COLLAGE OF COMPUTER SCIENCE & ENGINEERING

UNIVERSITY OF JEDDAH

كلية علوم و هندسة الحاسب

جامعة جدة

JS: JavaScript Fundamentals

CCSW 321 (Web Development)

What will be covered

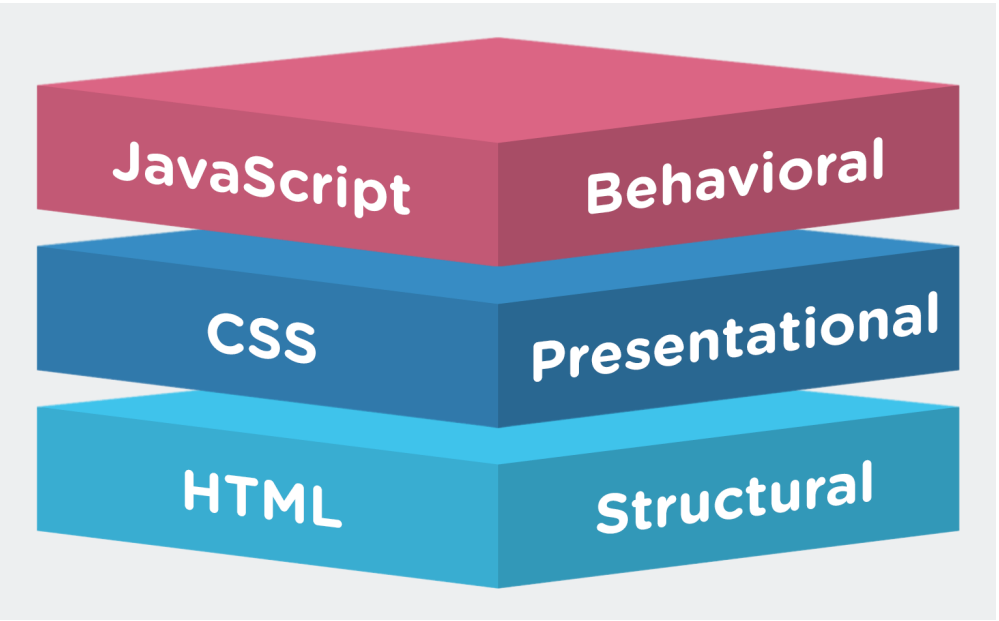
- JS Introduction.
- Variables and Scope.
- Datatypes and Operators.
- Functions and Objects.
- Arrays.
- Exception Handling.

What will be covered

- It is **important** to understand that we will **not** cover every JavaScript aspect. We aim to introduce the important concepts to you.
- You need to get **hands-on experience** with JavaScript to gain a **deeper understanding** of these concepts.
- It is **recommended** that you try to do JavaScript mini projects **by yourself**.

JS Introduction

- **JavaScript** often abbreviated as JS, is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS.
- JavaScript is used to describe behavior, i.e., add functionality or interactivity.



BE AWARE
Java != JavaScript

JS Introduction

JavaScript…

- **Object-oriented**.
  - Everything is an object, including primitives and functions.
- **Interpreted language**.
  - Native execution in browser. There's no compilation by the developer. Modern browsers use a technology known as **Just-In-Time (JIT) compilation**, which compiles JavaScript to executable bytecode just as it is about to run.
  - There is **no** "main method". The script file is executed from **top to bottom**. However, **classes and functions** are **moved** to the **top** of the scope in which it's defined **no matter where you place them in the code**.
  - The **loading order** of files **in** your **HTML** is important. You should **first** load all your dependencies, and then load your own code.

JS Introduction

JavaScript…

- **Dynamically typed** (like Python).
  - No declared type, but values have types.
- **JavaScript is case sensitive.**
  - Not using the proper uppercase and lowercase letters is a syntax error.
- **Semicolons are optional**.
  - However, still use them to end statements.

JS Introduction

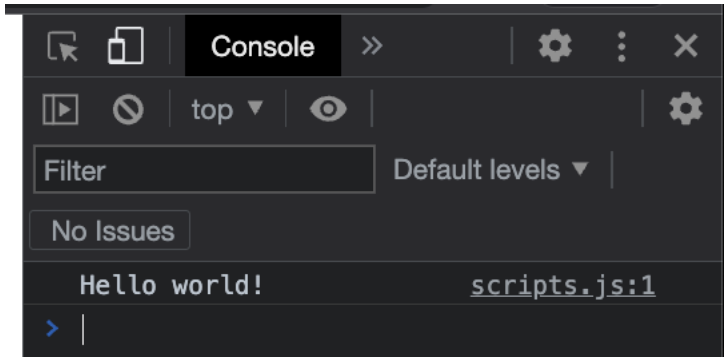
- **HTML** can embed JavaScript files into the web page via the <**script**> tag.
- You can print log messages in JavaScript console by calling console.log();

