodes are available as properties of the node objects. Every DOM object has a `parentNode` property, which refers to the object in which it is contained, if any. These parents also have links pointing back to their children, but because there can be more than one child, these are stored in a pseudo-array called `childNodes`.

```
show(document.body);
show(document.body.parentNode);
show(document.body.childNodes.length);
```

For convenience, there are also links called `firstChild` and `lastChild`, pointing at the first and last child inside a node, or `null` when there are no children.

```
show(document.documentElement.firstChild);
show(document.documentElement.lastChild);
```

Finally, there are properties called `nextSibling` and `previousSibling`, which point at the nodes sitting 'next' to a node — nodes that are children of the same parent, coming before or after the current node. Again, when there is no such sibling, the value of these properties is `null`.

```
show(document.body.previousSibling);
show(document.body.nextSibling);
```

To find out whether a node represents a simple piece of text or an actual HTML node, we can look at its `nodeType` property. This contains a number, `1` for regular nodes and `3` for text nodes. There are actually other kinds of objects with a `nodeType`, such as the `document` object, which has `9`, but the most common use for this property is distinguishing between text nodes and other nodes.

```
function isTextNode(node) {
  return node.nodeType == 3;
}

show(isTextNode(document.body));
show(isTextNode(document.body.firstChild.firstChild));
```

Regular nodes have a property called `nodeName`, indicating the type of HTML tag that they represent. Text nodes, on the other hand, have a `nodeValue`, containing their text content.

```
show(document.body.firstChild.nodeName);
show(document.body.firstChild.firstChild.nodeValue);
```

The `nodeName`s are always capitalised, which is something you need to take into account if you ever want to compare them to something.

```
function isImage(node) {
  return !isTextNode(node) && node.nodeName == "IMG";
}

show(isImage(document.body.lastChild));
```

Ex. 12.1

Write a function `asHTML` which, when given a DOM node, produces a string representing the HTML text for that node and its children. You may ignore attributes, just show nodes as `<nodename>`. The `escapeHTML` function from chapter 10 is available to properly escape the content of text nodes.

Hint: Recursion!

```
function asHTML(node) {
  if (isTextNode(node))
    return escapeHTML(node.nodeValue);
  else if (node.childNodes.length == 0)
    return "<" + node.nodeName + "/>";
  else
    return "<" + node.nodeName + ">" +
           map(asHTML, node.childNodes).join("") +
           "</" + node.nodeName + ">";
}

print(asHTML(document.body));
```

Nodes, in fact, already have something similar to `asHTML`. Their `innerHTML` property can be used to retrieve the HTML text *inside* of the node, without the tags for the node itself. Some browsers also support `outerHTML`, which does include the node itself, but not all of them.

```
print(document.body.innerHTML);
```

Some of these properties can also be modified. Setting the `innerHTML` of a node or the `nodeValue` of a text-node will change its content. Note that, in the first case, the given string is interpreted as HTML, while in the second case it is interpreted as plain text.

```
document.body.firstChild.firstChild.nodeValue =
  "Chapter 1: The deep significance of the bottle";
```

Or ...

```
document.body.firstChild.innerHTML =
  "Did you know the 'blink' tag yet? <blink>Joy!</blink>";
```

We have been accessing nodes by going through a series of `firstChild` and `lastChild` properties. This can work, but it is verbose and easy to break — if we add another node at the start of our document, `document.body.firstChild` no longer refers to the `h1` element, and code which assumes it does will go wrong. On top of that, some browsers will add text-nodes for things like spaces and newlines between tags, while others do not, so that the exact layout of the DOM tree can vary.

An alternative to this is to give elements that you need to have access to an `id` attribute. In the example page, the picture has an id `"picture"`, and we can use this to look it up.

```
var picture = document.getElementById("picture");
show(picture.src);
picture.src = "img/ostrich.png";
```

When typing `getElementById`, note that the last letter is lowercase. Also, when typing it a lot, beware of carpal-tunnel syndrome. Because `document.getElementById` is a ridiculously long name for a very common operation, it has become a convention among JavaScript programmers to aggressively abbreviate it to `$`. `$`, as you might remember, is considered a letter by JavaScript, and is thus a valid variable name.

```
function $(id) {
  return document.getElementById(id);
}
show($("picture"));
```

DOM nodes also have a method `getElementsByTagName` (another nice, short name), which, when given a tag name, returns an array of all nodes of that type contained in the node it was called on.

```
show(document.body.getElementsByTagName("BLINK")[0]);
```

Another thing we can do with these DOM nodes is creating new ones ourselves. This makes it possible to add pieces to a document at will, which can be used to create some interesting effects. Unfortunately, the interface for doing this is extremely clumsy. But that can be remedied with some helper functions.

The `document` object has `createElement` and `createTextNode` methods. The first is used to create regular nodes, the second, as the name suggests, creates text nodes.

```
var secondHeader = document.createElement("H1");
var secondTitle = document.createTextNode("Chapter 2: Deep magic");
```

Next, we'll want to put the title name into the `h1` element, and then add the element to the document. The simplest way to do this is the `appendChild` method, which can be called on every (non-text) node.

```
secondHeader.appendChild(secondTitle);
document.body.appendChild(secondHeader);
```

Often, you will also want to give these new nodes some attributes. For example, an `img` (image) tag is rather useless without an `src` property telling the browser which image it should show. Most attributes can be approached directly as properties of the DOM nodes, but there are also methods `setAttribute` and `getAttribute`, which are used to access attributes in a more general way:

```
var newImage = document.createElement("IMG");
newImage.setAttribute("src", "img/Hiva Oa.png");
document.body.appendChild(newImage);
show(newImage.getAttribute("src"));
```

But, when we want to build more than a few simple nodes, it gets very tiresome to create every single node with a call to `document.createElement` or `document.createTextNode`, and then add its attributes and child nodes one by one. Fortunately, it is not hard to write a function to do most of the work for us. Before we do so, there is one little detail to take care of — the `setAttribute` method, while working fine on most browsers, does not always work on Internet Explorer. The names of a few HTML attributes already have a special meaning in JavaScript, and thus the corresponding object properties got an adjusted name. Specifically, the `class` attribute becomes `className`, `for` becomes `htmlFor`, and `checked` is renamed to `defaultChecked`. On Internet Explorer, `setAttribute` and `getAttribute` also work with these adjusted names, instead of the original HTML names, which can be confusing. On top of that the `style` attribute, which, along with `class`, will be discussed later in this chapter, can not be set with `setAttribute` on that browser.

A workaround would look something like this:

```
function setNodeAttribute(node, attribute, value) {
  if (attribute == "class")
    node.className = value;
  else if (attribute == "checked")
    node.defaultChecked = value;
  else if (attribute == "for")
    node.htmlFor = value;
  else if (attribute == "style")
    node.style.cssText = value;
  else
    node.setAttribute(attribute, value);
}
```

For every case where Internet Explorer deviates from other browsers, it does something that works in all cases. Don't worry about the details — this is the kind of ugly trick that we'd rather not need, but which non-conforming browsers force us to write. Having this, it is possible to write a simple function for building DOM elements.

```
function dom(name, attributes) {
  var node = document.createElement(name);
  if (attributes) {
    forEachIn(attributes, function(name, value) {
      setNodeAttribute(node, name, value);
    });
  }
  for (var i = 2; i < arguments.length; i++) {
    var child = arguments[i];
    if (typeof child == "string")
      child = document.createTextNode(child);
    node.appendChild(child);
  }
  return node;
}

var newParagraph =
  dom("P", null, "A paragraph with a ",
      dom("A", {href: "http://en.wikipedia.org/wiki/Alchemy"},
          "link"),
      " inside of it.");
document.body.appendChild(newParagraph);
```

The `dom` function creates a DOM node. Its first argument gives the tag name of the node, its second argument is an object containing the attributes of the node, or `null` when no attributes are needed. After that, any amount of arguments may follow, and these are added to the node as child nodes. When strings appear here, they are first put into a text node.

`appendChild` is not the only way nodes can be inserted into another node. When the new node should not appear at the end of its parent, the `insertBefore` method can be used to place it in front of another child node. It takes the new node as a first argument, and the existing child as second argument.

```
var link = newParagraph.childNodes[1];
newParagraph.insertBefore(dom("STRONG", null, "great "), link);
```

If a node that already has a `parentNode` is placed somewhere, it is automatically removed from its current position — nodes can not exist in the document in more than one place.

When a node must be replaced by another one, use the `replaceChild` method, which again takes the new node as first argument and the existing one as second argument.

```
newParagraph.replaceChild(document.createTextNode("lousy "),
                          newParagraph.childNodes[1]);
```

And, finally, there is `removeChild` to remove a child node. Note that this is called on the *parent* of the node to be removed, giving the child as argument.

```
newParagraph.removeChild(newParagraph.childNodes[1]);
```

Ex. 12.2 Write the convenient function `removeElement` which removes the DOM node it is given as an argument from its parent node.

```
function removeElement(node) {
  if (node.parentNode)
    node.parentNode.removeChild(node);
}

removeElement(newParagraph);
```

When creating new nodes and moving nodes around it is necessary to be aware of the following rule: Nodes are not allowed to be inserted into another document from the one in which they were created. This means that if you have extra frames or windows open, you can not take a piece of the document from one and move it to another, and nodes created with methods on one `document` object must stay in that document. Some browsers, notably Firefox, do not enforce this restriction, and thus a program which violates it will work fine in those browsers but break on others.

