

JavaScript II

Programming with
Web Technologies



Auckland
ICT Graduate School

Last time, in JS I...

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

Quiz

1. What is the difference between let and const?
2. How could I convert a string to a number?
3. Given a string, how can I tell how many characters are in that string?
4. How can I get the first element on a page matching a particular CSS selector?
5. How would I change the background color of an element using JavaScript?
6. How would I change the text content of an element using JavaScript?
7. How would I add or remove a CSS class to or from an element using JavaScript?

Agenda

- Booleans & conditionals (`if-else` statements)
- Loops
 - `for`-loops
 - `while`-loops
- Arrays
 - `forEach()` function
- Refresher on `document.querySelectorAll()`
- Adding and removing HTML elements with JavaScript

Booleans & conditionals

Boolean variables

- A **boolean** is a value that is either **true** or **false**. We can easily create boolean variables:

```
const javascriptIsAwesome = true;  
const iAmConfused = false;
```

- We can also use **relational operators** to create **boolean expressions**
 - Boolean expressions are those which evaluate to either **true** or **false**
- The relational operators are:
 - `==` (equal), `!=` (NOT equal), `===` (type-safe equal), `!==` (type-safe NOT equal)
 - `>` (greater than), `<` (less than),
 - `>=` (greater than or equal), `<=` (less than or equal)

Relational operators

Operator	Meaning	Example
<code>==</code>	Equal to	<code>x == y</code> (x is equal to y)
<code>!=</code>	NOT equal to	<code>x != y</code> (x is not equal to y)
<code>===</code>	Equal to (type-safe)	<code>x === y</code> (x is equal to y AND of the same type)
<code>!==</code>	NOT equal to (type-safe)	<code>x !== y</code> (x is not equal to y OR is not the same type)
<code>></code>	Greater than	<code>x > y</code> (x is greater than y)
<code><</code>	Less than	<code>x < y</code> (x is less than y)
<code>>=</code>	Greater than or equal	<code>x >= y</code> (x is greater than or equal to y)
<code><=</code>	Less than or equal	<code>x <= y</code> (x is less than or equal to y)

Boolean expressions

- Boolean expressions always evaluate to either **true** or **false**.

```
const age = 83;
const x = 33;
const greeting = "Hello, World!";
const b1 = (7 <= 12);
const b2 = (35 != 35);
const b3 = (7 >= x / 10);
const b4 = (x == 42);
const b5 = (age > 35);
const b6 = (x < age);
const b7 = (greeting === "Hello, World!");
const b8 = (greeting.charAt(4) !== "o");
```

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

Boolean expressions - == / != versus === / !==

- === and !== perform *type checking* in addition to value checking.

```
const b9 = (42 === "42");
```

b9 will be **false**, because a string can never be equal to a number.

- == and != don't perform type checking - they try to *coerce* the variables into the same type.

```
const b10 = (42 == "42");
```

b10 will be **true**, because JavaScript sees that the string "42" **looks like** a number, so converts it before doing the comparison!

Logical operators

- We can **combine** multiple boolean expressions into one larger expression using **logical operators**. Examples:
 - Only accept new registrations from users between the ages of 18 **AND** 99
 - Allow students **OR** lecturers to download course content for free
 - Do **NOT** let users post articles which contain profanity

Operator	Syntax	Example	Meaning
AND	&&	<code>age >= 18 && age <= 99</code>	Age is greater than or equal to 18 and less than or equal to 99
OR	 	<code>position === "student" position === "lecturer"</code>	Position is equal to either "student" or "lecturer"
NOT	!	<code>!text.contains(word)</code>	Text does not contain word

Logical operators

```
const value = 5;  
  
const b11 = (value > 10 && value < 100);  
const b12 = (value > 10 || value == 1);  
const b13 = !(value > 10);  
const b14 = !(value > 10 || value == 1);
```

Quiz: In English, what to each of the boolean expressions on this page mean? What are the values of each of the variables on this slide?

Conditionals (if-else statements)

- Often, we want to execute certain code under certain conditions (i.e. *“If some condition is true, then do something”*).
- To do this, we use if-else statements.
- General form:

```
if ( [condition] ) {  
    [statements]  
}
```

← The **condition** is a boolean expression

← Any code in this block will execute **if and only if** the condition is true. Otherwise they will be ignored.

Conditionals (if-else statements)

- We can extend the basic if statement with any number of else if blocks
 - This lets us write statements of the form *"If some condition is true, then do something. Otherwise, if some other condition is true, do something else. Otherwise, if some other condition is true, do something else..."* etc.

```
if ([condition 1]) {  
    [statements 1]  
}  
else if ([condition 2]) {  
    [statements 2]  
}  
...
```

If condition 1 is true, execute the code in this block.

Otherwise, if condition 2 is true, execute the code in this block.

Can have as many else-if's as we like. The first one where its condition is true will be executed.

