JavaScript I

Programming with Web Technologies

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato
Auckland ICT Graduate School

Agenda

- Why JavaScript?
- Including JavaScript in your web pages
- Variables
- Strings
- Console debugging
- Getting HTML elements with JavaScript
- Modifying page elements with JavaScript

Why JavaScript?

- JavaScript is a full programming language, within the browser
- Allows us fine-grained control of everything within a webpage.

Quiz: Why is this useful? Thoughts?

Including JavaScript in your web pages

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato
Auckland ICT Graduate School

Including JavaScript in your web pages

The inclusion of JS code in a web page is similar to the inclusion of CSS

- Inline
  - Using HTML element attributes
- Internal
  - In the document using the `script` tag
- External
  - In a separate JS file
  - Included using the `script` tag again

Inline JS

JavaScript works in response to events on the webpage, such as elements being clicked. Many of these events can have the JS code for them supplied as **attributes** on the element they apply to

```
<button type="button" onclick="alert('clicked!')">
   Click me!
</button>
```

A number of events can be dealt with in this way using different attributes, including focus or loss of focus for inputs, mouse movement over an element, and page loading

Internal JS

Uses the HTML **tag** <script>, which can appear in the head or body of the document. JS code is placed between the opening and closing tags

```
<script type="text/javascript">
    alert("Welcome to this webpage!");
</script>
```

When the browser reads the <script> element, it starts executing the code inside it immediately unless the code is inside a function

Inline/Internal JS

Inline event handling quickly become messy, as the length of the method becomes longer. For long event handlers, the inline JS can be moved to an internal function, then called from the inline handler

```
<script type="text/javascript">
    function clickhandler() {
        alert("it's another annoying alert message!");
    }
</script>
...
<p onclick="clickhandler()">A clickable paragraph.</p>
```

External JS

Again, using the HTML **tag** <`script`>, an external JS file can be referenced. This uses the attribute `src` which contains a valid URL to the external source file

```
<script type="text/javascript" src="script.js"></script>
```

If the `src` attribute is present the `script` element **must** be empty. In this case the script behaves as if it was located exactly where the <`script`> tag is

Our code style

- There are many different ways we could write JavaScript code (and hook up event handlers)!
- To avoid confusion, we'll use the same method for *all* our client-side JavaScript code in this course.

In our HTML:

```
<head>
   <meta charset="UTF-8">
   <title>Hello, JavaScript!!</title>
   <script src="./stuff.js"></script>
</head>
```

We'll use external JS, linked in the page `<head>`

In our JavaScript file:

```
window.addEventListener("load", function() {

   // TODO Your code here
   alert("Hello!");

});
```

All code in this block will execute *once the HTML page has finished loading*. This way we can be sure that we can access anything on the page!

Variables

Variables in JavaScript

- Variables are containers which can store data. The data in a variable can be referred to later by the variable name.
- Variables must be declared using either **`let`** or **`const`**.
- Variables can be one of several *data types* (e.g. numbers, text, booleans, arrays, ...). You don't need to declare which data type a variable is, unlike in Java.

```
let meaningOfLife = 42;

const everythingIsAwesome = true;

let greeting = "Hello, World!";
```

A variable called meaningOfLife, with the numerical value 42.

A variable called everythingIsAwesome, with the boolean value true.

A variable called greeeting, with the *string* (text) value "Hello, World!".

Variable names

- You can’t use any JavaScript reserved keywords
  - E.g. boolean, break, case, continue, do, if, else, for, var, true, false, while
- Variables names must begin with a letter or an underscore
  - E.g. myVariable, x, y, _textbox
  - By convention, begin with a lower-case letter and use camelCase.
- Variable names are case-sensitive
  - I.e. myVar is a different variable to MyVar.

