JavaScript III

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

So far, in JavaScript...

- **Programming fundamentals**
  - Variables, numbers, strings, boolean logic, arrays
  - Conditionals (`if-else`), loops (`for`, `while`, `.forEach()`)
- **Finding HTML elements**
  - `document.querySelector()`, `document.querySelectorAll()`
- **Modifying HTML elements**
  - `.innerHTML`, `.style`, `.classList`, etc...
- **Adding & removing HTML elements**
  - `.innerHTML`, `document.createElement()`, `.appendChild()`, `.remove()`
- **Some event handling**
  - Window load, Button click

Quiz

1. How would I access the third element in the array called myArray?
2. Give an example of how I could write an infinite loop?
3. How would I set the text color of all <p> elements to red using JavaScript?
4. When would the else { … } block be executed?
5. What are two methods of adding new elements to a page using JavaScript? What are the strengths and weaknesses of each approach?

Agenda

- More on functions
- More on event handling
  - Timer events
- JavaScript objects & JSON

Functions

Functions

- So far, we have seen functions used in several places. For example:

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

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

    console.log(event);

});
```

Event handling (e.g. button clicks, window load...)

```
let months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"];

months.forEach(function(theMonth, index) {

    console.log(`Month #${index} is ${theMonth}`);

});
```

Iterating through arrays using `forEach()`

Functions

- The functions we have seen so far are examples of *anonymous* functions. We just specify the function arguments (if any), and the code inside the function.
- We can instead give functions a *name* using slightly different syntax:

```
function handleButtonClick(event) {
    console.log(event);
}

const myButton = document.querySelector("#clickme");
myButton.addEventListener("click", handleButtonClick);
```

This is a function named **`handleButtonClick`**. It takes one argument, called `event`.

This code specifies that whenever `myButton` is clicked, the `handleButtonClick` function should be invoked. Note that there are **no brackets ( )** when we name the `handleButtonClick` function here.

Functions

- Using named functions, we can also **invoke** them (call them) ourselves, in addition to being able to supply them as event handlers / callbacks.

```
function greet(name) {
    console.log(`Hello, ${name}!`);
}
```

This is a function named **greet**. It takes one argument, called name.

```
greet("Andrew");
greet("Tyne");
greet("Yu-Cheng");
```

This code invokes the greet function, supplying the string “Andrew” as the name argument. It then invokes the function twice more, supplying different values each time.

Quiz: What will this program’s console output be?

Functions - Returning values

- We can define our functions to **return** - or “give back” - a result, just as many built-in functions do. We use the `return` keyword for this.

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

Just as the `querySelector()` function returns (gives us back) an HTML element...

```
function add(a, b) {
   const sum = a + b;
   return sum;
}

const myResult = add(4, 5);
console.log(myResult);
```

... This `add()` function returns (gives us back) the result of adding a + b.

Quiz: What will this program’s console output be?

Functions - Returning

- More generally:
  - The **`return`** keyword causes a function to immediately exit. No more code inside that function will be executed.
  - If you just want a function to quit, you may type **`return;`** without supplying a value
  - If you supply a value (as on the previous slide), that value will be given back to the code which invoked the function.
  - You may include the return keyword multiple times within a function. The first one to be reached will take effect.

Functions - Syntax

The **name** of the function. Can't include spaces or start with a number. By convention, starts with a lowercase letter.

A comma-separated list of **arguments** (values, parameters) to be supplied to the function in order for it to do its job.

```
function add(a, b) {
  const sum = a + b;
  return sum;
}
```

The **body** - consists of **statements** (code) which will be executed when the function is **invoked** (called)

Optionally, functions may **return** a value. This value will be supplied to the code which invoked the function.

Function benefits - Reuse

- Functions allow us to **reuse** our code. Imagine checking multiple years for leap-year status if we couldn't write a function for it!

```
function isLeapYear(year) {

    if (year % 4 != 0) {
        return false;
    }
    else if (year % 100 != 0) {
        return true;
    }
    else if (year % 400 != 0) {
        return false;
    }
    else {
        return true;
    }
}
```

```
console.log(isLeapYear(2019));
console.log(isLeapYear(2004));
console.log(isLeapYear(2100));
console.log(isLeapYear(2000));
```

Quiz: What will this program's console output be?

Function benefits - Readability

- Use of well-named functions can increase readability of your code. Compare this version of `isLeapYear()` to the one on the previous slide.