An example of something useful that can be done with this `dom` function is a program that takes JavaScript objects and summarises them in a table. Tables, in HTML, are created with a set of tags starting with `t`s, something like this:

```
<table>
  <tbody>
    <tr> <th>Tree </th> <th>Flowers</th> </tr>
    <tr> <td>Apple</td> <td>White  </td> </tr>
    <tr> <td>Coral</td> <td>Red    </td> </tr>
    <tr> <td>Pine </td> <td>None   </td> </tr>
  </tbody>
</table>
```

Each `tr` element is a row of the table. `th` and `td` elements are the cells of the table, `td`s are normal data cells, `th` cells are 'header' cells, which will be displayed in a slightly more prominent way. The `tbody` (table body) tag does not have to be included when a table is written as HTML, but when building a table from DOM nodes it should be added, because Internet Explorer refuses to display tables created without a `tbody`.

Ex. 12.3 The function `makeTable` takes two arrays as arguments. The first contains the JavaScript objects that it should summarise, and the second contains strings, which name the columns of the table and the properties of the objects that should be shown in these

columns. For example, the following will produce the table above:

```
makeTable([{Tree: "Apple", Flowers: "White"},
           {Tree: "Coral", Flowers: "Red"},
           {Tree: "Pine",  Flowers: "None"}],
          ["Tree", "Flowers"]);
```

Write this function.

```
function makeTable(data, columns) {
  var headRow = dom("TR");
  forEach(columns, function(name) {
    headRow.appendChild(dom("TH", null, name));
  });

  var body = dom("TBODY", null, headRow);
  forEach(data, function(object) {
    var row = dom("TR");
    forEach(columns, function(name) {
      row.appendChild(dom("TD", null, String(object[name])));
    });
    body.appendChild(row);
  });

  return dom("TABLE", null, body);
}

var table = makeTable(document.body.childNodes,
                      ["nodeType", "tagName"]);
document.body.appendChild(table);
```

Do not forget to convert the values from the objects to strings before adding them to the table — our `dom` function only understands strings and DOM nodes.

Closely tied to HTML and the document-object model is the topic of style-sheets. It is a big topic, and I will not discuss it entirely, but some understanding of style-sheets is necessary for a lot of interesting JavaScript techniques, so we will go over the basics.

In old-fashioned HTML, the only way to change the appearance of elements in a document was to give them extra attributes or to wrap them in extra tags, such as `center` to center them horizontally, or `font` to change the font style or colour. Most of the time, this meant that if you wanted the paragraphs or the tables in your document to look a certain way, you had to add a bunch of attributes and tags to *every single one of them*. This quickly adds a lot of noise to such documents, and makes them very painful to write or change by hand.

Of course, people being the inventive monkeys they are, someone came up with a solution. Style-sheets are a way to make statements like 'in this document, all paragraphs use the Comic Sans font, and are purple, and all tables have a thick green border'. You specify them once, at the top of the document or in a separate file, and they affect the whole document. Here, for example, is a style-sheet to make headers 22 points big and centered, and make paragraphs use the font and colour mentioned earlier, when they are of the 'ugly' class.

```
<style type="text/css">
  h1 {
    font-size: 22pt;
    text-align: center;
  }

  p.ugly {
    font-family: Comic Sans MS;
    color: purple;
  }
</style>
```

Classes are a concept related to styles. If you have different kinds of paragraphs, ugly ones and nice ones for example, setting the style for all `p` elements is not what you want, so classes can be used to distinguish between them. The above style will only be applied to paragraphs like this:

```
<p class="ugly">Mirror, mirror...</p>
```

And this is also the meaning of the `className` property which was briefly mentioned for the `setNodeAttribute` function. The `style` attribute can be used to add a piece of style directly to an element. For example, this gives our image a solid border 4 pixels ('px') wide.

```
setNodeAttribute($("picture"), "style",
                 "border-width: 4px; border-style: solid;");
```

There is much more to styles: Some styles are inherited by child nodes from parent nodes, and interfere with each other in complex and interesting ways, but for the purpose of DOM programming, the most important thing to know is that each DOM node has a `style` property, which can be used to manipulate the style of that node, and that there are a few kinds of styles that can be used to make nodes do extraordinary things.

This `style` property refers to an object, which has properties for all the possible elements of the style. We can, for example, make the picture's border green.

```
$("picture").style.borderColor = "green";
show($("picture").style.borderColor);
```

Note that in style-sheets, the words are separated by hyphens, as in `border-color`, while in JavaScript, capital letters are used to mark the different words, as in `borderColor`.

A very practical kind of style is `display: none`. This can be used to temporarily hide a node: When `style.display` is `"none"`, the element does not appear at all to the viewer of the document, even though it does exist. Later, `display` can be set to the empty string, and the element will re-appear.

```
$("picture").style.display = "none";
```

And, to get our picture back:

```
$("picture").style.display = "";
```

Another set of style types that can be abused in interesting ways are those related to positioning. In a simple HTML document, the browser takes care of determining the screen positions of all the elements — each element is put next to or below the elements that come before it, and nodes (generally) do not overlap.

When its `position` style is set to `"absolute"`, a node is taken out of the normal document 'flow'. It no longer takes up room in the document, but sort of floats above it. The `left` and `top` styles can then be used to influence its position. This can be used for various purposes, from making a node obnoxiously follow the mouse cursor to making 'windows' open on top of the rest of the document.

```
$("picture").style.position = "absolute";
var angle = 0;
var spin = setInterval(function() {
  angle += 0.1;
  $("picture").style.left = (100 + 100 * Math.cos(angle)) + "px";
  $("picture").style.top = (100 + 100 * Math.sin(angle)) + "px";
}, 100);
```

If you aren't familiar with goniometry, just believe me when I tell you that the cosine and sine stuff is used to build coordinates lying on the outline of a circle. Ten times per second, the angle at which we place the picture is changed, and new coordinates are computed. It is a common error, when setting styles like this, to forget to append `"px"` to your value. In most cases, setting a style to a number without a unit does not work, so you must add `"px"` for pixels, `"%"` for percent, `"em"` for 'ems' (the width of an `M` character), or `"pt"` for points.

(Now put the image to rest again...)

```
clearInterval(spin);
```

The place that is treated as 0,0 for the purpose of these positions depends on the place of the node in the document. When it is placed inside another node that has `position: absolute` or `position: relative`, the top left of this node is used. Otherwise, you get the top left corner of the document.

One last aspect of DOM nodes that is fun to play with is their size. There are style types called `width` and `height`, which can be used to set the absolute size of an element.

```
$("picture").style.width = "400px";
$("picture").style.height = "200px";
```

But, when you need to accurately set the size of an element, there is an tricky problem to take into account. Some browsers, in some circumstances, take these sizes to mean the outside size of the object, including any border and internal padding. Other browsers, in other circumstances, use the size of the space inside of the object instead, and do not count the width of borders and padding. Thus, if you set the size of an object that has a border or a padding, it will not always appear the same size.

Fortunately, you can inspect the inner and outer size of a node, which, when you really need to accurately size something, can be used to compensate for browser behaviour. The `offsetWidth` and `offsetHeight` properties give you the outer size of your element (the space it takes up in the document), while the `clientWidth` and `clientHeight` properties give the space inside of it, if any.

```
print("Outer size: ", $("picture").offsetWidth,
      " by ", $("picture").offsetHeight, " pixels.");
print("Inner size: ", $("picture").clientWidth,
      " by ", $("picture").clientHeight, " pixels.");
```

If you've followed through with all the examples in this chapter, and maybe did a few extra things by yourself, you will have completely mutilated the poor little document that we started with. Now let me moralise for a moment and tell you that you do not want to do this to real pages. The temptation to add all kinds of moving bling-bling will at times be strong. Resist it, or your pages shall surely become unreadable or even, if you go far enough, induce the occasional seizure.

Chapter 13:

Browser Events

To add interesting functionality to a web-page, just being able to inspect or modify the document is generally not enough. We also need to be able to detect what the user is doing, and respond to it. For this, we will use a thing called event handlers. Pressed keys are events, mouse clicks are events, even mouse motion can be seen as a series of events. In chapter 11, we added an `onclick` property to a button, in order to do something when it was pressed. This is a simple event handler.

The way browser events work is, fundamentally, very simple. It is possible to register handlers for specific event types and specific DOM nodes. Whenever an event occurs, the handler for that event, if any, is called. For some events, such as key presses, knowing just that the event occurred is not good enough, you also want to know which key was pressed. To store such information, every event creates an event object, which the handler can look at.

It is important to realise that, even though events can fire at any time, no two handlers ever run at the same moment. If other JavaScript code is still running, the browser waits until it finishes before it calls the next handler. This also holds for code that is triggered in other ways, such as with `setTimeout`. In programmer jargon, browser JavaScript is single-threaded, there are never two 'threads' running at the same time. This is, in most cases, a good thing. It is very easy to get strange results when multiple things happen at the same time.

An event, when not handled, can 'bubble' through the DOM tree. What this means is that if you click on, for example, a link in a paragraph, any handlers associated with the link are called first. If there are no such handlers, or these handlers do not indicate that they have finished handling the event, the handlers for the paragraph, which is the parent of the link, are tried. After that, the handlers for `document.body` get a turn. Finally, if no JavaScript handlers have taken care of the event, the browser handles it. When clicking a link, this means that the link will be followed.