Debugging your code

- We can use the `console.log()` function to print anything we like to the *browser console* - including the values of our variables.
  - This can help us see what's going on, and fix any errors!
- The browser console can be accessed using **F12** in Chrome, or **Ctrl+Shift+K** in Firefox.

```
const meaningOfLife = 42;
const greeting = "Hello, World!";

console.log(meaningOfLife);
console.log(greeting);
console.log(3);
console.log("Stuff & Things");
```

Variables - `let` vs `const`

- Any variables declared using **`let`** can be *reassigned* later on.

```
let greeting = "Hello, World!";
...
greeting = "Kia Ora, World!";
```

Assigning the value “Hello, World!” to the greeting variable

Reassigning the value “Kia Ora, World!” to the greeting variable

- Any variables declared using **`const`** cannot.

```
const greeting = "Hello, World!";
...
greeting = "Kia Ora, World!";
```

Assigning the value “Hello, World!” to the greeting constant

Will not work.

- **Best practice:** Use `const` where you can; only use `let` when you need to re-assign.

Variables - **`var`**

- When browsing JavaScript examples online, you'll probably come across variables declared using **`var`**, as well as with `let` and `const`.
- `Var` is the "old way" of declaring variables in JavaScript. It has unintuitive *scoping rules* (we'll talk more about scope later!) and should **never be used** anymore, unless you're writing backward-compatible apps for older versions of JavaScript (which you aren't in this course).

Arithmetic in JavaScript

- We can perform arithmetic in JavaScript, using **+**, **-**, *****, **/**, and brackets **( )**.

```
const x = 1;
const y = 2;
const sum = x + y;
const difference = y - x;
const something = (x * 4) / (y + difference);
```

- We can also use the *increment* and *decrement* operators as shorthand
  - The operators are **++**, **--**, **+=**, **-=**, ***=**, and **/=**
  - These only work if the variables are declared using `let`. **Why?**

```
let a = 4;
let b = 3;
a++; // Same as a = a + 1
b *= 10; // Same as b = b * 10
```

Quiz: What are the values of each of the variables on this slide?

Strings

String concatenation

- We can also *concatenate* (join) two strings together, using **+**.

```
const name = "Alice";
const greeting = "Hello, " + name + "!";
```

- We can even “add” numbers and strings together!
  - The number is converted to a string first, and then concatenated.

```
const age = 21;
const sentence = "Your age is: " + age;
```

Quiz: What are the values of each of the variables on this slide?

Template literals

- If we want to concatenate string literals and variables together, we should use **template literals** rather than concatenation.

```
const name = "Alice";
const age = 21;
const status = "Awesome";
```

Lots of “”’s and +’s - lots of room for error. Difficult to read.

- Compare this:

```
const greeting = "Hello, my name is " + name + ". I am " + age + " and I am " + status + "!";
```

String starts and ends with ` `, rather then “ “. Variables are added with ${}. No need for +’s.

- To this. Much nicer!

```
const betterGreeting = `Hello, my name is ${name}. I am ${age} and I am ${status}!`;
```

Strings ⟵ ⟶ numbers

- If required, you can convert between strings and numbers easily
  - For example, the value of an `<input>` will always be a string, so you might need to convert it to a number if you want to do arithmetic with it.
- Numbers to strings:

```
const num = 4;
const str = `${num}`;
```

(Ab)using template literals to convert for us

- Strings to integers (numbers with no decimal point):

```
const strInt = "42";
const numInt = parseInt(strInt);
```

parseInt() function

- Strings to floating-point numbers (numbers with a decimal point):

```
const strPi = "3.14159";
const numPi = parseFloat(strPi);
```

parseFloat() function

More on strings

- Strings can be thought of as an *array* of characters.
- Each character has an *index*, starting at **0**.
- For example:

```
let myString = "HELLO WORLD";
```

More on strings

- Strings have a number of useful *functions* and *properties* we can use to tell us more about them. Some very useful ones:
- **length:** tells us how many characters are in a string.

```
const myString = "Hello World";
const theLength = myString.length;
```

- **slice():** Gives us back *part of* the original string, given a start (inclusive) and end (exclusive) index.

```
const owo = myString.slice(4, 8);
```

Quiz: What are the values of each of the variables on this slide?

More on strings

- Other really useful functions include:
  - indexOf(), toLowerCase(), toUpperCase(), charAt() (same as Java string methods)
- Excellent reference: W3Schools string reference

```
const myString = "Hello World";
const stringLength = myString.length;
const stringIndex = myString.indexOf("e");
const subString = myString.slice(0, 4);
const stringAllLower = myString.toLowerCase();
const simpleChar = myString.charAt(myString.length - 1);
```

Bonus Quiz (in your own time)**:** What are the values of each of the variables on this slide?