Conditionals (if-else statements)

- Finally, we can conclude our if-else statement with an `else` block, which will be executed only if all conditions are false.
 - I.e. "If some condition is true, do something. Otherwise, if some other condition is true, do something else. Finally, if none of the conditions are true, do something else".*

```
if ([condition 1]) {  
    [statements 1]  
}
```

If condition 1 is true, execute the code in this block.

```
else if ([condition 2]) {  
    [statements 2]  
}
```

Otherwise, if condition 2 is true, execute the code in this block.

```
else {  
    [statements 3]  
}
```

Otherwise, if condition 1 and 2 are both false, execute the code in this block.

Conditionals (if-else statements)

```
if (x == 5) {  
    console.log("A.");  
}  
else if (x > 3 && x < 10) {  
    console.log("B.");  
}  
else if (x >= 7 && x < 20) {  
    console.log("C.");  
}  
else {  
    console.log("D.");  
}
```

Quiz: What is printed to the console:

- When x is 1?
- When x is 5?
- When x is 8?
- When x is 15?
- When x is 20?

Loops



Auckland
ICT Graduate School

Webpage development problem

- Design an HTML `<table>` which shows the 3-times table, from “3 * 1 = 3”... all the way up to “3 * 1000 = 3000”!
- Clearly, we need a better solution than this...

```
<table>
```

```
<tr><th>Multiplier</th><th>Result</th></tr>
```

```
<tr><td>1</td><td>3</td></tr>
```

```
<tr><td>2</td><td>6</td></tr>
```

```
<tr><td>3</td><td>9</td></tr>
```

```
<tr><td>4</td><td>12</td></tr>
```

```
...
```

Loops

- Loops allow us to execute the same code over and over again until certain conditions are met.
- Common situations where loops are used:
 - Counters
 - Reading an unspecified number of values
 - Iterating through a sequence of data
 - Doing something X number of times

while-loops

- while-loops appear similar to an if-statement, but with the keyword **while** instead of if.
- Code inside the loop is executed continuously **until the condition is false** (if the condition was never true to begin with, the code is never executed).

```
let counter = 0;
while (counter < 10) {
  console.log(counter);
  counter++;
}
```

Quiz: What is the output of the program on this slide?


for-loops

- for-loops are equivalent (*functionally identical*) to while-loops but use a different *syntax*. Generally, for-loops are used for iterating through collections or for doing something a specified number of times.

1. This **initializer** is run *once* first, before any code inside the loop.

2. This **condition** is evaluated *before* each iteration of the loop. If false, the loop will exit.

3. This **increment** is run *after* each iteration of the loop.



```
for (let counter = 0; counter < 10; counter++) {  
  console.log(counter);  
}
```

- This for-loop is equivalent to the while-loop on the previous slide.

Arrays



Auckland
ICT Graduate School

Arrays

- For CS718 folk: JavaScript has arrays, which are *similar to* - but *not* the same as - Java arrays.

Similarities	Differences
<ul style="list-style-type: none">• Zero-based indices• Bracket [] syntax for indexing arrays• length property for determining size• for-loop syntax for looping through array elements	<ul style="list-style-type: none">• Syntax for creating new arrays• Can grow or shrink in size after they've been created (like Java Lists).• Extra functions for adding to / removing from / iterating through

Arrays

- Arrays can be used to hold more than one item of data at the same time.
- In JavaScript, array size is **unbounded** – an array can have any number of elements, and each will resize itself as needed to fit all of them
 - CS718 folk: In this way, JavaScript arrays are similar to Java Lists!
- We can create arrays with the following syntax:

```
const array1 = [];
```

Creating an initially empty array

```
const x = 10;
```

```
const array2 = ["Stuff", 4, "Things", x, 3.7];
```

Creating an array pre-populated with elements

Arrays can contain any kind of data.

Arrays - Indexing elements

- We can access individual “slots” - or *elements* - in an array using the `[]` (*square bracket*) notation.
- Within the `[]`, supply the *index* of the element you’re trying to access. The index of the *first element* in the array is **0**.
- We can both **get** and **set** element values this way.

```
const numbers = [1, 2, 3, 4, 5];
```

```
console.log(numbers[0]);
```

Will print “1”.

```
console.log(numbers[4]);
```

Will print “5”.

```
console.log(numbers[5]);
```

Will print “undefined” (there is no element at this index).

```
numbers[2] = 10;
```

Will replace the 3 above with the value 10.

Arrays - Adding elements

- We can use `[]` notation to add elements past the end of the array.
 - Elements in the “gaps” will be set to undefined.

```
let myArray = ["Stuff", "Things", "Foo"];
```

```
myArray[5] = "Bar";
```

- We can also call use the `push()` function to add elements to the end of an array, or use the `unshift()` function to add elements to the beginning.

```
myArray.push("Baz");
```

```
myArray.unshift("Thomas");
```

Contents after these lines:

```
["Thomas", "Stuff", "Things", "Foo", undefined, undefined, "Bar", "Baz"];
```

Arrays - Removing elements

- We can use `shift()` to remove and return the *first* element, and `pop()` to remove and return the *last* element

```
let myArray = ["Stuff", "Things", "Foo"];
```

```
let stuff = myArray.shift();
```

"Stuff"

```
let foo = myArray.pop();
```

"Foo"

Contents after these lines:
["Things"]

- Note that when we remove elements like this, the size of the array changes.

