Adventures in JAVAScript

COMP 102B, Lecture 13

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.

Chapter 6:

Functional Programming

ForEach

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);
```

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.

ForEach

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.

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.

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.

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.