Getting page elements

Getting page elements

- The main reason to use JavaScript on a page is to be able to programatically examine and change page contents. Browsers provide access to the **`document`** object, which lets us do this.
- The functions we use to get elements on a page match nicely with CSS - if you know how to write a CSS selector for a certain group of elements, then you already know how to get those elements using JavaScript!

document.querySelector()

- The `document.querySelector()` function takes a single `string` argument, representing a valid CSS selector.
- The function will return the **first HTML element** on your page which matches the given CSS selector. Any valid CSS selector will work!

```
const elem1 = document.querySelector("p");

const elem2 = document.querySelector("#myP");

const elem3 = document.querySelector("ol > .item");

const elem4 = document.querySelector("li:nth-child(2)");
```

Quiz: What are the values elem3 and elem4?

```
<body>

<h1>Hello, JavaScript!</h1>
<p>This is a paragraph.</p>
<p id="myP">This is also a paragraph,
    with an id "myP".</p>
<ol>
   <li>Item 1</li>
   <li>Item 2</li>
   <li class="item">Item 3</li>
</ol>

</body>
```

document.querySelectorAll()

- The `document.querySelectorAll()` function takes a single `string` argument, representing a valid CSS selector.
- The function will return the an **array of all HTML elements** on your page which match the given CSS selector *(more on arrays next lecture)*.

```
const elem1 = document.querySelectorAll("p");

const elem2 = document.querySelectorAll("#myP");

const elem3 = document.querySelectorAll("ol > li");
```

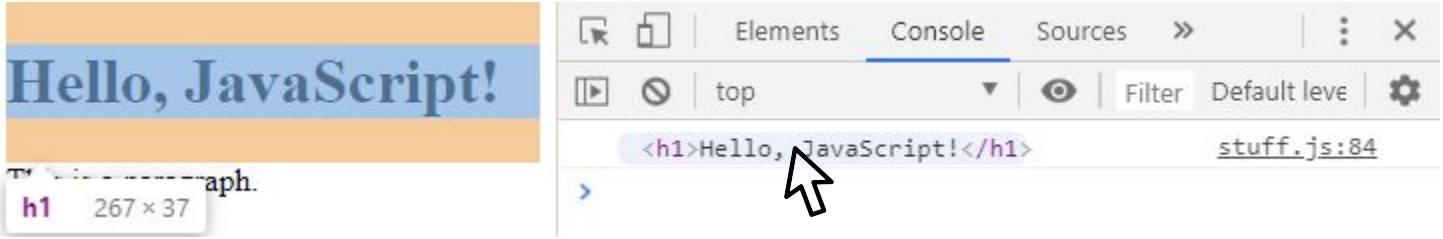
```
<body>

<h1>Hello, JavaScript!</h1>
<p>This is a paragraph.</p>
<p id="myP">This is also a paragraph,
    with an id "myP".</p>
<ol>
    <li>Item 1</li>
    <li>Item 2</li>
    <li class="item">Item 3</li>
</ol>

</body>
```

Debugging tip

- If you use `console.log()` to print out an HTML element returned with `querySelector` / `querySelectorAll`, you can mouse-over the element in the browser console to see it highlighted within your page.

```
const heading = document.querySelector("h1");
console.log(heading);
```

```
<body>
  <h1>Hello, JavaScript!</h1>
  <p>This is a paragraph.</p>
  ...
</body>
```



Getting page elements - The old way

- Before `querySelector` / `querySelectorAll`, we would use other functions to get elements by id, element (tag) name, or class.
- These functions still work, and you'll find many examples online using these.