```
function isLeapYear(year) {

    if (!isDivisibleBy(year, 4)) {
        return false;
    }
    else if (!isDivisibleBy(year, 100)) {
        return true;
    }
    else if (!isDivisibleBy(year, 400)) {
        return false;
    }
    else {
        return true;
    }
}
```

```
function isDivisibleBy(value, divisor) {
    return (value % divisor == 0);
}
```

Function benefits - Reducing nesting

- Nesting refers to having multiple *control structures* (e.g. function definitions, if-statements, loops, etc) inside one another.
- The more layers of nesting you have, the harder your code is to understand.

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

paragraphs.forEach(function(p) {
   p.addEventListener("click", function(event) {

       if (event.target.innerHTML === "GOOD") {
           event.target.style.color = "green";
       }
       else {
           event.target.style.color = "red";
       }

   });
})
```

An if, inside an anonymous function, inside another anonymous function...

Function benefits - Reducing nesting

- We can *extract* some of our complicated code out into a function to reduce the nesting level of our program
- Changing anonymous functions into named functions will achieve the same effect.

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

paragraphs.forEach(function(p) {
   p.addEventListener("click", handleParagraphClick);
});

function handleParagraphClick(event) {
   if (event.target.innerHTML === "GOOD") {
       event.target.style.color = "green";
   }
   else {
       event.target.style.color = "red";
   }
}
```

By changing one of the anonymous functions into a named function, we've reduced the maximum nesting level of our program!

More on event handling

Event handling on HTML / DOM elements

- `addEventListener()` is the preferred method for registering an *event handler* on an HTML element in modern JavaScript.
- We supply the name of the event we're handling (e.g. "click"), and a function (anonymous or named) to be called when the event occurs
- The event handler function takes a single argument - **`event`** - supplying information about the event
  - E.g. **`event.target`** will give us the HTML element which raised the event (e.g. the button which was clicked)
- We can use the `removeEventListener()` function later on if we don't want to handle that event any more.

Event handling - Anonymous vs named functions

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

    console.log(event);

});
```

Use of anonymous event handler function

```
function handleButtonClick(event) {
    console.log(event);
}

myButton.addEventListener("click", handleButtonClick);
```

Use of named event handler function

Event handling on HTML / DOM elements

- There are over 100 different events in JavaScript code.
- Different events are broken into categories based on what causes them. These include Mouse, Keyboard, Clipboard, Media, and Touch events
- The most commonly used events are **`change`**, **`click`**, **`load`**, **`focus`** and **`keydown`**.
- A complete list of events: [MDN Events reference page]
- All of these events are used with `addEventListener()` / `removeEventListener()`.

Event handling - **this**

- We have seen that we can get the HTML element that raised an event, using event.target.
- We can also get the element using **this**.

```
function handleButtonClick() {
    console.log(this);
}

myButton.addEventListener("click", handleButtonClick);
```

Will log whichever button was clicked to the console.

Timer events

Timer events in JavaScript

- So far, we've looked at how we can execute code in response to an event generated by an HTML element (e.g. a button being clicked)
- Often, we want to execute code *after a certain delay*, or *repeatedly at certain intervals*. We use **timer events** for this.
- There are two types of timer events:
  - **One-shot** timers wait for a specified duration, then execute their code
  - **Continuous** timers repeatedly execute their code, waiting for a specified duration between each execution
- You can have as many timers as you want in your code.

One-shot timers - `setTimeout()`

- To start a one-shot timer, invoke the **`setTimeout()`** function. This takes two arguments - a function to call, and a delay (in milliseconds) after which that function will be called.
- The function returns a value which we can use to stop the timer later if required (see later slide)

```
function sayHello() {
   console.log("Hello, COMPSCI 719!");
}

const timer = setTimeout(sayHello, 3000);
```

This will cause the `sayHello` function to be called once, after three seconds.

Continuous timers - `setInterval()`

- To start a continuous timer, invoke the **`setInterval()`** function. This takes two arguments - a function to call, and a delay (in milliseconds) between repeated calls to that function.
- The function returns a value which we can use to stop the timer later if required (see later slide)

```
function sayHello() {
   console.log("Hello, COMPSCI 719!");
}

const timer = setInterval(sayHello, 3000);
```

This will cause the `sayHello` function to be called repeatedly, every three seconds.

Stopping timers

- We can stop any one-shot timer using `clearTimeout()`. We can stop any continuous timer using `clearInterval()`.

```
let timer = setTimeout(sayHello, 3000);
...
clearTimeout(timer);
```

To stop a timer, we supply the value we got from `setTimeout()` / `setInterval()`, to `clearTimeout()` / `clearInterval()`.

```
let timer = setInterval(sayHello, 3000);
...
clearInterval(timer);
```

Preventing duplicate timers

- Consider the following code. This will start a new continuous timer every time the button is clicked, which is probably not what you want!

```
function sayHello() {
    console.log("Hello, COMPSCI 719!");
}

const myButton = document.querySelector("#clickme");
myButton.addEventListener(("click"), function() {
    let timer = setInterval(sayHello, 1000);
});
```

Quiz: How do we solve this problem?