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js"></script>
  </head>
  <body>
  </body>
</html>
```

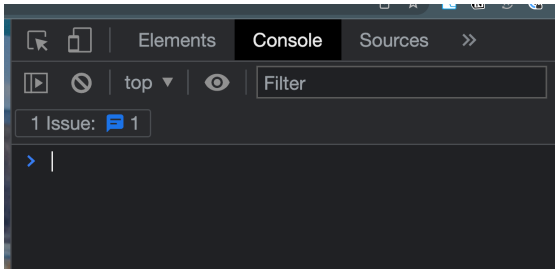
script.js

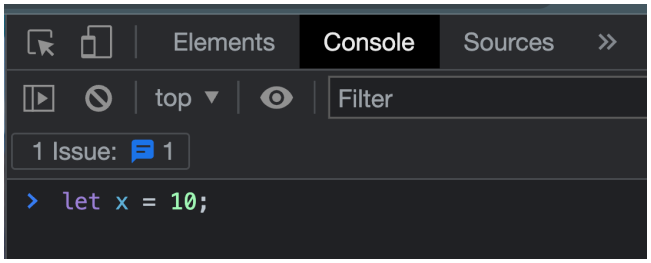
```
console.log('Hello world!');
```

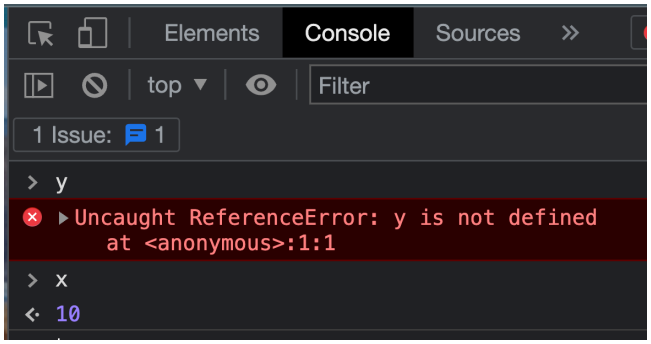


JS Introduction

- Tools for working with **JavaScript**:
  - **Code Editor**: Recommendation is VS Code.
  - **Modern browser**: Recommendation is Google Chrome.
  - **Console**: We will start with the browser's console.







JS code can be written directly in console. Also, you can interact with any existing JavaScript code that is embedded/loaded in the page.

The console prints out your errors. Use it to debug your code!

JS Introduction

- The **echo system** around JavaScript is growing rapidly. Here is a **quick summary** to help you gain a better overall understanding.
  - **JavaScript (JS)**: The core language. Sometimes referred to as Vanilla JavaScript..
  - **ECMAScript**: The browser's implementation of the JS language specification, i.e., how the language should be implemented by the browser.
  - **React, Vue, Angular**: JavaScript frameworks that help us build front-end web applications. For example, React simplifies how we mix HTML and JS code.
  - **NPM, WebPack, Gulp:** Build and infrastructure tools that helps us optimize our JavaScript code and environment. For example, NPM is a dependency manager that we use to install and update all the libraries we use as part of our project.
  - **Node.js:** JavaScript server runtime environment. It is used to run JavaScript outside the browser, i.e., on the server.

JS Introduction

Including JavaScript to a page

- We can add JavaScript code to a page using the **<script>** tag.
- We can use an **inline** approach, an **embedded** approach, or **external file** approach:

Inline JS

```
<button onclick=
"console.log('You
Clicked Me');" >
This is a button

</button>
```

Embedded JS

```
<head>
  <script>
  console.log('You
  Clicked Me');
   </ script >

</head>
```

External JS

```
<head>
 <script
 src="scripts.js">
</head>
```

script.js

```
console.log('Hello world!');
```

JS Introduction

- The **placement** of the <script> tag that allows us to include our JavaScript code can **affect its run time**.
  - Placing <script> inside the <head> tag will result in the code being executed **before** the page is loaded.
  - Placing <script> at the end of the <body> tag will result in the code being executed **after** the page is completely loaded.
- Often, JavaScript is included in the <head> section of the HTML5 document. We can add the **defer** attribute to request that our code is run only **after the page is loaded**. In JavaScript we usually manipulate the content of the page, thus, we need the page to be fully loaded before we run our JS code. Otherwise, we will run into the null error.

JS Introduction

Including JavaScript as an **external file** is the recommended approach. Why?

- Separate behavior from content and presentation.
- Define behavior once and re-use for different pages.
- Easier modification: When changes to the behavior are required, you need to modify only a single JS file to make style changes across all the pages.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js" defer></script>
  </head>
  <body>
  </body>
</html>
```

Do not forget to use the **defer** keyword

JS Introduction

- JavaScript has a similar syntax to the **Java/c/c++** programming languages. You have already extensively studied at least one of those languages in previous courses.

