UNIT 1: HTML, CSS & Client Side Scripting

The 'global' object

When the JavaScript interpreter starts up, there is always a single global object instance created. All other objects are properties of this object. Also, all variables and functions are properties of this global object.

For client-side JavaScript, this object is the instance window of the constructor Window.

The Window object for client-side JavaScript: window

When an HTML document is loaded into a browser, a single Window object instance, named window, is created. All other objects are properties of this window object. Since everything is a property of the window object, there is a relaxation of the dot notation when referring to properties. Thus each reference to a variable, function or object is not required to start with "window." although this would be a more "accurate" notation to use, it is far less convenient.

Thus instead of writing:

```
document.write( "Hello" );
```

It can be written:

```
window.document.write( "Hello" );
```

One of the properties of the window object is the status property. This property is a String value that is displayed in the browser window's status bar. The status bar can be changed with, or without, an explicit reference to the window object. Both these lines have the same effect:

```
window.status = "this is a test";
status = "this is a test";
```

The Document object: document

When an HTML document is loaded into a frame of a browser window, a Document object instance named document is created for that frame. This document object, like most objects, has a collection of properties and methods. Perhaps the most frequently used method of the document object is the write() method:

document.write("sorry, that item is out of stock<p>");

One useful property of the document object is the forms property. This is actually an array of Form objects with an element for each form defined in the document. So it could refer to the first form defined in the document as follows: document.forms[0] .

The 'call' object

When a function (or method) is executed, a temporary object is created that exists for as long as the function is executing. This object is called the call object, and the arguments and local variables and functions are properties of this object.

It is through use of this call object that functions/methods are able to use local argument and variable/ function names that are the same as global variables/functions, without confusion. The call object is JavaScript's implementation of the concepts of variable/function scoping i.e. determining which piece of memory is referred to by the name of a variable or function, when there are global and local properties with the same name.

String objects

Strings are objects. A frequently used property of String is length. String includes methods to return a new String containing the same text but in upper or

lower case. So we could create an upper case version of the String "hello" as follows:

var name = "hello"; alert(name.toUpperCase());

It should be noted that none methods belonging to string objects never change the string's value, but they may return a new String object,.

Array objects

Since Arrays are rather an important topic in the own right, Array objects are given their own section at the end of this unit although you may wish to skip ahead to read that section now to help your understanding of object with this more familiar example.

Function objects

Functions are of the type Function, and can be treated as data variables or properties, or they can be treated as sub-programmes and executed using the () operator.

Math objects

The Math object provides a number of useful methods and properties for mathematical processing. For example, Math provides the following property:

Math.PI

that is useful for many geometric calculations.

An example of some other methods provided by the Math object include:

- Math.abs();
- Math.round();
- Math.max(n1, n2);

These may be used in the following way:

document.write("<p>PI is " + Math.PI);

document.write("<p>The signless magnitudes of the numbers -17 and 7 are " + Math.abs(-17) + " and " + Math.abs(7));

document.write("<p>The interger part of 4.25 is " + Math.round(4.25));

document.write("<p>The larger of 17 and 19 is " + Math.max(17, 19));

setTimeout and setInterval

We may decide to execute a function not right now, but at a certain time later. That's called "scheduling a call".

There are two methods for it:

- **setTimeout** allows us to run a function once after the interval of time.
- **setInterval** allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

These methods are not a part of JavaScript specification. But most environments have the internal scheduler and provide these methods. In particular, they are supported in all browsers and Node.js.

setTimeout

The syntax:

let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)

Parameters:

func|code

Function or a string of code to execute. Usually, that's a function. For historical reasons, a string of code can be passed, but that's not recommended.

delay

The delay before run, in milliseconds (1000 ms = 1 second), by default 0.

arg1, arg2…

Arguments for the function (not supported in IE9-)

For instance, this code calls sayHi() after one second:

```
function sayHi() {
  alert('Hello');
}


setTimeout(sayHi, 1000);
```

With arguments:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

If the first argument is a string, then JavaScript creates a function from it.

So, this will also work:

```
setTimeout("alert('Hello')", 1000);
```

But using strings is not recommended, use arrow functions instead of them, like this:

```
setTimeout(() => alert('Hello'), 1000);
```

Pass a function, but don't run it

Novice developers sometimes make a mistake by adding brackets () after the function:

```
// wrong!
setTimeout(sayHi(), 1000);
```

That doesn't work, because setTimeout expects a reference to a function. And here sayHi() runs the function, and the *result of its execution* is passed to setTimeout. In our case the result of sayHi() is undefined (the function returns nothing), so nothing is scheduled.

Canceling with clearTimeout

A call to setTimeout returns a "timer identifier" timerId that we can use to cancel the execution.

The syntax to cancel:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier

clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

As we can see from alert output, in a browser the timer identifier is a number. In other environments, this can be something else. For instance, Node.js returns a timer object with additional methods.

Again, there is no universal specification for these methods, so that's fine.

For browsers, timers are described in the timers section of HTML5 standard.

setInterval

The setInterval method has the same syntax as setTimeout:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

All arguments have the same meaning. But unlike setTimeout it runs the function not only once, but regularly after the given interval of time.

To stop further calls, we should call clearInterval(timerId).

The following example will show the message every 2 seconds. After 5 seconds, the output is stopped:

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
```

```
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Time goes on while alert is shown

In most browsers, including Chrome and Firefox the internal timer continues "ticking" while showing alert/confirm/prompt.

So if you run the code above and don't dismiss the alert window for some time, then in the next alert will be shown immediately as you do it. The actual interval between alerts will be shorter than 2 seconds.

Nested setTimeout

There are two ways of running something regularly.

One is setInterval. The other one is a nested setTimeout, like this:

```
/** instead of:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

The setTimeout above schedules the next call right at the end of the current one (*).

The nested setTimeout is a more flexible method than setInterval. This way the next call may be scheduled differently, depending on the results of the current one.

For instance, we need to write a service that sends a request to the server every 5 seconds asking for data, but in case the server is overloaded, it should increase the interval to 10, 20, 40 seconds…

Here's the pseudocode:

```
let delay = 5000;
```

```
let timerId = setTimeout(function request() {
  ...send request...

  if (request failed due to server overload) {
    // increase the interval to the next run
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

And if the functions that we're scheduling are CPU-hungry, then we can measure the time taken by the execution and plan the next call sooner or later.

Nested setTimeout allows to set the delay between the executions more precisely than setInterval.