# WORKING WITH TIME AND INTRODUCING OBJECTS

To understand a little more about Javascript and the DOM we are going to create a number of features for the web pages in the ZIP. These will include a digital and analogue clock and an amendable staff list.

## SETTING UP THE EXTERNAL FILE

Open the file *index.html*. Create a js file at *js/clock.js* with the following skeleton code:

```
(function(){
console.info("Hello JS");
})();
```

We create the Javascript in an external file and encapsulate it within an IIFE to create a private scope and prevent variable conflicts with other scripts.

Add the script to the index.html by adding the following inside the `<head>` tag.

```
<script src="js/clock.js" defer></script>
```

Note the use of the defer attribute that executes the script when the document has been parsed by the browser.

Save and test the file. The console message should appear in your browser's developer console.

## ADDING CONTENT TO THE DOM

As well editing existing elements in the DOM, we can use Javascript to add completely new elements.

To create a new DOM element use `document.createElement().` This takes a parameter that is the HTML element you wish to add. With this method it is useful to store the result for easy reference. As such we'll declare a constant with the `const` keyword as follows:

```
const myNode = document.createElement('div');
```

If you add this to your JS file, and save and test it, you won't see any change. This is because the element has been created but not yet added to the DOM.
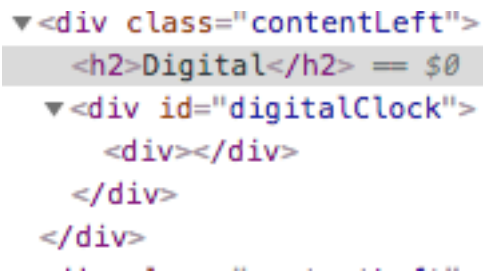
To add the element to the DOM we need to identify where we'd like to place it. We'd like the new element adding inside of the `<div id="digitalClock">` and after the `<h1>` inside that.

As such we could use `document.getElementById()` to target the element. Then we could use `appendChild()` which is a method of the document object that is used to add a new element as a child of the targeted one.

Add the following after the line creating the new `<div>`.

```
digitalClock.appendChild(myNode);
```

This will have added the new element to the DOM.  Use Chrome Developer tools to inspect the element and it will reveal an empty `<div>`.

```
▼<div class="contentLeft">
    <h2>Digital</h2> == $0
  ▼<div id="digitalClock">
      <div></div>
    </div>
  </div>
```

You could now use `innerHTML` to add some content to it eg:

```
myNode.innerHTML = "Welcome";
```

## DISPLAYING DATES WITH JAVASCRIPT

The document is not the only object available to Javascript developers, there are many more.  A good one to experiment with is the Date object.  We'll be able to use this to display the current date and time.

To create a new Date object we instantiate it.  This is done with the `new` keyword eg:

```
const myDate = new Date();
```

To view what this has done add the following to your code:

```
console.dir(myDate);
```

When viewing the output in the console you will see:

```
▼ Mon Mar 05 2018 17:32:11 GMT+0000 (GMT) ℹ
   ▼ __proto__:
      ▶ constructor: f Date()
      ▶ getDate: f getDate()
      ▶ getDay: f getDay()
      ▶ getFullYear: f getFullYear()
      ▶ getHours: f getHours()
      ▶ getMilliseconds: f getMilliseconds()
      ▶ getMinutes: f getMinutes()
      ▶ getMonth: f getMonth()
      ▶ getSeconds: f getSeconds()
      ▶ getTime: f getTime()
      ▶ getTimezoneOffset: f getTimezoneOffset()
      ▶ getUTCDate: f getUTCDate()
      ▶ getUTCDay: f getUTCDay()
      ▶ getUTCFullYear: f getUTCFullYear()
      ▶ getUTCHours: f getUTCHours()
      ▶ getUTCMilliseconds: f getUTCMilliseconds()
      ▶ getUTCMinutes: f getUTCMinutes()
      ▶ getUTCMonth: f getUTCMonth()
      ▶ getUTCSeconds: f getUTCSeconds()
      ▶ getYear: f getYear()
```

As the above demonstrates there are a large number of methods related to the Date object.  The `toDateString()` methods returns just the date part of the object.

Add the current date to the DOM via the node we created above.

```
myNode.innerHTML = myDate.toDateString();
```

Save and test your file.  You should see the date displayed.

## UPDATING VARIABLES WITH SETINTERVAL()

As the Date object provides information about the current time we can use that to add a clock to our page.  To do so we'll need to call our code every second to update the time.

Fortunately, Javascript has a method that can be used to call a function at a set time interval.  As well as the document object, Javascript has a window object.  The window object, like the document object, has a range of methods and properties.  One method of the window object is setInterval().  The setInterval() methods takes two parameters, a function to call and then the time interval between function calls (a value in milliseconds).  The function to be called can either be named or anonymous.