So, as you see, events are easy. The only hard thing about them is that browsers, while all supporting more or less the same functionality, support this functionality through different interfaces. As usual, the most incompatible browser is Internet Explorer, which ignores the standard that most other browsers follow. After that, there is Opera, which does not properly support some useful events, such as the `onunload` event which fires when leaving a page, and sometimes gives confusing information about keyboard events.

There are four event-related actions one might want to take.

- Registering an event handler.
- Getting the event object.
- Extracting information from this object.
- Signalling that an event has been handled.

None of them work the same across all major browsers.

As a practice field for our event-handling, we open a document with a button and a text field. Keep this window open (and attached) for the rest of the chapter.

```
attach(window.open("example_events.html"));
```

The first action, registering a handler, can be done by setting an element's `onclick` (or `onkeypress`, and so on) property. This does in fact work across browsers, but it has an important drawback — you can only attach one handler to an element. Most of the time, one is enough, but there are cases, especially when a program has to be able to work together with other programs (which might also be adding handlers), that this is annoying.

In Internet Explorer, one can add a click handler to a button like this:

```
$("button").attachEvent("onclick", function(){print("Click!");});
```

On the other browsers, it goes like this:

```
$("button").addEventListener("click", function(){print("Click!");},
                             false);
```

Note how `"on"` is left off in the second case. The third argument to `addEventListener`, `false`, indicates that the event should 'bubble' through the DOM tree as normal. Giving `true` instead can be used to give this handler priority over the handlers 'beneath' it, but since Internet Explorer does not support such a thing, this is rarely useful.

Ex. 13.1

Write a function called `registerEventHandler` to wrap the incompatibilities of these two models. It takes three arguments: first a DOM node that the handler should be attached to, then the name of the event type, such as `"click"` or `"keypress"`, and finally the handler function.

To determine which method should be called, look for the methods themselves — if the DOM node has a method called `attachEvent`, you may assume that this is the correct method. Note that this is much preferable to directly checking whether the browser is Internet Explorer. If a new browser arrives which uses Internet Explorer's model, or Internet Explorer suddenly switches to the standard model, the code will still work. Both are rather unlikely, of course, but doing something in a smart way never hurts.

```
function registerEventHandler(node, event, handler) {
  if (typeof node.addEventListener == "function")
    node.addEventListener(event, handler, false);
  else
    node.attachEvent("on" + event, handler);
}

registerEventHandler($("button"), "click",
                     function(){print("Click (2)");});
```

Don't fret about the long, clumsy name. Later on, we will have to add an extra wrapper to wrap this wrapper, and it will have a shorter name.

It is also possible to do this check only once, and define `registerEventHandler` to hold a different function depending on the browser. This is more efficient, but a little strange.

```
if (typeof document.addEventListener == "function")
  var registerEventHandler = function(node, event, handler) {
    node.addEventListener(event, handler, false);
  };
else
  var registerEventHandler = function(node, event, handler) {
    node.attachEvent("on" + event, handler);
  };
```

Removing events works very much like adding them, but this time the methods `detachEvent` and `removeEventListener` are used. Note that, to remove a handler, you need to have access to the function you attached to it.

```
function unregisterEventHandler(node, event, handler) {
  if (typeof node.removeEventListener == "function")
    node.removeEventListener(event, handler, false);
  else
    node.detachEvent("on" + event, handler);
}
```

Exceptions produced by event handlers can, because of technical limitations, not be caught by the console. Thus, they are handled by the browser, which might mean they get hidden in some kind of 'error console' somewhere, or cause a message to pop up. When you write an event handler and it does not seem to work, it might be silently aborting because it causes some kind of error.

Most browsers pass the event object as an argument to the handler. Internet Explorer stores it in the top-level variable called `event`. When looking at JavaScript code, you will often come across something like `event || window.event`, which takes the local variable `event` or, if that is undefined, the top-level variable by that same name.

```
function showEvent(event) {
  show(event || window.event);
}

registerEventHandler($("textfield"), "keypress", showEvent);
```

Type a few characters in the field, look at the objects, and shut it up again:

```
unregisterEventHandler($("textfield"), "keypress", showEvent);
```

When the user clicks his mouse, three events are generated. First `mousedown`, at the moment the mouse button is pressed. Then, `mouseup`, at the moment it is released. And finally, `click`, to indicate something was clicked. When this happens two times in quick succession, a `dblclick` (double-click) event is also generated. Note that it is possible for the `mousedown` and `mouseup` events to happen some time apart — when the mouse button is held for a while.

When you attach an event handler to, for example, a button, the fact that it has been clicked is often all you need to know. When the handler, on the other hand, is attached to a node that has children, clicks from the children will 'bubble' up to it, and you will want to

find out which child has been clicked. For this purpose, event objects have a property called `target`... or `srcElement`, depending on the browser.

Another interesting piece of information are the precise coordinates at which the click occurred. Event objects related to the mouse contain `clientX` and `clientY` properties, which give the `x` and `y` coordinates of the mouse, in pixels, on the screen. Documents can scroll, though, so often these coordinates do not tell us much about the part of the document that the mouse is over. Some browsers provide `pageX` and `pageY` properties for this purpose, but others (guess which) do not. Fortunately, the information about the amount of pixels the document has been scrolled can be found in `document.body.scrollLeft` and `document.body.scrollTop`.

This handler, attached to the whole document, intercepts all mouse clicks, and prints some information about them.

```
function reportClick(event) {
  event = event || window.event;
  var target = event.target || event.srcElement;
  var pageX = event.pageX, pageY = event.pageY;
  if (pageX == undefined) {
    pageX = event.clientX + document.body.scrollLeft;
    pageY = event.clientY + document.body.scrollTop;
  }

  print("Mouse clicked at ", pageX, ", ", pageY,
        ". Inside element:");
  show(target);
}
registerEventHandler(document, "click", reportClick);
```

And get rid of it again:

```
unregisterEventHandler(document, "click", reportClick);
```

Obviously, writing all these checks and workarounds is not something you want to do in every single event handler. In a moment, after we have gotten acquainted with a few more incompatibilities, we will write a function to 'normalise' event objects to work the same across browsers.

It is also sometimes possible to find out which mouse button was pressed, using the `which` and `button` properties of event objects. Unfortunately, this is very unreliable — some browsers pretend mouses have only one button, others report right-clicks as clicks during which the control key was held down, and so on.

Apart from clicks, we might also be interested in the movement of the mouse. The `mousemove` event of a DOM node is fired whenever the mouse moves while it is over that element. There are also `mouseover` and `mouseout`, which are fired only when the mouse enters or leaves a node. For events of this last type, the `target` (or `srcElement`) property points at the node that the event is fired for, while the `relatedTarget` (or `toElement`, or `fromElement`) property gives the node that the mouse came from (for `mouseover`) or left to (for `mouseout`).

`mouseover` and `mouseout` can be tricky when they are registered on an element that has child nodes. Events fired for the child nodes will bubble up to the parent element, so you will also see a `mouseover` event when the mouse enters one of the child nodes. The `target` and `relatedTarget` properties can be used to detect (and ignore) such events.

For every key that the user presses, three events are generated: `keydown`, `keyup`, and `keypress`. In general, you should use the first two in cases where you really want to know which key was pressed, for example when you want to do something when the arrow keys are pressed. `keypress`, on the other hand, is to be used when you are interested in the character that is being typed. The reason for this is that there is often no character information in `keyup` and `keydown` events, and Internet Explorer does not generate a `keypress` event at all for special keys such as the arrow keys.

Finding out which key was pressed can be quite a challenge by itself. For `keydown` and `keyup` events, the event object will have a `keyCode` property, which contains a number. Most of the time, these codes can be used to identify keys in a reasonably browser-independant way. Finding out which code corresponds to which key can be done by simple experiments...

```
function printKeyCode(event) {
  event = event || window.event;
  print("Key ", event.keyCode, " was pressed.");
}

registerEventHandler($("textfield"), "keydown", printKeyCode);
```

```
unregisterEventHandler($("textfield"), "keydown", printKeyCode);
```

In most browsers, a single key code corresponds to a single *physical* key on your keyboard. The Opera browser, however, will generate different key codes for some keys depending on whether shift is pressed or not. Even worse, some of these shift-is-pressed codes are the same codes that are also used for other keys — shift-9, which on most keyboards is used to type a parenthesis, gets the same code as the down arrow, and as such is hard to distinguish from it. When this threatens to sabotage your programs, you can usually resolve it by ignoring key events that have shift pressed.

To find out whether the shift, control, or alt key was held during a key or mouse event, you can look at the `shiftKey`, `ctrlKey`, and `altKey` properties of the event object.

For `keypress` events, you will want to know which character was typed. The event object will have a `charCode` property, which, if you are lucky, contains the Unicode number corresponding to the character that was typed, which can be converted to a 1-character string by using `String.fromCharCode`. Unfortunately, some browsers do not define this property, or define it as `0`, and store the character code in the `keyCode` property instead.

```
function printCharacter(event) {
  event = event || window.event;
  var charCode = event.charCode;
  if (charCode == undefined || charCode === 0)
    charCode = event.keyCode;
  print("Character '", String.fromCharCode(charCode), "'");
}

registerEventHandler($("textfield"), "keypress", printCharacter);
```

```
unregisterEventHandler($("textfield"), "keypress", printCharacter);
```

An event handler can 'stop' the event it is handling. There are two different ways to do this. You can prevent the event from bubbling up to parent nodes and the handlers

defined on those, and you can prevent the browser from taking the standard action associated with such an event. It should be noted that browsers do not always follow this — preventing the default behaviour for the pressing of certain 'hotkeys' will, on many browsers, not actually keep the browser from executing the normal effect of these keys.