Get the single element on the page with `id="my-id"` (e.g. `<p id="my-id">…</p>`)

```
let myElement = document.getElementById("my-id");
```

Get an *array* of all `<p>` elements on the page

```
let paragraphs = document.getElementsByTagName("p");
```

Get an *array* of all elements on the page with CSS `class="important"` (e.g. `<p class="important">…</p>`)

```
let importantElements = document.getElementsByClassName("important");
```

Modifying page elements

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato
Auckland ICT Graduate School

Getting / setting element properties

- Elements have many *properties* which can be read / written directly to examine / change what's being displayed.
- Example:

```
const imageElement = document.querySelector("#myImage");

const imgSrc = imageElement.src;

console.log(imgSrc);

imageElement.src = "theLie.png";
```

```
<img id="myImage" src="cake.jpg">
```

Getting the value of a property

Setting the value of a property

Quiz: What will this code snippet print to the console? What will the `<img>` element `src` attribute be set to?

Getting / setting element properties

- Some common properties shown here. There are many more available to use!

Property	Description
`innerText`	Gets / sets the *text* content of an element. ● Does *not* interpret HTML tags. E.g. if we set `innerText` to "`<strong>Hello</strong>`", it will appear that way, verbatim, in the browser.
`innerHTML`	Gets / sets the *html* content of an element. ● *Does* interpret HTML tags. E.g. if we set `innerHTML` to "`<strong>Hello</strong>`", it will appear as **Hello** in the browser.
`src`	Gets / sets the `src` of an image element.
`id`	Gets / sets the `id` attribute of an element.

Getting / setting HTML attributes

- While commonly used HTML attributes such as `id` and `src` have associated JavaScript properties, not all of them do.
- When a property isn't directly available, we can use an element's `getAttribute()` and `setAttribute()` functions to query / modify *any* attribute.
- Example:

```
const ol = document.querySelector("ol");

const isReversed = ol.getAttribute("reversed");

ol.setAttribute("type", "A");
```

Get a value indicating whether the `<ol>` is in reverse order

Sets the `type` of the `<ol>` to "A. B. C. ..." etc.

Changing CSS

- CSS can be queried / changed by getting / setting properties found inside an element's `style` property. For example:

```
const p1 = document.querySelector("#p1");
p1.style.fontFamily = "Gotham";
```

- **Note:** JavaScript names for CSS properties are slightly different. For example:

CSS property names:	`font-family`, `color`, `margin-left`, `border-bottom-width`, ...
JavaScript property names:	`fontFamily`, `color`, `marginLeft`, `borderBottomWidth`, ...

- Yes, this is silly. Unfortunately you do need to watch out for it.

Changing CSS

- We can also add or remove CSS classes from elements, through their `classList` property:

```
bauble.classList.add("animated");
```

Adds the given CSS class to an element

```
bauble.classList.remove("animated");
```

Removes the given CSS class from an element

```
bauble.classList.toggle("animated");
```

Toggles the given CSS class on an element (i.e. adds it if it wasn't there already, removes it if it was)

Handling button clicks

Intro to event handling

- Usually, when we write JavaScript code, we don't want it all to run at once.
  - We want some of our code to run **in response to events** such as when the user clicks a button or types some text into an `<input>`.
- To do this, we can add an **event handler** to an element.
  - An event handler is a **function** which will run whenever something happens.
  - We've already seen one of these - the `window.addEventListener("load", …)`.
- We'll cover event handling and functions in much more detail in later lectures - but for now, let's see how we can do something when the user clicks a button (or any other page element)

Responding to a button click

- Use the `addEventListener()` function to add handlers for various events.
- A list of available events can be found here.

The name of the event we're handling

The "event" parameter can give us info about the event.

```
const myButton = document.querySelector("#clickme");

myButton.addEventListener("click", function(event) {

  console.log(event);

});
```

Any code in this function will run when the event occurs (clicking the button, in this case).

Intro to event handling

- There are other ways we can do event handling in JavaScript, aside from `addEventListener()`.
  - HTML event attributes, e.g. `<button onclick="...">Click me!</button>`
  - JavaScript event properties, e.g. `myButton.onclick = function() { ... }`
- We will not be teaching these in this course, nor will we expect you to learn them yourself.

Further reading

- W3Schools Intro to JavaScript
- Reference: `var` vs `let` vs `const`
- Reference: Element properties and methods
- Reference: Getting and Setting attributes
- Reference: JavaScript events