Calling an anonymous function would appear:

```
setInterval(function(){
// do something
}, 1000);
```

Notice that with the anonymous approach the first parameter is an entire function.
Calling a named function would appear as follows:

```
let updateTime = function(){
     console.info('tick tock');
}
setInterval(updateTime, 1000);
```

The Date object method toTimeString() will return the time as a string.  We can use this as a quick way to update the time.  Re-jig your code to place the creation of the Date object inside of the updateTime() function as follows:

```
let updateTime = function(){
     let myDate = new Date();
     myNode.innerHTML = myDate.toTimeString();
}
setInterval(updateTime, 1000);
```

## TIDYING UP THE OUTPUT

The above works but the `toTimeString()` method returns information about the time zone (GMT) that isn't really needed.  There are a couple of ways to fix this.

**Option One: Breaking the Time Down into Hours Minutes and Seconds**

The first option would be to concatenate the time today from three values for the hours, minutes and seconds.  Each component has a Date object method.

```
let myDate = new Date();
let myHours = myDate.getHours();
let myMins = myDate.getMinutes();
let mySeconds = myDate.getSeconds();
let myTimeDisplay = myHours + ":"+ myMins +":"+ mySeconds;
myNode.innerHTML = myTimeDisplay;
```

This will work but there is an issue when values for hours, minutes and secondsd are less than ten. As such we could introduce some conditional logic to check if that is the case and then add a leading zero.

```
if(myHours < 10){
      myHours = "0"+ myHours;
}
if(myMins < 10){
      myMins = "0"+myMins;
}
if(mySeconds < 10){
      mySeconds = "0"+mySeconds;
}
```

This fixes the issue, but we could refactor it, that is reduce the code and try and make it more efficient.  This will give us an excuse to build a function.  Add a function outside the `update` function as follows:

```
let checkForLeadingZeros  = function (timeVal){
      if(timeVal < 10){
            timeVal = "0"+timeVal;
      }
      return timeVal;
}
```

We can then add a call to this function when the three values are created eg:

```
let myDate = new Date();
let myHours = checkForLeadingZeros(myDate.getHours());
let myMins = checkForLeadingZeros(myDate.getMinutes());
let mySeconds = checkForLeadingZeros(myDate.getSeconds());
```

**Option Two: String Object Manipulation**

The second approach just involves manipulating the string value acquired via `toTimeString()`. Javascript has a range of methods and properties related to strings because a string variable is an object. One string methods we could use here is `substr()`. This returns a specified number of characters from a string from a specific staring point. As we want the time in a format such as 10:00:00 we need eight characters and should start at the beginning ie 0.

Remove the previous code and recreate the time as follows:

```
myNode.innerHTML = myDate.toTimeString().substr(0, 8);
```

This technique is known as chaining where one method can be set to follow another.
Bugs to fix. The time isn't displayed until the interval is called - ie there is a one second delay. Therefore make a call to the function before the setInterval is set up eg:

```
upDateTime();
setInterval(updateTime, 1000);
```

## CREATING AN ANALOGUE VERSION OF THE CLOCK

Add the following HTML to the file to add a clock face and hands. Place this where prompted inside the `div#contentLeft` where the comment appears.

```
 <div id="clockFace">
       <div id="secondHand"></div>
       <div id="minHand"></div>
       <div id="hrHand"></div>
       <div id="pin"></div>
    </div>
```

Inline styles can be added directly to HTML elements using the Javascript style property for example:

```
document.getElementById('hrHand').style.backgroundColor = '#0f0';
```

We can use CSS3 transformations to manipulate the HTML above to rotate the clock hands as appropriate. The CSS property `transform` takes a value in degrees.

This will require some maths to convert seconds, minutes and hours into degrees for the rotations.

All three clocks hands are initially lying at 90 degrees to the centre of the clock.

To set seconds we would therefore use:

```
let rotSeconds = (seconds * 6)  - 90;
```

To apply this rotation to the *div#secondHand* we would use:

```
document.getElementById('secondHand').style.transform =
'rotate('+rotSeconds+'deg)';
```

Can you complete the task with hour and minute hands?

## WORKING WITH JAVASCRIPT OBJECTS

Open the file *staffList.html*.  Create a skeleton JS file at *js/staff.js* as follows:

```
(function(){
console.info("Hello JS");
})();
```

Add the script to the *staffList.html* by adding the following before the closing </body> tag.

```
<script src="js/staff.js"></script>
```

The javascript we add for this next example will all be in this *js/staff.js* file.

