Advanced JavaScript for Web Sites and Web Applications

Arrays, Objects and Prototypes

Arrays

- Arrays hold sequences or sets of values
  - the *values* can be strings, numbers, objects, functions, other arrays, etc.
- They are used throughout the language to store and manipulate data
- They have a set of properties and methods that can help us do this.

Defining arrays

- An array with some values:

```
var myData = ["a", 500, true, 12, "string"];
```

- An empty array:

```
var myData = [];
// OR
var myData = new Array();
```

Associative Arrays

- Although it is considered bad practice, you will sometimes see non-numerical, associative arrays:

```
var myData = [];

myData["size"] = "small";
myData["colour"] = "red";
```

- However, to store this type of data, Objects are preferred.

Accessing Array values

```
var myArray = ["a", "b", "c"];

// Access elements by their index (position)
// Indexes start at 0
myArray[0]; // returns a
myArray[2]; // returns c

// Accessing non-existing indexes
myArray[3]; // returns undefined
```

Array properties

- Every array we create has several built-in methods and properties:
- We use *dot notation* to access them:

```
myArray.name_of_property_or_method;
```

Arrays - The `length` property

```
var myArray = ["a", "b", "c"];

// Use "length" to get number of items in the array
var totalItems = myArray.length;

// Useful for iterating arrays
for(var i = 0; i < totalItems; i++) {
    console.log(myArray[i]);
}
```

Arrays - the `push` method

- Add new items to the end of an array with the `push()` method:

```
array.push(item_to_add);
```

- Example:

```
var myArray = ["a", "b", "c"];
myArray.push("d");

console.log(myArray); // ["a", "b", "c", "d"]
```

Arrays - the `push` method

- To add several items to the array at once, pass a comma separated list of values to `push()`:

```javascript
var myArray= ["a", "b", "c"];
myArray.push("d", "e", "f");

console.log(myArray);
// result:  ["a", "b", "c", "d", "e", "f"]
```

Arrays - the `concat` method

- Combine 2 arrays with the `concat` method:

```
array.concat(array_to_append);
```

- `concat()` returns a new array consisting of the elements of the array upon which it was called, followed by the elements from **array_to_append**

Arrays - the `concat` method

```javascript
// The arrays to join
var letters = ["a", "b", "c"],
    numbers = [1, 2, 3];

// Call concat method of "letters" array
var combined = letters.concat(numbers);

console.log(combined);
// Result: ["a", "b", "c", 1, 2, 3]
// "letters" & "numbers" remain unchanged
```

Arrays - the `slice` method

- Extract part of an array with the `slice` method:

```
array.slice(start, end);
```

- Where:
  - **start** is the *index* of the element you want to start extracting from
  - **end** is the *index* at which the extraction should stop. Note, the end element will **not** be included in the extraction

Arrays - the `slice` method

- `slice()` *returns* a new array, the original array is left unchanged

```
var myArray = ["a", "b", "c", "d"],
    newArray = myArray.slice(1, 3);

console.log(myArray); // ["a", "b", "c", "d"]
// "b" has index: 1, "d" has index: 3
// So the slice contains "b" and "c"
console.log(newArray); // ["b", "c"]
```

Copying Arrays

- When we make a copy of an array variable, JavaScript assigns by reference
- This means that the new variable is a reference to the same data as the original variable

Copying Arrays - example

```
var myArray = ['a', 'b', 'c'];
// Make copy of array variable
var cpy = myArray;
// add new value to cpy
cpy.push('d');

// Both arrays are altered!
console.log(myArray); // ['a','b','c','d']
console.log(cpy); // ['a','b','c','d']
```

Arrays - copying with `slice`

- While this behaviour can be useful, we often want our *copy* to be separate from the original data
- We can do this with the `slice()` method.
- When called with no start or end parameters, `slice()` returns the entire array