Variables:

```
let x = 10;
let name = "Moayad";
```

Conditionals:

```
if (condition) {
     exprIfTrue;
}else{
     exprIfFalse;
}
```

```
condition ? exprIfTrue : exprIfFalse;
```

JS Introduction

Comments:

```
// comment   or    /* comment */
```

Loops

```
for (let i = 0; i < 5; i++) { … }
```

```
while (notFinished) { … }
```

Functions

```
function name(arg1, arg2) {
     statement;
     statement;
     return ...;
}
name(..,..);
```

```
const name = (arg1, arg2) => {
    statement;
    statement;
    return ...;
}
name(..,..);
```

JS Introduction

Arrays:

```
let x = [20,70];//option 1
let y = new Array(20,70);//option 2

//accessing values
console.log(x[0]);
console.log(y[1]);
```

Objects

```
let z = {name:"moayad", age:35};

//accessing values
console.log(z.name);
console.log(z.age);
```

JS Introduction

- We can use the prompt() method to request user's input.
- We can use the document.write() or document.writeln() methods to **add HTML** to a page using JavaScript.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 7.7: average.html -->
4  <!-- Counter-controlled repetition to calculate a class average.
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Class Average Program</title>
9        <script>
10
11          var total; // sum of grades
12          var gradeCounter; // number of grades entered
13          var grade; // grade typed by user (as a string)
14          var gradeValue; // grade value (converted to integer)
15          var average; // average of all grades
16
17          // initialization phase
18          total = 0; // clear total
19          gradeCounter = 1; // prepare to loop
20
```

ig. 7.7 | Counter-controlled repetition to calculate a class average. Part 1 of 4.)

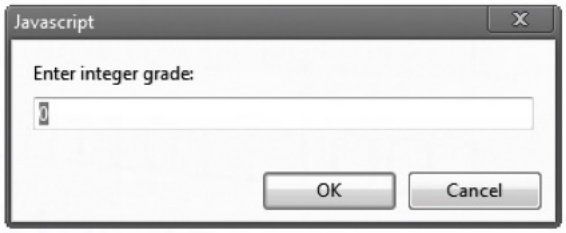
```
21          // processing phase
22          while ( gradeCounter <= 10 ) // loop 10 times
23          {
24
25             // prompt for input and read grade from user
26             grade = window.prompt( "Enter integer grade:", "0" );
27
28             // convert grade from a string to an integer
29             gradeValue = parseInt( grade );
30
31             // add gradeValue to total
32             total = total + gradeValue;
33
34             // add 1 to gradeCounter
35             gradeCounter = gradeCounter + 1;
36          } // end while
37
38          // termination phase
39          average = total / 10;    // calculate the average
40
41          // display average of exam grades
42          document.writeln(
43             "<h1>Class average is " + average + "</h1>" );
44
45       </script>
46    </head><body></body>
47 </html>
```

Fig. 7.7 | Counter-controlled repetition to calculate a class average. (Part 3 of 4.)

JS Introduction

- We can use the prompt() method to request user's input.
- We can use the document.write() or document.writeln() methods to **add HTML** to a page using JavaScript.

a) This dialog is displayed 10 times. User input is 100, 88, 93, 55, 68, 77, 83, 95, 73 and 62. User enters each grade and presses **OK**.



b) The class average is displayed in a web page

Fig. 7.7 | Counter-controlled repetition to calculate a class average. (Part 4 of 4.)

JS Introduction

- We can also **show messages** to a user by displaying text in an **alert Dialog** using the **alert() method**.
  - Useful to display information in windows that "pop up" on the screen to grab user's attention.
  - Typically used to display important messages to the user browsing the web page.
  - Browser's window object uses method alert to display an alert dialog.
  - Method alert requires as its argument the string to be displayed.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 6.3: welcome3.html -->
4  <!-- Alert dialog displaying multiple lines. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Printing Multiple Lines in a Dialog Box</title>
9        <script type = "text/javascript">
10          <!--
11          window.alert( "Welcome to\nJavaScript\nProgramming!" );
12          // -->
13       </script>
14    </head>
15    <body>
16       <p>Click Refresh (or Reload) to run this script again.</p>
17    </body>
18 </html>
```

Fig. 6.3 | Alert dialog displaying multiple lines. (Part 1 of 2.)

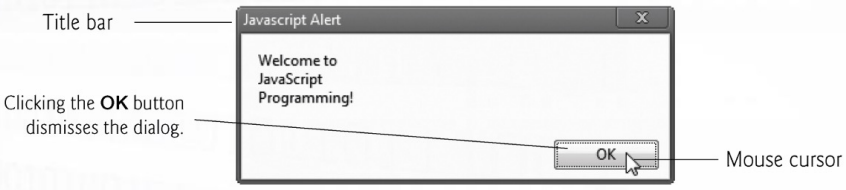


Fig. 6.3 | Alert dialog displaying multiple lines. (Part 2 of 2.)

