Adventures in JAVAScript

COMP 102A, Lecture 7

slides=extracts of http://eloquentjavascript.net/

Chapter 1:

Introduction

Programming

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 00000001 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Programming

```
 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'
```

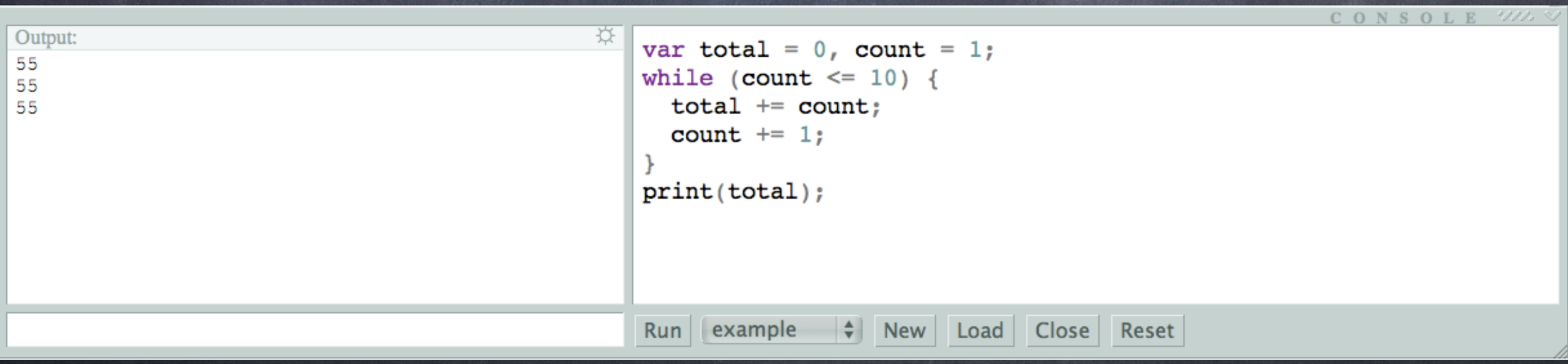
Programming

Here is the same program in JavaScript:

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

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);
```



Chapter 2:

Basic JavaScript: values, variables, and control flow

Basics of JAVAScript: numbers

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.

Arithmetic

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.

Arithmetic

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.

Basics of JAVAScript: strings

The next data type is the string. Its use is not as evident from its name as with numbers, but it also fulfils 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:

```
"I 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.

Strings

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[3].

```
"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

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.

Unary Operators

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)
```

Booleans

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
```

Comparing Strings

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.

Boolean Operators

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`.

Statements

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.

Variables

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 a 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.

Functions

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 amount.

While loop

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;
}
```

For loop

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.

Short hands...

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--`.

Adventures in JAVAScript

COMP 102B, Lecture 8

slides=extracts of http://eloquentjavascript.net/

If

Loops are said to effect 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.

If

If we wanted to print all of the numbers between 0 and 20, but put parentheses around the ones 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 + ")");
}
```

Break

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

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

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

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

Variable naming

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.

Reserved names

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.

What is it ?

In the second solution to the previous exercise there is a statement `var value;`. This creates a variable named `value`, 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);
```

What is it, really ?

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`.

Chapter 3:

Functions

Functions

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

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

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

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

show(add(2, 2));
```

`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.

Functions

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));
```

Functions

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.

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");
```

Functions

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");
```

Functions and local variables

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();
```

Functions and local variables

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.

Recursive Functions

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.

Recursive Functions

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.

Nameless Functions

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

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

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

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.

Adventures in JAVAScript

COMP 102B, Lecture 9

slides=extracts of http://eloquentjavascript.net/

Chapter 4:

Data structures: Objects and Arrays

Properties & Objects

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.

Properties & Objects

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);
```

Properties & Objects

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);
```

Properties & Objects

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-existant 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 & Objects

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"]);
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]);
```

Properties & Objects

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);
```

from Objects to Arrays

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

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

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

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

```
var mailArchive = {0: "Dear nephew, ... (mail number 1)",
                   1: "(mail number 2)",
                   2: "(mail number 3)"};

for (var current = 0; current in mailArchive; current++)
  print("Processing e-mail #", current, ": ", mailArchive[current]);
```

Arrays

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

Arrays

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));
```

Math "Object" functions

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.

Adventures in JAVAScript

COMP 102B, Lecture 10

slides=extracts of http://eloquentjavascript.net/

Chapter 6:

Functional Programming

ForEach

One ugly detail that, 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);
```

ForEach

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.

Modifying a function

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?

Apply-ing!

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.

Reduce-ing

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.

Map-ping!

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.