## JAVASCRIPT OBJECTS

Javascript objects are more complex data types that consist of name / value pairs.  They are created with {…} syntax with the name separated from the value with a colon :.  For example:

```
let myObject = { name : "Martin" };
```

Objects can contain many name:value pairs separated by commas such as:

```
let myObject = { name : "Martin", email: "myemail@company.com"};
```

Objects can be accessed for debugging with `console.dir` eg:

```
let myObject = { name : "Martin", email: "myemail@company.com"};
console.dir(myObject);
```

The 'name' in the name/value pair are not quoted.  The value can be Javascript values such as strings, integers, floats, Booleans but also arrays and other objects eg:

```
let bobDetails = {
      name : 'Bob',
      age : 21,
      qualifications: {
            gsce : true,
            alevels : true,
            degree : false
      }
}
console.dir(bobDetails);
```

Javascript Objects can also have methods – that is a value can be function eg:

```
let bobDetails = {
      name : 'Bob',
      age : 21,
      getInfo : function(){
      return this.name + " " this.age;
      }
}
console.info(bobDetails.getInfo);
```

It is also common to have an array of objects, with each array item been an object eg:

```
let qualifications = [
        {
                subject : "Maths",
                grade : "A"
        },
        {
                subject : "English",
                grade : "B"
        },
        {
                subject : "French",
                grade : "C"
        },
        {
                subject : "Physics",
                grade : "B"
        }
    ];
```

These can be looped around various ways.  One technique is to use the `forEach()` method that extracts each array element eg:

```
qualifications.forEach(function(element) {
        console.dir(element);
    });
```

If each element of the array is an object then its value of the `name:value` pairs can be extracted with dot notation eg:

```
qualifications.forEach(function(element) {
        console.info(element.subject);
        console.info(element.grade);
    });
```

## ADDING DATA TO OUR PAGE VIA A JAVASCRIPT OBJECT

In this example we'll use data from a Javascript Object to populate the HTML table in the *staffList.html* page.

The table is currently structured as:

```
<table>
     <thead>
          <tr>
          <th>Name</th>
          <th>Email</th>
          </tr>
     </thead>
     <tbody id="staffTable">
     </tbody>
</table>
```

Note the use of <thead> and <tbody> valid HTML tags to differentiate between the head of the table and its body content.  This will help when adding data to the table.
In *js/staff.js* create an array of objects as follows:

```
let staff =
     [
         {
             name : "Bob Smith",
             email : "b.smith@busy.com"
         },
         {
             name : "Jane Johnson",
             email : "j.johnson@busy.com"
         },
         {
             name : "Ethan Hawkson",
             email : "e.hawkson@busy.com"
         },
         {
             name : "Julie Grant",
             email : "j.grant@busy.com"
         }
     ]
```

To add this to the DOM we'll use a forEach().

```
staff.forEach(function(element) {
      let newStaffRow = document.createElement("tr");
      let newStaffName = document.createElement("td");
      newStaffName.innerHTML = element.name;
      let newStaffEmail = document.createElement("td");
      newStaffEmail.innerHTML = element.email;
      newStaffRow.appendChild(newStaffName);
      newStaffRow.appendChild(newStaffEmail);
      document.getElementById('staffTable').appendChild(newStaffRow);
});
```

This loop creates a new `<tr>` node for each member of staff.  We then create a new `<td>` to contain the staff name and one for the staff email.  These two `<td>` nodes are appended to the `<tr>`.  Then the `<tr>` node (with its two child `<td>` nodes) is appended to the DOM `staffTable` element that represents the `<tbody>` of the table.

## USING LOCALSTORAGE

Instead of loading the data from a Javascript object we could store the data in the browser using a feature known as localStorage.  This similar to cookies that are used to store data in the browser.  Like cookies, localStorage data can persist even when the browser is closed.
Data can be stored locally in the browser with `localStorage.setItem(<item name>, <item value>)` and values retrieved with `localStorage.getItem(<item name>)`.