On most browsers, stopping event bubbling is done with the `stopPropagation` method of the event object, and preventing default behaviour is done with the `preventDefault` method. For Internet Explorer, this is done by setting the `cancelBubble` property of this object to `true`, and the `returnValue` property to `false`, respectively.

And that was the last of the long list of incompatibilities that we will discuss in this chapter. Which means that we can finally write the event normaliser function and move on to more interesting things.

```
function normaliseEvent(event) {
  if (!event.stopPropagation) {
    event.stopPropagation = function() {this.cancelBubble = true;};
    event.preventDefault = function() {this.returnValue = false;};
  }
  if (!event.stop) {
    event.stop = function() {
      this.stopPropagation();
      this.preventDefault();
    };
  }

  if (event.srcElement && !event.target)
    event.target = event.srcElement;
  if ((event.toElement || event.fromElement) && !event.relatedTarget)
    event.relatedTarget = event.toElement || event.fromElement;
  if (event.clientX != undefined && event.pageX == undefined) {
    event.pageX = event.clientX + document.body.scrollLeft;
    event.pageY = event.clientY + document.body.scrollTop;
  }
  if (event.type == "keypress") {
    if (event.charCode === 0 || event.charCode == undefined)
      event.character = String.fromCharCode(event.keyCode);
    else
      event.character = String.fromCharCode(event.charCode);
  }

  return event;
}
```

A `stop` method is added, which cancels both the bubbling and the default action of the event. Some browsers already provide this, in which case we leave it as it is.

Next we can write convenient wrappers for `registerEventHandler` and `unregisterEventHandler`:

```
function addHandler(node, type, handler) {
  function wrapHandler(event) {
    handler(normaliseEvent(event || window.event));
  }
  registerEventHandler(node, type, wrapHandler);
  return {node: node, type: type, handler: wrapHandler};
}

function removeHandler(object) {
  unregisterEventHandler(object.node, object.type, object.handler);
}

var blockQ = addHandler($("textfield"), "keypress", function(event) {
  if (event.character.toLowerCase() == "q")
    event.stop();
});
```

The new `addHandler` function wraps the handler function it is given in a new function, so it can take care of normalising the event objects. It returns an object that can be given to `removeHandler` when we want to remove this specific handler. Try typing a '`q`' in the text field.

```
removeHandler(blockQ);
```

Armed with `addHandler` and the `dom` function from the last chapter, we are ready for more challenging feats of document-manipulation. As an exercise, we will implement the game known as Sokoban. This is something of a classic, but you may not have seen it before. The rules are this: There is a grid, made up of walls, empty space, and one or more 'exits'. On this grid, there are a number of crates or stones, and a little dude that the player controls. This dude can be moved horizontally and vertically into empty squares, and can push the boulders around, provided that there is empty space behind them. The goal of the game is to move a given number of boulders into the exits.

Just like the terraria from chapter 8, a Sokoban level can be represented as text. The variable `sokobanLevels`, in the `example_events.html` window, contains an array of level objects. Each level has a property `field`, containing a textual representation of the level, and a property `boulders`, indicating the amount of boulders that must be expelled to finish the level.

```
show(sokobanLevels.length);
show(sokobanLevels[1].boulders);
forEach(sokobanLevels[1].field, print);
```

In such a level, the `#` characters are walls, spaces are empty squares, `0` characters are used for for boulders, an `@` for the starting location of the player, and a `*` for the exit.

But, when playing the game, we do not want to be looking at this textual representation. Instead, we will put a table into the document. I made small style-sheet (sokoban.css, if you are curious what it looks like) to give the cells of this table a fixed square size, and added it to the example document. Each of the cells in this table will get a background image, representing the type of the square (empty, wall, or exit). To show the location of the player and the boulders, images are added to these table cells, and moved to different cells as appropriate.

It would be possible to use this table as the main representation of our data — when we

want to look whether there is a wall in a given square, we just inspect the background of the appropriate table cell, and to find the player, we just search for the image node with the correct `src` property. In some cases, this approach is practical, but for this program I chose to keep a separate data structure for the grid, because it makes things much more straightforward.

This data structure is a two-dimensional grid of objects, representing the squares of the playing field. Each of the objects must store the type of background it has and whether there is a boulder or player present in that cell. It should also contain a reference to the table cell that is used to display it in the document, to make it easy to move images in and out of this table cell.

That gives us two kinds of objects — one to hold the grid of the playing field, and one to represent the individual cells in this grid. If we want the game to also do things like moving the next level at the appropriate moment, and being able to reset the current level when you mess up, we will also need a 'controller' object, which creates or removes the field objects at the appropriate moment. For convenience, we will be using the prototype approach outlined at the end of chapter 8, so object types are just prototypes, and the `create` method, rather than the `new` operator, is used to make new objects.

Let us start with the objects representing the squares of the game's field. They are responsible for setting the background of their cells correctly, and adding images as appropriate. The `img/sokoban/` directory contains a set of images, based on another ancient game, which will be used to visualise the game. For a start, the `Square` prototype could look like this.

```
var Square = {
  construct: function(character, tableCell) {
    this.background = "empty";
    if (character == "#")
      this.background = "wall";
    else if (character == "*")
      this.background = "exit";

    this.tableCell = tableCell;
    this.tableCell.className = this.background;

    this.content = null;
    if (character == "0")
      this.content = "boulder";
    else if (character == "@")
      this.content = "player";

    if (this.content != null) {
      var image = dom("IMG", {src: "img/sokoban/" +
                                   this.content + ".gif"});
      this.tableCell.appendChild(image);
    }
  },

  hasPlayer: function() {
    return this.content == "player";
  },
  hasBoulder: function() {
    return this.content == "boulder";
  },
  isEmpty: function() {
    return this.content == null && this.background == "empty";
  },
  isExit: function() {
    return this.background == "exit";
  }
};

var testSquare = Square.create("@", dom("TD"));
show(testSquare.hasPlayer());
```

The `character` argument to the constructor will be used to transform characters from the level blueprints into actual `Square` objects. To set the background of the cells, style-sheet classes are used (defined in sokoban.css), which are assigned to the `td` elements' `className` property.

The methods like `hasPlayer` and `isEmpty` are a way to 'isolate' the code that uses objects of this type from the internals of the objects. They are not strictly necessary in this case, but they will make the other code look better.

Ex. 13.2

Add methods `moveContent` and `clearContent` to the `Square` prototype. The first one takes another `Square` object as an argument, and moves the content of the `this` square into the argument by updating the `content` properties and moving the image node associated with this content. This will be used to move boulders and players around the grid. It may assume the square is not currently empty. `clearContent` removes the content from the square without moving it anywhere. Note that the `content` property for empty squares contains `null`.

The `removeElement` function we defined in chapter 12 is available in this chapter too, for your node-removing convenience. You may assume that the images are the only child

nodes of the table cells, and can thus be reached through, for example, `this.tableCell.lastChild`.

```
Square.moveContent = function(target) {
  target.content = this.content;
  this.content = null;
  target.tableCell.appendChild(this.tableCell.lastChild);
};
Square.clearContent = function() {
  this.content = null;
  removeElement(this.tableCell.lastChild);
};
```

The next object type will be called `SokobanField`. Its constructor is given an object from the `sokobanLevels` array, and is responsible for building both a table of DOM nodes and a grid of `Square` objects. This object will also take care of the details of moving the player and boulders around, through a `move` method that is given an argument indicating which way the player wants to move.

To identify the individual squares, and to indicate directions, we will again use the `Point` object type from chapter 8, which, as you might remember, has an `add` method.

The base of the field prototype looks like this:

```
var SokobanField = {
  construct: function(level) {
    var tbody = dom("TBODY");
    this.squares = [];
    this.bouldersToGo = level.boulders;

    for (var y = 0; y < level.field.length; y++) {
      var line = level.field[y];
      var tableRow = dom("TR");
      var squareRow = [];
      for (var x = 0; x < line.length; x++) {
        var tableCell = dom("TD");
        tableRow.appendChild(tableCell);
        var square = Square.create(line.charAt(x), tableCell);
        squareRow.push(square);
        if (square.hasPlayer())
          this.playerPos = new Point(x, y);
      }
      tbody.appendChild(tableRow);
      this.squares.push(squareRow);
    }

    this.table = dom("TABLE", {"class": "sokoban"}, tbody);
    this.score = dom("DIV", null, "...");
    this.updateScore();
  },

  getSquare: function(position) {
    return this.squares[position.y][position.x];
  },
  updateScore: function() {
    this.score.firstChild.nodeValue = this.bouldersToGo +
                                      " boulders to go.";
  },
  won: function() {
    return this.bouldersToGo <= 0;
  }
};

var testField = SokobanField.create(sokobanLevels[0]);
show(testField.getSquare(new Point(10, 2)).content);
```

The constructor goes over the lines and characters in the level, and stores the `Square` objects in the `squares` property. When it encounters the square with the player, it saves this position as `playerPos`, so that we can easily find the square with the player later on. `getSquare` is used to find a `Square` object corresponding to a certain `x,y` position on the field. Note that it doesn't take the edges of the field into account — to avoid writing some boring code, we assume that the field is properly walled off, making it impossible to walk out of it.