Iterating through arrays

- Arrays have a property called **length**, just as strings do. It returns the number of elements in the array.
- We can use this in combination with a for-loop to iterate through every element in an array.

```
const myArray = ["Arrays", "Are", "Really", 42, "Awesome!"];

for (let index = 0; index < myArray.length; index++) {
  const element = myArray[index];
  console.log(`myArray[${index}] = ${element}`);
}
```

Quiz: What is the value of `myArray.length`? What is the output of this program?

Iterating through arrays

- We can also use the `forEach()` function. This takes a **callback function** (more on these in future lectures!) which will be called once per item in the array. The callback receives two parameters: the current element in the array and (optionally) the index of that element.

```
let months =  
    ["Jan", "Feb", "Mar", "Apr", "May", "Jun"];  
  
months.forEach(function(theMonth, index) {  
    console.log(`Month #${index} is ${theMonth}`);  
});
```

In this example, the provided function will be called six times. `theMonth` will be "Jan" through "Jun", and `index` will be 0 through 5, respectively.

Array functions & properties

- Arrays have a huge number of functions and properties which might be useful to you
- A comprehensive list, with examples, is available [here](#).

querySelectorAll() revisited

document.querySelectorAll()

- From last lecture: 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.

```
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>
```

Iterating through HTML elements

- Since the result of `querySelectorAll()` is an array, we can use any of the array functions and properties we've just seen.
- The most common use case is to iterate through all HTML elements in the returned array, and do something with them.

```
const imp = document.querySelectorAll(".important");
for (let i = 0; i < imp.length; i++) {
  imp[i].style.color = "red";
}
```

```
const items = document.querySelectorAll("ol > li");
items.forEach(function(element, index) {
  element.innerHTML = `List item #${index}`;
});
```

Quiz: In plain English, describe the functionality of the two programs above.

document.querySelectorAll() - Common mistake

- Common mistake:

```
let importantElements = document.querySelectorAll(".important");  
importantElements.style.color = "red";
```

- **Remember:** querySelectorAll() **always** returns an array.
 - Even if *you* know there's only one matching element on the page, the computer is dumber than you are :)
- Even if there's only one element in the array, it's still an array. You'd need to access the first element in the array to modify it, like this:

```
importantElements[0].style.color = "red";
```

Adding & removing HTML elements

Adding elements - Using innerHTML

- We've already seen one way in which we can add new elements to a page:
 - By appending to an existing element's `innerHTML` property.
- For example, to append a new `<p>` to the `<body>`, we could do something like:

```
const content = "New paragraph content!";
```

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

```
body.innerHTML += `

${content}</p>`;


```

Adding elements - Using `document.createElement()`

- If we want more fine-grained control, we can create new elements using the `document.createElement()` method, like so:

```
const para = document.createElement("p");
```

Just specify the tag name of the new element you wish to create (a `<p>` in this case).

- We can modify any of a new element's properties just as we can modify existing elements:

```
para.innerHTML = "This is a paragraph!";  
para.style.color = "red";
```

Adding elements - Using document.createElement()

- Once we have our new element, we need to put it somewhere!
 - We do this by getting the element that will be its *parent*, and using the parent's `appendChild()` method.
- Example:

```
const div = document.querySelector(".container");
```

```
const para = document.createElement("p");
```

```
...
```

```
div.appendChild(para);
```

```
<div class="container">  
  <h1>New elements go here!</h1>
```

New <p> will be inserted here.

```
</div>
```

Adding elements - Comparison of approaches

- Using `innerHTML` can be easier and less time-consuming to code - particularly when creating large numbers of nested elements in one go.
 - Imagine creating a table with 10 columns, using `createElement()` for all those `<td>`s...
- Using `createElement()` provides us with more fine-grained control, and is required in certain cases - particularly if we want to add an event handler to the element we create
 - e.g. add a new `<button>` to the page which does something when clicked
- Generally, `createElement()` is seen as **best practice** - but in this course, you're free to use whichever you prefer, as long as the result works!

Removing Elements

- We can easily remove an element from the page using its `remove()` method.

```
const thingToRemove = ...;
```

```
thingToRemove.remove();
```

Further reading

- W3Schools [intro to JavaScript](#)
- Reference: [if-statements](#)
- Reference: loops ([for](#), [while](#))
- Reference: [JavaScript arrays](#)
- Reference: [querySelectorAll\(\)](#)
- Reference: [adding elements](#)