The name/value pairs in localStorage are always stored as strings.
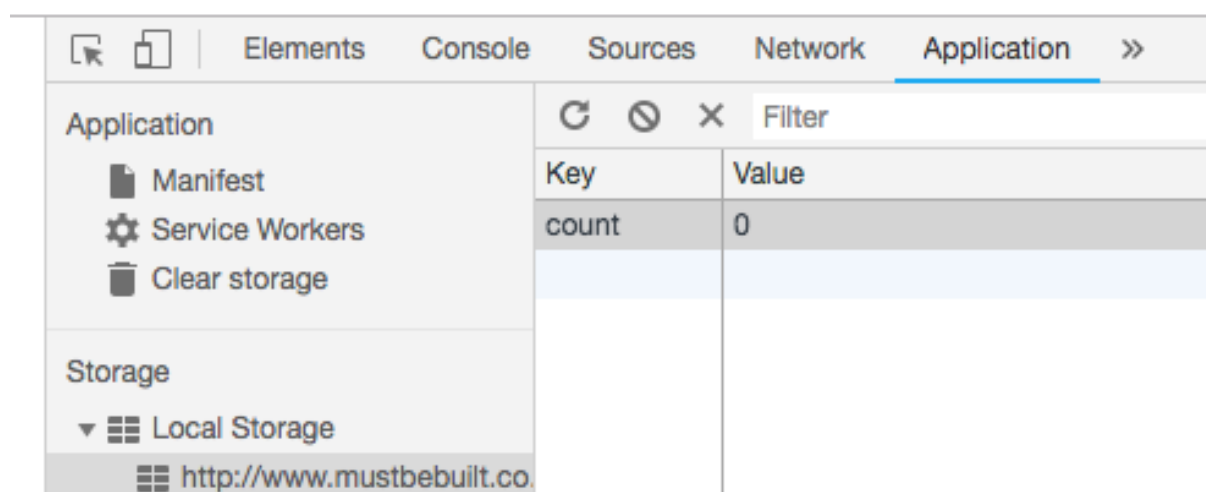
Try the following:

```
localStorage.setItem("count", 1);
```

To view the data use the Chrome Dev Tools Application tab and locate Local storage:

To increment this value, we could amend the code as follows:

```
if(localStorage.getItem("count") === null){
      localStorage.setItem("count", 1);
}else{
      localStorage.setItem("count", parseInt(localStorage.getItem("count"))+1);
}
```

Note as name/value pairs in localStorage are always stored as strings we use `parseInt()` to convert the value retrieved via `getItem()` as an integer.

## ADDING A NEW MEMBER OF STAFF

Comment or remove the hardcoded staff object and replace with a variable declaration for staff as an empty array eg:

```
let staff = [];
```

Note the HTML form to allow the adding of new staff.  Create an event handler for the submission of the form:

```
document.getElementById('addStaffForm').addEventListener('submit',
function(ev){
            ev.preventDefault();
            let newStaffName = document.getElementById('staffName').value;
            let newStaffEmail= document.getElementById('staffEmail').value;
            staff.push({name: newStaffName, email: newStaffEmail});
            console.dir(staff)
})
```

This will retrieve the values submitted from the form and push them into the staff variable.  You should be able to see submitted values via the `console.dir(staff)` call.

Place the `forEach()` inside of a function eg:

```
let rebuildList = function(){
      staff.forEach(function(element) {
            let newStaffRow = document.createElement("tr");
            let newStaffName = document.createElement("td");
            newStaffName.innerHTML = element.name;
            let newStaffEmail = document.createElement("td");
            newStaffEmail.innerHTML = element.email;
            newStaffRow.appendChild(newStaffName);
            newStaffRow.appendChild(newStaffEmail);
            document.getElementById('staffTable').appendChild(newStaffRow);
            });
}
```

Call this function at the end of the submit event.

When testing the new staff member should now appear.

## STORING STAFF DATA IN LOCALSTORAGE

After the `forEach()` in the `rebuildList()` function add the data to localStorage:

```
localStorage.setItem('staffList', JSON.stringify(staff));
```

This uses a method we have not seen before called `JSON.stringify()`.  This converts a Javascript Object into a string.  Remember localStorage values need to be strings.

Finally, replace the empty staff array declaration with the followng:

```
let staff = JSON.parse(localStorage.getItem('staffList'));
      if(staff === null){
            staff = [];
      }
```

This checks to see if there is a localStorage value called staffList and if so uses it to populate the staff array.  Again, the data needs to be manipulated for storage, been converted from a string to a Javascript Object.  The method `JSON.parse()` performs this task.  If the string is correctly formatted it is able to convert it into Javascript object.

Test this page.  There is a bug in that the table is not cleared when new data is added.  Amend the `rebuildList()` function with:

```
document.getElementById('staffTable').innerHTML = "";
```

This will clear the table prior to the loop reassigning the data.

Also, when the page first loads we'd like the table to populate add a call to `rebuildList()` immediately after the function declaration.

Refreshing the page should now show the data stored in localStorage.  You can also close the browser and restart and the data will persist.

## JAVASCRIPT OBJECTS VS JSON

The above application makes use of Javascript Objects.  Javascript Objects are related to a data format known as Javascript Object Notation (JSON).

The JSON format is massively popular way of moving data between applications.

JSON differs from Javascript Objects in that 'names' in the name:value pair should be in double quotes.  Also, unlike Javascript Objects, JSON values cannot be functions.