The word `"class"` in the `dom` call that makes the `table` node is quoted as a string. This is necessary because `class` is a 'reserved word' in JavaScript, and may not be used as a variable or property name.

The amount of boulders that have to be cleared to win the level (this may be less than the total amount of boulders on the level) is stored in `bouldersToGo`. Whenever a boulder is brought to the exit, we can subtract 1 from this, and see whether the game is won yet. To show the player how he is doing, we will have to show this amount somehow. For this purpose, a `div` element with text is used. `div` nodes are containers without inherent markup. The score text can be updated with the `updateScore` method. The `won` method will be used by the controller object to determine when the game is over, so the player can

move on to the next level.

If we want to actually see the playing field and the score, we will have to insert them into the document somehow. That is what the `place` method is for. We'll also add a `remove` method to make it easy to remove a field when we are done with it.

```
SokobanField.place = function(where) {
  where.appendChild(this.score);
  where.appendChild(this.table);
};
SokobanField.remove = function() {
  removeElement(this.score);
  removeElement(this.table);
};

testField.place(document.body);
```

If all went well, you should see a Sokoban field now.

Ex. 13.3

But this field doesn't do very much yet. Add a method called `move`. It takes a `Point` object specifying the move as argument (for example `-1,0` to move left), and takes care of moving the game elements in the correct way.

The correct way is this: The `playerPos` property can be used to determine where the player is trying to move. If there is a boulder here, look at the square behind this boulder. When there is an exit there, remove the boulder and update the score. When there is empty space there, move the boulder into it. Next, try to move the player. If the square he is trying to move into is not empty, ignore the move.

```
SokobanField.move = function(direction) {
  var playerSquare = this.getSquare(this.playerPos);
  var targetPos = this.playerPos.add(direction);
  var targetSquare = this.getSquare(targetPos);

  // Possibly pushing a boulder
  if (targetSquare.hasBoulder()) {
    var pushTarget = this.getSquare(targetPos.add(direction));
    if (pushTarget.isEmpty()) {
      targetSquare.moveContent(pushTarget);
    }
    else if (pushTarget.isExit()) {
      targetSquare.moveContent(pushTarget);
      pushTarget.clearContent();
      this.bouldersToGo--;
      this.updateScore();
    }
  }
  // Moving the player
  if (targetSquare.isEmpty()) {
    playerSquare.moveContent(targetSquare);
    this.playerPos = targetPos;
  }
};
```

By taking care of boulders first, the move code can work the same way when the player is moving normally and when he is pushing a boulder. Note how the square behind the boulder is found by adding the `direction` to the `playerPos` twice. Test it by moving left two squares:

```
testField.move(new Point(-1, 0));
testField.move(new Point(-1, 0));
```

If that worked, we moved a boulder into a place from which we can't get it out anymore, so we'd better throw this field away.

```
testField.remove();
```

All the 'game logic' has been taken care of now, and we just need a controller to make it playable. The controller will be an object type called `SokobanGame`, which is responsible for the following things:

- Preparing a place where the game field can be placed.
- Building and removing `SokobanField` objects.
- Capturing key events and calling the `move` method on current field with the correct argument.
- Keeping track of the current level number and moving to the next level when a level is won.
- Adding buttons to reset the current level or the whole game (back to level 0).

We start again with an unfinished prototype.

```
var SokobanGame = {
  construct: function(place) {
    this.level = null;
    this.field = null;

    var newGame = dom("BUTTON", null, "New game");
    addHandler(newGame, "click", method(this, "newGame"));
    var reset = dom("BUTTON", null, "Reset level");
    addHandler(reset, "click", method(this, "reset"));
    this.container = dom("DIV", null,
                         dom("H1", null, "Sokoban"),
                         dom("DIV", null, newGame, " ", reset));
    place.appendChild(this.container);

    addHandler(document, "keydown", method(this, "keyDown"));
    this.newGame();
  },

  newGame: function() {
    this.level = 0;
    this.reset();
  },
  reset: function() {
    if (this.field)
      this.field.remove();
    this.field = SokobanField.create(sokobanLevels[this.level]);
    this.field.place(this.container);
  },

  keyDown: function(event) {
    // To be filled in
  }
};
```

The constructor builds a `div` element to hold the field, along with two buttons and a title. Note how `method` is used to attach methods on the `this` object to events.

We can put a Sokoban game into our document like this:

```
var sokoban = SokobanGame.create(document.body);
```

Ex. 13.4

All that is left to do now is filling in the key event handler. Replace the `keyDown` method of the prototype with one that detects presses of the arrow keys and, when it finds them, moves the player in the correct direction. The following `Dictionary` will probably come in handy:

```
var arrowKeyCodes = new Dictionary({
  37: new Point(-1, 0), // left
  38: new Point(0, -1), // up
  39: new Point(1, 0),  // right
  40: new Point(0, 1)   // down
});
```

After an arrow key has been handled, check `this.field.won()` to find out if that was the winning move. If the player won, use `alert` to show a message, and go to the next level. If there is no next level (check `sokobanLevels.length`), restart the game instead.

It is probably wise to stop the events for key presses after handling them, otherwise pressing arrow-up and arrow-down will scroll your window, which is rather annoying.

```
SokobanGame.keyDown = function(event) {
  if (arrowKeyCodes.contains(event.keyCode)) {
    event.stop();
    this.field.move(arrowKeyCodes.lookup(event.keyCode));
    if (this.field.won()) {
      if (this.level < sokobanLevels.length - 1) {
        alert("Excellent! Going to the next level.");
        this.level++;
        this.reset();
      }
      else {
        alert("You win! Game over.");
        this.newGame();
      }
    }
  }
};
```

It has to be noted that capturing keys like this — adding a handler to the `document` and stopping the events that you are looking for — is not very nice when there are other elements in the document. For example, try moving the cursor around in the text field at the top of the document. — It won't work, you'll only move the little man in the Sokoban game. If a game like this were to be used in a real site, it is probably best to put it in a frame or window of its own, so that it only grabs events aimed at its own window.

Ex. 13.5

When brought to the exit, the boulders vanish rather abrubtly. By modifying the `Square.clearContent` method, try to show a 'falling' animation for boulders that are about to be removed. Make them grow smaller for a moment before, and then disappear. You can use `style.width = "50%"`, and similarly for `style.height`, to make an image appear, for example, half as big as it usually is.

We can use `setInterval` to handle the timing of the animation. Note that the method

makes sure to clear the interval after it is done. If you don't do that, it will continue wasting your computer's time until the page is closed.

```
Square.clearContent = function() {
  self.content = null;
  var image = this.tableCell.lastChild;
  var size = 100;

  var animate = setInterval(function() {
    size -= 10;
    image.style.width = size + "%";
    image.style.height = size + "%";

    if (size < 60) {
      clearInterval(animate);
      removeElement(image);
    }
  }, 70);
};
```

Now, if you have a few hours to waste, try finishing all levels.

Other event types that can be useful are `focus` and `blur`, which are fired on elements that can be 'focused', such as form inputs. `focus`, obviously, happens when you put the focus on the element, for example by clicking on it. `blur` is JavaScript-speak for 'unfocus', and is fired when the focus leaves the element.

```
addHandler($("textfield"), "focus", function(event) {
  event.target.style.backgroundColor = "yellow";
});
addHandler($("textfield"), "blur", function(event) {
  event.target.style.backgroundColor = "";
});
```

Another event related to form inputs is `change`. This is fired when the content of the input has changed... except that for some inputs, such as text inputs, it does not fire until the element is unfocused.

```
addHandler($("textfield"), "change", function(event) {
  print("Content of text field changed to '",
        event.target.value, "'.");
});
```

You can type all you want, the event will only fire when you click outside of the input, press tab, or unfocus it in some other way.

Forms also have a `submit` event, which is fired when they submit. It can be stopped to prevent the submit from taking place. This gives us a *much* better way to do the form validation we saw in the previous chapter. You just register a `submit` handler, which stops the event when the content of the form is not valid. That way, when the user does not have JavaScript enabled, the form will still work, it just won't have instant validation.

Window objects have a `load` event that fires when the document is fully loaded, which can be useful if your script needs to do some kind of initialisation that has to wait until the whole document is present. For example, the scripts on the pages for this book go over the current chapter to hide solutions to exercises. You can't do that when the exercises are not loaded yet. There is also an `unload` event, firing when the user leaves the

document, but this is not properly supported by all browsers.

Most of the time it is best to leave the laying out of a document to the browser, but there are effects that can only be produced by having a piece of JavaScript set the exact sizes of some nodes in a document. When you do this, make sure you also listen for `resize` events on the window, and re-calculate the sizes of your element every time the window is resized.

Finally, I have to tell you something about event handlers that you would rather not know. The Internet Explorer browser (which means, at the time of writing, the browser used by a majority of web-surfers) has a bug that causes values to not be cleaned up as normal: Even when they are no longer used, they stay in the machine's memory. This is known as a memory leak, and, once enough memory has been leaked, will seriously slow down a computer.

When does this leaking occur? Due to a deficiency in Internet Explorer's garbage collector, the system whose purpose it is to reclaim unused values, when you have a DOM node that, through one of its properties or in a more indirect way, refers to a normal JavaScript object, and this object, in turn, refers back to that DOM node, both objects will not be collected. This has something to do with the fact that DOM nodes and other JavaScript objects are collected by different systems — the system that cleans up DOM nodes will take care to leave any nodes that are still referenced by JavaScript objects, and vice versa for the system that collects normal JavaScript values.