Hint: We need a way to check if a timer is running first, before starting a new one...

Preventing duplicate timers

```
let timer = null;

const startButton = document.querySelector("#startButton");
startButton.addEventListener(("click"), function() {
    if (timer == null) {
        timer = setInterval(sayHello, 1000);
    }
});

const stopButton = document.querySelector("#stopButton");
stopButton.addEventListener(("click"), function() {
    if (timer != null) {
        clearInterval(timer);
        timer = null;
    }
});
```

Initialize a timer variable to null

Only start the timer if timer == null

Only stop the timer if timer != null

If we do stop the timer, set the variable back to null so we know we're allowed to start one again.

JavaScript Object Notation (JSON)

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

JavaScript Object Notation (JSON)

- Often, we want to represent more complex data than just numbers, strings, and booleans.
- For example, what if we want to store information about a **person** in our code?
  - Let's say a person has a **name**, an **age**, and an **address**...

```
const personName = "Walter White";
const personAge = 50;
const personAddress = "308 Negra Arroyo Lane";
```

- These values are all related (i.e. they refer to the same person), but there's nothing tying them together. Can we do better than this?

JavaScript Object Notation (JSON)

- JavaScript Object Notation (JSON) was designed as a simple way to represent more complex data.
- JSON has its beginnings in JavaScript, but is now a ubiquitous *data interchange format* which has replaced XML in many applications.
- JSON is a simple, user-readable data format. It is well documented in a single page. A JSON document is comprised of:
  - Key-value pairs, separated with full colons **:**
  - Comma-separated elements
  - Five data types (`object`, `array`, `string`, `number`, `boolean`) and `null`

JSON

- Let’s see our person as a JSON document:

```
{
 "name": "Walter White",
 "age": 50,
 "address": "308 Negra Arroyo Lane"
}
```

Keys

Values

JavaScript Objects

- In JavaScript, the notation is virtually identical, but without quotes " " around the keys.

```
const person = {
    name: "Walter White",
    age: 50,
    address: "308 Negra Arroyo Lane"
};
```

- To access individual properties of our JSON
  - We can use **dot notation**:

```
console.log(person.name);
person.age = 51;
```

  - Or **[ ] notation:**

```
console.log(person["address"])
person["age"] = 100;
```

More complex example

```
let person = {
    firstName: "Walter",
    lastName: "White",
    isAlive: false,
    age: 52,
    address: {
        streetAddress: "308 Negra Arroyo Lane",
        city: "Albuquerque", state: "NM",
        country: "USA"
    },
    phoneNumbers: [
        {type: "home", number: "212 555-1234"},
        {type: "office", number: "646 555-4567"},
        {type: "mobile", number: "123 456-7890"}
    ],
    spouse: null
}
```

String

Boolean

Number

Object

Array

null

More complex example

```
let person = {
    firstName: "Walter",
    lastName: "White",
    isAlive: false,
    age: 52,
    address: {
        streetAddress: "308 Negra Arroyo Lane",
        city: "Albuquerque", state: "NM",
        country: "USA"
    },
    phoneNumbers: [
        {type: "home", number: "212 555-1234"},
        {type: "office", number: "646 555-4567"},
        {type: "mobile", number: "123 456-7890"}
    ],
    spouse: null
}
```

person.firstName

?

person.age

?

?

?

More complex example

```
let person = {
    firstName: "Walter",
    lastName: "White",
    isAlive: false,
    age: 52,
    address: {
        streetAddress: "308 Negra Arroyo Lane",
        city: "Albuquerque", state: "NM",
        country: "USA"
    },
    phoneNumbers: [
        {type: "home", number: "212 555-1234"},
        {type: "office", number: "646 555-4567"},
        {type: "mobile", number: "123 456-7890"}
    ],
    spouse: null
}
```

person.firstName

person.isAlive

person.age

person.address.state

person.phoneNumbers[0].number

person.spouse

JavaScript Objects ⟵ ⟶ Strings

- Sometimes we need to convert between JavaScript objects and strings.
- To convert a JavaScript object to a string, use the **`JSON.stringify()`** function

```
const person = { name: "Walter White", age: 50 };
const string = JSON.stringify(person);
```

```
{"name": "Walter White", "age": 50}
```

- To convert a string to a JavaScript object, use the **`JSON.parse()`** function.

```
const alsoWalter = JSON.parse(string);
console.log(alsoWalter.name);
```

```
"Walter White"
```

Further reading

- W3Schools intro to JavaScript
- Reference: Functions
- Reference: JavaScript timers
- Reference: JavaScript objects
- Reference: JSON