Arrays - copying with `slice`

```
var myArray = ['a', 'b', 'c'],
    cpy = myArray.slice();

// add new value to cpy
cpy.push('d');

// Only cpy is altered!
console.log(myArray); // ['a','b','c']
console.log(cpy); // ['a','b','c','d']
```

Arrays - the `join` method

- Convert an array to a string with the `join` method:

```
array.join(separator);
```

- Where:
  - **separator** is the character(s) to separate each element of the array with

Arrays - the `join` method

- `join()` *returns* a string:

```
var myArray = ["a", "b", "c"],
    x = myArray.join(""),
    y = myArray.join(","),
    z = myArray.join(" and ");

console.log(x); // "abc"
console.log(y); // "a,b,c"
console.log(z); // "a and b and c"
```

Arrays - building complex strings

- Building long strings in JavaScript is awkward
- However, we can simplify the process with an *array* and it’s `join()` method

Arrays - building complex strings

- Step 1: We create an array of the things we want to join
    - can be strings or variables containing strings
- Step 2: We use `join()` to *glue* them together with an appropriate character(s)

Arrays - building complex strings

```
var myArray = [];

// Each element is a section of the final string
myArray.push("This is a very long sentence");
myArray.push("which is awkward to construct in");
myArray.push("JavaScript, plus some browsers");
myArray.push("don't like assigning long values");
myArray.push("to variables");

// Join the sections with a space character
var myString = myArray.join(" ");
```

Arrays - building HTML strings

```javascript
var text = "A paragraph of text",
    text2 = "A second paragraph of text",
    htmlData = [];
htmlData.push('<div class="my-class">');
htmlData.push('<p>');
htmlData.push(text);
htmlData.push('</p>');
htmlData.push('<p>');
htmlData.push(text2);
htmlData.push('</p>');
htmlData.push('</div>');
// Join array elements with a new line character
var htmlOutput= htmlData.join("\n");
```

Objects

- Objects are collections of *properties*, stored as **name : value** pairs.
- The *values* can be almost anything
  - Strings, numbers, arrays, functions, other objects
- If the value stored in an object property is a function, it is called a `method`.

Creating objects

- To create a new, empty object:

```
var myObject = {};
```

- Or, create an object with properties/methods:

```
var myObject= {
    colour: "red",
    state: true,
    action: function () {
        console.log('Hello');
    }
};
```

Object properties/methods

- Use *dot notation* to access the *properties* and *methods* of an object:

```
var myObject = {
    // see previous slide
};

// get the value of the colour property
var colour = myObject.colour;
// run the function defined in the object
myObject.action();
```

Modifying objects

- Once created, you can add new properties and methods to the object, or overwrite existing ones
- Again, we use *dot notation* to reference object properties and methods
- Effectively, object properties are just like regular variables

Modifying objects - example

```
var myObject= {
    // see earlier slide
};

// Define new properties/methods
myObject.newProperty = 'I am new';
myObject.newMethod = function () {
    console.log('Hello from new method!');
};

// Modify existing property
myObject.colour = 'blue';
```

Object properties - alternative syntax

- We can also use *bracket notation* to access properties:

```
var myObject = {
        colour: "red",
        action: function () {
            console.log('Hello');
        }
    };
console.log(myObject["colour"]); // red
myObject["action"](); // Logs: Hello
```

Bracket notation

- Bracket notation is useful when the property name we want to access is stored in a variable:

```
var myObject = {
    colour: "red",
    action: function () {
        console.log('Hello');
    }
};
var propertyName = 'colour';
var objectColour = myObject[propertyName];
```

Objects: Iteration

- To iterate an object’s properties, use a `for...in` loop

```
for (propertyName in object) {
    // this runs once for each property
}
```

- On each iteration of the loop, **propertyName** will hold a different property *name*
- We can then use *bracket notation* to access the value of that property

for...in example

```
var prop, message,
    myObject = {
        a: 1,
        b: 2
    };

for (prop in myObject) {
    // "prop" is "name" of property
    message = "Property name: " + prop + ", ";
    // Use myObject[prop] to get it's value
    message += "Value: " + myObject[prop];
    console.log(message);
}
```