As the above description shows, the problem is not specifically related to event handlers. This code, for example, creates a bit of un-collectable memory:

```
var jsObject = {link: document.body};
document.body.linkBack = jsObject;
```

Even after such an Internet Explorer browser goes to a different page, it will still hold on to the `document.body` shown here. The reason this bug is often associated with event handlers is that it is extremely easy to make such circular links when registering a handler. The DOM node keeps references to its handlers, and the handler, most of the time, has a reference to the DOM node. Even when this reference is not intentionally made, JavaScript's scoping rules tend to add it implicitly. Consider this function:

```
function addAlerter(element) {
  addHandler(element, "click", function() {
    alert("Alert! ALERT!");
  });
}
```

The anonymous function that is created by the `addAlerter` function can 'see' the `element` variable. It doesn't use it, but that does not matter — just because it can see it, it will have a reference to it. By registering this function as an event handler on that same `element` object, we have created a circle.

There are three ways to deal with this problem. The first approach, a very popular one, is to ignore it. Most scripts will only leak a little bit, so it takes a long time and a lot of pages before the problems become noticeable. And, when the problems are so subtle, who's going to hold *you* responsible? Programmers given to this approach will often searingly denounce Microsoft for their shoddy programming, and state that the problem is not their fault, so *they* shouldn't be fixing it.

Such reasoning is not entirely without merit, of course. But when half your users are having problems with the web-pages you make, it is hard to deny that there is a practical problem. Which is why people working on 'serious' sites usually make an attempt not to leak any memory. Which brings us to the second approach: Painstakingly making sure that no circular references between DOM objects and regular objects are created. This means, for example, rewriting the above handler like this:

```
function addAlerter(element) {
  addHandler(element, "click", function() {
    alert("Alert! ALERT!");
  });
  element = null;
}
```

Now the `element` variable no longer points at the DOM node, and the handler will not leak. This approach is viable, but requires the programmer to *really* pay attention.

The third solution, finally, is to not worry too much about creating leaky structures, but to make sure to clean them up when you are done with them. This means unregistering any event handlers when they are no longer needed, and registering an `onunload` event to unregister the handlers that are needed until the page is unloaded. It is possible to extend an event-registering system, like our `addHandler` function, to automatically do this. When taking this approach, you must keep in mind that event handlers are not the only possible source of memory leaks — adding properties to DOM node objects can cause similar problems.

Chapter 14:

HTTP requests

As mentioned in chapter 11, communication on the World Wide Web happens over the HTTP protocol. A simple request might look like this:

```
GET /files/fruit.txt HTTP/1.1
Host: eloquentjavascript.net
User-Agent: The Imaginary Browser
```

Which asks for the file `files/fruit.txt` from the server at `eloquentjavascript.net`. In addition, it specifies that this request uses version 1.1 of the HTTP protocol — version 1.0 is also still in use, and works slightly differently. The `Host` and `User-Agent` lines follow a pattern: They start with a word that identifies the information they contain, followed by a colon and the actual information. These are called 'headers'. The `User-Agent` header tells the server which browser (or other kind of program) is being used to make the request. Other kinds of headers are often sent along, for example to state the types of documents that the client can understand, or the language that it prefers.

When given the above request, the server might send the following response:

```
HTTP/1.1 200 OK
Last-Modified: Mon, 23 Jul 2007 08:41:56 GMT
Content-Length: 24
Content-Type: text/plain

apples, oranges, bananas
```

The first line indicates again the version of the HTTP protocol, followed by the status of the request. In this case the status code is `200`, meaning 'OK, nothing out of the ordinary happened, I am sending you the file'. This is followed by a few headers, indicating (in this case) the last time the file was modified, its length, and its type (plain text). After the headers you get a blank line, followed by the file itself.

Apart from requests starting with `GET`, which indicates the client just wants to fetch a document, the word `POST` can also be used to indicate some information will be sent along with the request, which the server is expected to process in some way.[1]

When you click a link, submit a form, or in some other way encourage your browser to go to a new page, it will do an HTTP request and immediately unload the old page to show the newly loaded document. In typical situations, this is just what you want — it is how the web traditionally works. Sometimes, however, a JavaScript program wants to communicate with the server without re-loading the page. The 'Load' button in the console, for example, can load files without leaving the page.

To be able to do things like that, the JavaScript program must make the HTTP request itself. Contemporary browsers provide an interface for this. As with opening new windows, this interface is subject to some restrictions. To prevent a script from doing anything scary, it is only allowed to make HTTP requests to the domain that the current page came from.

An object used to make an HTTP request can, on most browsers, be created by doing `new`

`XMLHttpRequest()`. Older versions of Internet Explorer, which originally invented these objects, require you to do `new ActiveXObject("Msxml2.XMLHTTP")` or, on even older versions, `new ActiveXObject("Microsoft.XMLHTTP")`. `ActiveXObject` is Internet Explorer's interface to various kinds of browser add-ons. We are already used to writing incompatibility-wrappers by now, so let us do so again:

```
function makeHttpObject() {
  try {return new XMLHttpRequest();}
  catch (error) {}
  try {return new ActiveXObject("Msxml2.XMLHTTP");}
  catch (error) {}
  try {return new ActiveXObject("Microsoft.XMLHTTP");}
  catch (error) {}

  throw new Error("Could not create HTTP request object.");
}

show(typeof(makeHttpObject()));
```

The wrapper tries to create the object in all three ways, using `try` and `catch` to detect which ones fail. If none of the ways work, which might be the case on older browsers or browsers with strict security settings, it raises and error.

Now why is this object called an *XML* HTTP request? This is a bit of a misleading name. XML is a way to store textual data. It uses tags and attributes like HTML, but is more structured and flexible — to store your own kinds of data, you may define your own types of XML tags. These HTTP request objects have some built-in functionality for dealing with retrieved XML documents, which is why they have XML in their name. They can also handle other types of documents, though, and in my experience they are used just as often for non-XML requests.

Now that we have our HTTP object, we can use it to make a request similar the example shown above.

```
var request = makeHttpObject();
request.open("GET", "files/fruit.txt", false);
request.send(null);
print(request.responseText);
```

The `open` method is used to configure a request. In this case we choose to make a `GET` request for our `fruit.txt` file. The URL given here is relative, it does not contain the `http://` part or a server name, which means it will look for the file on the server that the current document came from. The third parameter, `false`, will be discussed in a moment. After `open` has been called, the actual request can be made with the `send` method. When the request is a `POST` request, the data to be sent to the server (as a string) can be passed to this method. For `GET` requests, one should just pass `null`.

After the request has been made, the `responseText` property of the request object contains the content of the retrieved document. The headers that the server sent back can be inspected with the `getResponseHeader` and `getAllResponseHeaders` functions. The first looks up a specific header, the second gives us a string containing all the headers. These can occasionally be useful to get some extra information about the document.

```
print(request.getAllResponseHeaders());
show(request.getResponseHeader("Date"));
```

If, for some reason, you want to add headers to the request that is sent to the server, you can do so with the `setRequestHeader` method. This takes two strings as arguments, the name and the value of the header.

The response code, which was `200` in the example, can be found under the `status` property. When something went wrong, this cryptic code will indicate it. For example, `404` means the file you asked for did not exist. The `statusText` contains a slightly less cryptic description of the status.

```
show(request.status);
show(request.statusText);
```

When you want to check whether a request succeeded, comparing the `status` to `200` is usually enough. In theory, the server might in some situations return the code `304` to indicate that the older version of the document, which the browser has stored in its 'cache', is still up to date. But it seems that browsers shield you from this by setting the `status` to `200` even when it is `304`. Also, if you are doing a request over a non-HTTP protocol[2], such as FTP, the `status` will not be usable because the protocol does not use HTTP status codes.

When a request is done as in the example above, the call to the `send` method does not return until the request is finished. This is convenient, because it means the `responseText` is available after the call to `send`, and we can start using it immediately. There is a problem, though. When the server is slow, or the file is big, doing a request might take quite a while. As long as this is happening, the program is waiting, which causes the whole browser to wait. Until the program finishes, the user can not do anything, not even scroll the page. Pages that run on a local network, which is fast and reliable, might get away with doing requests like this. Pages on the big great unreliable Internet, on the other hand, should not.

When the third argument to `open` is `true`, the request is set to be 'asynchronous'. This means that `send` will return right away, while the request happens in the background.

```
request.open("GET", "files/fruit.xml", true);
request.send(null);
show(request.responseText);
```

But wait a moment, and...

```
print(request.responseText);
```

'Waiting a moment' could be implemented with `setTimeout` or something like that, but there is a better way. A request object has a `readyState` property, indicating the state it is in. This will become `4` when the document has been fully loaded, and have a smaller value before that[3]. To react to changes in this status, you can set the `onreadystatechange` property of the object to a function. This function will be called every time the state changes.

```
request.open("GET", "files/fruit.xml", true);
request.send(null);
request.onreadystatechange = function() {
  if (request.readyState == 4)
    show(request.responseText.length);
};
```

When the file retrieved by the request object is an XML document, the request's `responseXML` property will hold a representation of this document. This representation works like the DOM objects discussed in chapter 12, except that it doesn't have HTML-specific functionality, such as `style` or `innerHTML`. `responseXML` gives us a document object, whose `documentElement` property refers to the outer tag of the XML document.

```
var catalog = request.responseXML.documentElement;
show(catalog.childNodes.length);
```

Such XML documents can be used to exchange structured information with the server. Their form — tags contained inside other tags — is often very suitable to store things that would be tricky to represent as simple flat text. The DOM interface is rather clumsy for extracting information though, and XML documents are notoriously wordy: The `fruit.xml` document looks like a lot, but all it says is 'apples are red, oranges are orange, and bananas are yellow'.

As an alternative to XML, JavaScript programmers have come up with something called JSON. This uses the basic notation of JavaScript values to represent 'hierarchical' information in a more minimalist way. A JSON document is a file containing a single JavaScript object or array, which in turn contains any number of other objects, arrays, strings, numbers, booleans, or `null` values. For an example, look at `fruit.json`:

```
request.open("GET", "files/fruit.json", true);
request.send(null);
request.onreadystatechange = function() {
  if (request.readyState == 4)
    print(request.responseText);
};
```

Such a piece of text can be converted to a normal JavaScript value by using the `eval` function. Parentheses should be added around it before calling `eval`, because otherwise JavaScript might interpret an object (enclosed by braces) as a block of code, and produce an error.

```
function evalJSON(json) {
  return eval("(" + json + ")");
}
var fruit = evalJSON(request.responseText);
show(fruit);
```

When running `eval` on a piece of text, you have to keep in mind that this means you let the piece of text run whichever code it wants. Since JavaScript only allows us to make requests to our own domain, you will usually know exactly what kind of text you are getting, and this is not a problem. In other situations, it might be unsafe.

Ex. 14.1

Write a function called `serializeJSON` which, when given a JavaScript value, produces a string with the value's JSON representation. Simple values like numbers and booleans can be simply given to the `String` function to convert them to a string. Objects and arrays can be handled by recursion.

Recognizing arrays can be tricky, since its type is `"object"`. You can use `instanceof Array`, but that only works for arrays that were created in your own window — others will use the

`Array` prototype from other windows, and `instanceof` will return `false`. A cheap trick is to convert the `constructor` property to a string, and see whether that contains `"function Array"`.

When converting a string, you have to take care to escape special characters inside it. If you use double-quotes around the string, the characters to escape are `\"`, `\\`, `\f`, `\b`, `\n`, `\t`, `\r`, and `\v`[4].

```
function serializeJSON(value) {
  function isArray(value) {
    return /^\s*function Array/.test(String(value.constructor));
  }

  function serializeArray(value) {
    return "[" + map(serializeJSON, value).join(", ") + "]";
  }
  function serializeObject(value) {
    var properties = [];
    forEachIn(value, function(name, value) {
      properties.push(serializeString(name) + ": " +
                      serializeJSON(value));
    });
    return "{" + properties.join(", ") + "}";
  }
  function serializeString(value) {
    var special =
      {"\"": "\\\"", "\\": "\\\\", "\f": "\\f", "\b": "\\b",
       "\n": "\\n", "\t": "\\t", "\r": "\\r", "\v": "\\v"};
    var escaped = value.replace(/[\"\\\f\b\n\t\r\v]/g,
                                function(c) {return special[c];});
    return "\"" + escaped + "\"";
  }

  var type = typeof value;
  if (type == "object" && isArray(value))
    return serializeArray(value);
  else if (type == "object")
    return serializeObject(value);
  else if (type == "string")
    return serializeString(value);
  else
    return String(value);
}

print(serializeJSON(fruit));
```

The trick used in `serializeString` is similar to what we saw in the `escapeHTML` function in chapter 10. It uses an object to look up the correct replacements for each of the characters. Some of them, such as `"\\\\"`, look quite weird because of the need to put two backslashes for every backslash in the resulting string.

Also note that the names of properties are quoted as strings. For some of them, this is not necessary, but for property names with spaces and other strange things in them it is, so the code just takes the easy way out and quotes everything.

When making lots of requests, we do, of course, not want to repeat the whole `open`, `send`, `onreadystatechange` ritual every time. A very simple wrapper could look like this:

```
function simpleHttpRequest(url, success, failure) {
  var request = makeHttpObject();
  request.open("GET", url, true);
  request.send(null);
  request.onreadystatechange = function() {
    if (request.readyState == 4) {
      if (request.status == 200)
        success(request.responseText);
      else if (failure)
        failure(request.status, request.statusText);
    }
  };
}

simpleHttpRequest("files/fruit.txt", print);
```

The function retrieves the url it is given, and calls the function it is given as a second argument with the content. When a third argument is given, this is used to indicate failure — a non-`200` status code.

To be able to do more complex requests, the function could be made to accept extra parameters to specify the method (`GET` or `POST`), an optional string to post as data, a way to add extra headers, and so on. When you have so many arguments, you'd probably want to pass them as an arguments-object as seen in chapter 9.

Some websites make use of intensive communication between the programs running on the client and the programs running on the server. For such systems, it can be practical to think of some HTTP requests as calls to functions that run on the server. The client makes request to URLs that identify the functions, giving the arguments as URL parameters or `POST` data. The server then calls the function, and puts the result into JSON or XML document that it sends back. If you write a few convenient support functions, this can make calling server-side functions almost as easy as calling client-side ones... except, of course, that you do not get their results instantly.

1. These are not the only types of requests. There is also `HEAD`, to request just the headers for a document, not its content, `PUT`, to add a document to a server, and `DELETE`, to delete a document. These are not used by browsers, and often not supported by web-servers, but — if you add server-side programs to support them — they can be useful.
2. Not only the 'XML' part of the `XMLHttpRequest` name is misleading — the object can also be used for request over protocols other than HTTP, so `Request` is the only meaningful part we have left.
3. `0` ('uninitialized') is the state of the object before `open` is called on it. Calling `open` moves it to `1` ('open'). Calling `send` makes it proceed to `2` ('sent'). When the server responds, it goes to `3` ('receiving'). Finally, `4` means 'loaded'.
4. We already saw `\n`, which is a newline. `\t` is a tab character, `\r` a 'carriage return', which some systems use before or instead of a newline to indicate the end of a line. `\b` (backspace), `\v` (vertical tab), and `\f` (form feed) are useful when working with old printers, but less so when dealing with Internet browsers.

Appendix 1:

More (obscure) control structures

In chapter 2, a number of control statements were introduced, such as `while`, `for`, and `break`. To keep things simple, I left out some others, which, in my experience, are a lot less useful. This appendix briefly describes these missing control statements.

First, there is `do`. `do` works like `while`, but instead of executing the loop body zero or more times, it executes it one or more times. A `do` loop looks like this:

```
do {
  var answer = prompt("Say 'moo'.", "");
  print("You said '", answer, "'.");
} while (answer != "moo");
```

To emphasise the fact that the condition is only checked *after* the loop has run once, it is written at the end of the loop's body.

Next, there is `continue`. This one is closely related to `break`, and can be used in the same places. While `break` jumps *out* of a loop and causes the program to proceed after the loop, `continue` jumps to the next iteration of the loop.

```
for (var i = 0; i < 10; i++) {
  if (i % 3 != 0)
    continue;
  print(i, " is divisible by three.");
}
```

A similar effect can usually be produced using just `if`, but there are cases where `continue` looks nicer.

When there is a loop sitting inside another loop, a `break` or `continue` statement will affect only the inner loop. Sometimes you want to jump out of the *outer* loop. To be able to refer to a specific loop, loop statements can be labelled. A label is a name (any valid variable name will do), followed by a colon (`:`).

```
outer: for (var sideA = 1; sideA < 10; sideA++) {
  inner: for (var sideB = 1; sideB < 10; sideB++) {
    var hypotenuse = Math.sqrt(sideA * sideA + sideB * sideB);
    if (hypotenuse % 1 == 0) {
      print("A right triangle with straight sides of length ",
            sideA, " and ", sideB, " has a hypotenuse of ",
            hypotenuse, ".");
      break outer;
    }
  }
}
```

Next, there is a construct called `switch` which can be used to choose which code to execute based on some value. This is a very useful thing to do, but the syntax JavaScript

uses for this (which it took from the C programming language) is so clumsy and ugly that I usually prefer to use a chain of `if` statements instead.

```
function weatherAdvice(weather) {
  switch(weather) {
    case "rainy":
      print("Remember to bring an umbrella.");
      break;
    case "sunny":
      print("Dress lightly.");
    case "cloudy":
      print("Go outside.");
      break;
    default:
      print("Unknown weather type: ", weather);
      break;
  }
}

weatherAdvice("sunny");
```

Inside the block opened by `switch`, you can write a number of `case` labels. The program will jump to the label that corresponds to the value that `switch` was given, or to `default` if no matching value is found. Then it start executing statements there, and *continues* past other labels, until it reaches a `break` statement. In some cases, such as the `"sunny"` case in the example, this can be used to share some code between cases (it recommends going outside for both sunny and cloudy weather). Most of the time, this just adds a lot of ugly `break` statements, or causes problems when you forget to add one.

Like loops, `switch` statements can be given a label.