Exercises

- Download the *exercises* document from Moodle and do *exercise 1* and *exercise 2*

Objects: as function arguments

- Instead of passing multiple arguments to a function, you can pass an object with multiple properties.
- Each *property* of the object will represent a function argument
- This approach can lead to more manageable code

Objects: as function arguments - example

```javascript
function greeting(data) {
    var message = data.text + " " + data.name;
    console.log(message);
}
// The argument object
var myData = {
    text: 'Hello',
    name: 'John'
};
// Pass object to function
greeting(myData);
```

Objects: as function arguments - hands-on

- Task: In the following code, modify `appendText` so that it accepts a single object as its argument
- The object should have properties to match the current arguments
- You will also need to modify the function call

```
function appendText(element, text) {
    element.textContent= text;
}
// Calling the function
appendText(DOMelement, "text to set");
```

Patterns

- When writing large and complex applications, you can make things a lot easier on yourself by organising the code using patterns.
- This means moving away from writing linear, procedural code, to a more modular, object oriented style of coding.
- There are lots of different code organisation patterns out there, each one with its own specific application and use case.

Module pattern

- For the rest of this course, we'll have a look at the *revealing module* pattern, which is an extension of the *module* pattern.
- When using the *module* pattern, we place all of the code needed to perform a specific task or set of tasks in an *object literal*
  - The variables and functions used by our script become *properties* of the object

Module pattern example

```
var myApp = {
    text: "World",
    config: {
        language: "en",
        debug: false
    },
    doStuff: function () {
        return "Hello " + this.text;
    }
};
// Using the module properties/methods
myApp.text; // World
myApp.doStuff(); // Hello World!
myApp.config.language; // en
```

Exercises

- Now do *exercise 3*
- Now do *exercise 4*
- Now do *exercise 5*

Prototypes

- Almost everything in JavaScript is an object.
  - arrays, functions, strings, numbers etc.
- All objects have a *prototype*
  - A *prototype* is another object with properties and methods which will be available to all objects which have it as a prototype.

Prototype inheritance

- An object *inherits* the properties of it's prototype, and can add some of it's own
- Prototype objects can have prototypes too.
- Ultimately, all objects/prototypes are descended from Object.prototype

```
Object.prototype
    --> Array.prototype
        --> var myArray = new Array();
```

Adding to Prototypes

- To add to the prototype of Arrays:

```javascript
var myArray= [1, 3];

// New method for arrays
Array.prototype.newMethod = function () {
    console.log('Hello');
}
myArray.newMethod(); // call new method
```

- The new method will be available to **all** arrays defined within the scope of your script!

Adding to Prototypes

- Avoid adding properties and methods to native objects and types!
  - You risk breaking existing functionality and code.
- You can however define your own object types and use them as constructors to build other objects.
- These built objects will inherit the properties and methods of your constructors.
  - And you can reasonably safely change the prototype for these.

Custom Prototype constructors

- Using the *new* operator, you can create a new *instance* of your own constructor object:

```
function Car(text) {
    this.name = text;
}
var myCar = new Car("Mustang");
var myOtherCar = new Car("Cadillac");
```

- *this* in the function definition is a reference to the object that will be built
  - More on this later in the course

Adding to the Prototype

- And then add methods/properties to its prototype:

```
// Add a new method for cars
Car.prototype.getName = function () {
    return this.name;
};
// Use the new method
var carName = myCar.getName();
console.log(carName);
var otherCarName = myOtherCar.getName();
console.log(otherCarName);
```

Prototype constructors

- You can check the *constructor* of custom and built-in objects:

```
var myArray = [];
console.log(myArray.constructor);// Array()

var myObject = {};
console.log(myObject.constructor);// Object()

var myString = "";
console.log(myString.constructor);// String()
```