Finally, there is a keyword named `with`. I've never actually *used* this in a real program, but I have seen other people use it, so it is useful to know what it is. Code using `with` looks like this:

```
var scope = "outside";
var object = {name: "Ignatius", scope: "inside"};
with(object) {
  print("Name == ", name, ", scope == ", scope);
  name = "Raoul";
  var newVariable = 49;
}
show(object.name);
show(newVariable);
```

Inside the block, the properties of the object given to `with` act as variables. Newly introduced variables are *not* added as properties to this object though. I assume the idea behind this construct was that it could be useful in methods that make lots of use of the properties of their object. You could start such a method with `with(this) {...}`, and not have to write `this` all the time after that.

Appendix 2:

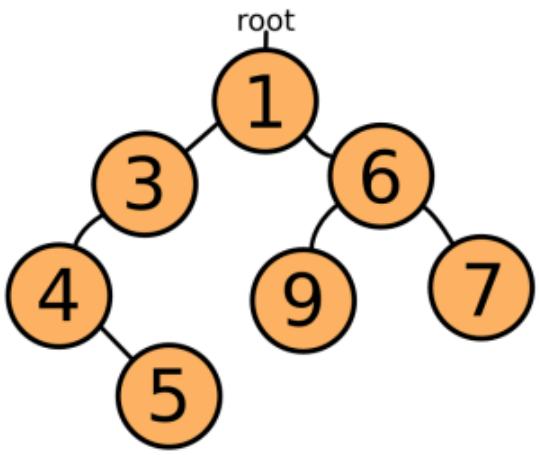
Binary Heaps

In chapter 7, the binary heap was introduced as a method to store a collection of objects in such a way that the smallest element can be quickly found. As promised, this appendix will explain the details behind this data structure.

Consider again the problem we needed to solve. The A* algorithm created large amounts of small objects, and had to keep these in an 'open list'. It was also constantly removing the smallest element from this list. The simplest approach would be to just keep all the objects in an array, and search for the smallest one when we need it. But, unless we have a *lot* of time, this will not do. Finding the smallest element in an unsorted array requires going over the whole array, and checking each element.

The next solution would be, of course, to sort our array. JavaScript arrays have a wonderful `sort` method, which can be used to do the heavy work. Unfortunately, re-sorting a whole array every time an element is removed is more work than searching for a minimum value in an unsorted array. Some tricks can be used, such as, instead of re-sorting the whole array, just making sure new values are inserted in the right place so that the array, which was sorted before, stays sorted. This is coming closer to the approach a binary heap uses already, but inserting a value in the middle of an array requires moving all the elements after it one place up, which is still just too slow.

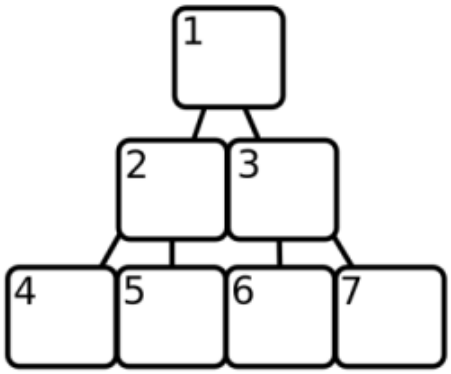
Another approach is to not use an array at all, but to store the values in a set of interconnected objects. A simple form of this is to have every object hold one value and two (or less) links to other objects. There is one root object, holding the smallest value, which is used to access all the other objects. Links always point to objects holding greater values, so the whole structure looks something like this:



Such structures are usually called trees, because of the way they branch. Now, when you need the smallest element, you just take off the top element and rearrange the tree so that one of the top element's children — the one with the lowest value — becomes the new top. When inserting new elements, you 'descend' the tree until you find an element less than the new element, and insert it there. This takes a lot less searching than a sorted array does, but it has the disadvantage of creating a lot of objects, which also slows things down.

A binary heap, then, does make use of a sorted array, but it is only partially sorted, much

like the tree above. Instead of objects, the positions in the array are used to form a tree, as this picture tries to show:



Array element `1` is the root of the tree, array element `2` and `3` are its children, and in general array element `x` has children `x * 2` and `x * 2 + 1`. You can see why this structure is called a 'heap'. Note that this array starts at `1`, while JavaScript arrays start at `0`. The heap will always keep the smallest element in position `1`, and make sure that for every element in the array at position `x`, the element at `x / 2` (round down) is smaller.

Finding the smallest element is now a matter of taking the element at position `1`. But when this element is removed, the heap must make sure that there are no holes left in the array. To do this, it takes the last element in the array and moves it to the start, and then compares it to its child elements at position `2` and `3`. It is likely to be greater, so it is exchanged with one of them, and the process of comparing it with its children is repeated for the new position, and so on, until it comes to a position where its children are greater, or a position where it has no children.

```
[2, 3, 5, 4, 8, 7, 6]
Take out 2, move 6 to the front.
[6, 3, 5, 4, 8, 7]
6 is greater than its first child 3, so swap them.
[3, 6, 5, 4, 8, 7]
Now 6 has children 4 and 8 (position 4 and 5). It is greater than
4, so we swap again.
[3, 4, 5, 6, 8, 7]
6 is in position 4, and has no more children. The heap is in order
again.
```

Similarly, when an element has to be added to the heap, it is put at the end of the array and 'sunk' down by repeatedly exchanging it with its parent, until we find a parent that is less than the new node.

```
[3, 4, 5, 6, 8, 7]
Element 2 gets added again, it starts at the back.
[3, 4, 5, 6, 8, 7, 2]
2 is in position 7, its parent is at 3, which is a 5. 5 is greater
than 2, so we swap.
[3, 4, 2, 6, 8, 7, 5]
The parent of position 3 is position 1. Again, we swap.
[2, 4, 3, 6, 8, 7, 5]
The element can not sink below position 1, so we are done.
```

Note how adding or inserting an element does not require it to be compared with every element in the array. In fact, because the jumps between parents and children get bigger as the array gets bigger, this advantage is especially large when we have a lot of elements[1].

Here is the full code of a binary heap implementation. Two things to note are that, instead of directly comparing the elements put into the heap, a function (`scoreFunction`) is first applied to them, so that it becomes possible to store objects that can not be directly compared.

Also, because JavaScript arrays start at `0`, and the parent/child calculations use a system that starts at `1`, there are a few strange calculations to compensate.

```
function BinaryHeap(scoreFunction){
  this.content = [];
  this.scoreFunction = scoreFunction;
}

BinaryHeap.prototype = {
  push: function(element) {
    // Add the new element to the end of the array.
    this.content.push(element);
    // Allow it to sink down.
    this.sinkDown(this.content.length - 1);
  },

  pop: function() {
    // Store the first element so we can return it later.
    var result = this.content[0];
    // Get the element at the end of the array.
    var end = this.content.pop();
    // If there are any elements left, put the end element at the
    // start, and let it bubble up.
    if (this.content.length > 0) {
      this.content[0] = end;
      this.bubbleUp(0);
    }
    return result;
  },

  remove: function(node) {
    var len = this.content.length;
    // To remove a value, we must search through the array to find
    // it.
    for (var i = 0; i < len; i++) {
      if (this.content[i] == node) {
        // When it is found, the process seen in 'pop' is repeated
        // to fill up the hole.
        var end = this.content.pop();
        if (i != len - 1) {
          this.content[i] = end;
          this.bubbleUp(i);
        }
        return;
      }
    }
    throw new Error("Node not found.");
  },

  size: function() {
    return this.content.length;
  },

  sinkDown: function(n) {
    // Fetch the element that has to be sunk.
    var element = this.content[n];
    // When at 0, an element can not sink any further.
    while (n > 0) {
      // Compute the parent element's index, and fetch it.
      var parentN = Math.floor((n + 1) / 2) - 1,
          parent = this.content[parentN];
      // Swap the elements if the parent is greater.
      if (this.scoreFunction(element) < this.scoreFunction(parent)) {
        this.content[parentN] = element;
        this.content[n] = parent;
        // Update 'n' to continue at the new position.
        n = parentN;
      }
      // Found a parent that is less, no need to sink any further.
```

```
      else {
        break;
      }
    }
  },

  bubbleUp: function(n) {
    // Look up the target element and its score.
    var length = this.content.length,
        element = this.content[n],
        elemScore = this.scoreFunction(element);

    while(true) {
      // Compute the indices of the child elements.
      var child2N = (n + 1) * 2, child1N = child2N - 1;
      // This is used to store the new position of the element,
      // if any.
      var swap = null;
      // If the first child exists (is inside the array)...
      if (child1N < length) {
        // Look it up and compute its score.
        var child1 = this.content[child1N],
            child1Score = this.scoreFunction(child1);
        // If the score is less than our element's, we need to swap.
        if (child1Score < elemScore)
          swap = child1N;
      }
      // Do the same checks for the other child.
      if (child2N < length) {
        var child2 = this.content[child2N],
            child2Score = this.scoreFunction(child2);
        if (child2Score < (swap == null ? elemScore : child1Score))
          swap = child2N;
      }

      // If the element needs to be moved, swap it, and continue.
      if (swap != null) {
        this.content[n] = this.content[swap];
        this.content[swap] = element;
        n = swap;
      }
      // Otherwise, we are done.
      else {
        break;
      }
    }
  }
};
```

And a simple test...

```
var heap = new BinaryHeap(function(x){return x;});
forEach([10, 3, 4, 8, 2, 9, 7, 1, 2, 6, 5],
        method(heap, "push"));

heap.remove(2);
while (heap.size() > 0)
  print(heap.pop());
```

1. The amount of comparisons and swaps that are needed — in the worst case — can be approached by taking the logarithm (base 2) of the amount of elements in the heap.