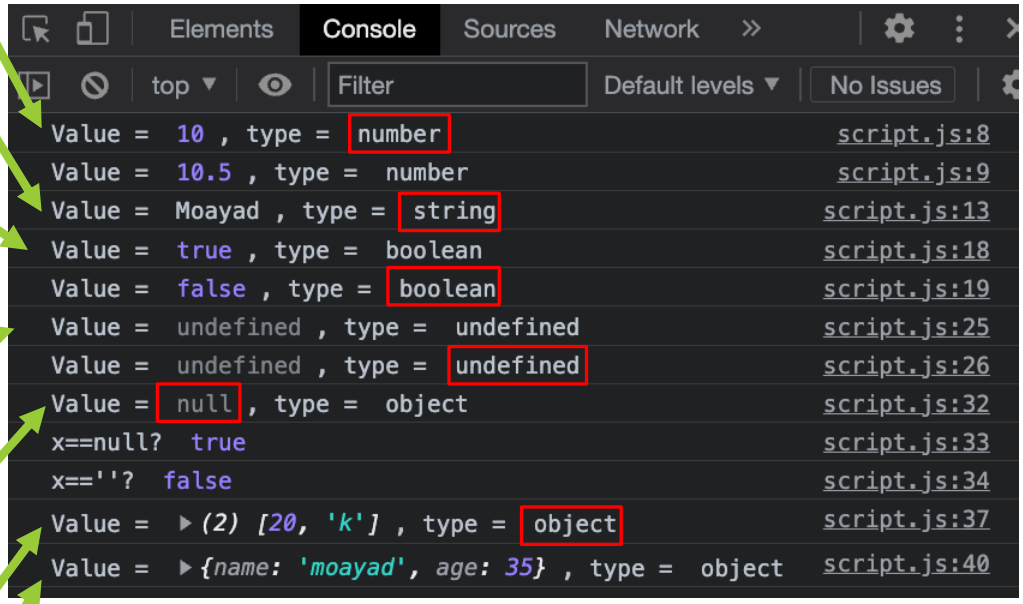
Variables and Scope

- JS variables do not have types, but the values do.
- There are multiple main JavaScript types, aka, **primitive types**:
  - **Number** : everything is a double (no integers).
  - **String**: in 'single' or "double-quotes".
  - **Boolean** : true or false.
  - **Null**: an "intentionally absent" value.
  - **Undefined**: the value of a variable with no value assigned.
  - **Object:** a collection of related data and/or functionality.

Variables and Scope

```
// numbers
let x = 10;//int
let y = 10.5;//float
console.log("Value = ",x,", type = " , typeof(x));
console.log("Value = ",y,", type = " , typeof(y));

//strings
x = "Moayad";
console.log("Value = ",x,", type = " , typeof(x));

//boolean
x = true;
y = false;
console.log("Value = ",x,", type = " , typeof(x));
console.log("Value = ",y,", type = " , typeof(y));


//undefined
x = undefined;
let z;
console.log("Value = ",x,", type = " , typeof(x));
console.log("Value = ",z,", type = " , typeof(z));

//objects
//null
x = null;
y = "";
console.log("Value = ",x,", type = " , typeof(x));
console.log("x==null? ",(x==null));
console.log("x==''? ",(y==null));
//array
x = [20,"k"];
console.log("Value = ",x,", type = " , typeof(x));
//object
z = {name:"moayad", age:35};
console.log("Value = ",z,", type = " , typeof(z));
```

Note that an empty string != null



Variables and Scope

- Several ways to define a variable.

```
let x;       //defines x, no initial value (undefined)
let y=10;    //defines and initializes y
const z=50; //defines a constant value of z (can't change)
```

- There is also the ‘var’ keyword. However, avoid using var. **Why?**

```
var x=10;    //defines and initializes x
```

- The difference between ’var’ and ‘let’ is all about the scope.
  - **var** creates a **function** scoped variable.
  - **let** creates a **block** scoped variable.

What does that mean?

مرتبي Variables and Scope

- **Variable scope** refers to the accessibility or visibility of a variable within a particular context in a JavaScript program. The scope determines where the variable can be accessed and modified.
- In JavaScript, there are two types of variable scopes: **global scope** and **local scope**.
- **Global Scope:** A variable declared **outside** of any function has global scope. It can be **accessed** from **any part of the program**, including any function, and can be modified accordingly.

default

Variables and Scope

- If a variable is declared **without** the var, let, or const keyword, it automatically becomes a **global variable**, even if it is declared inside a function.
- The **local scope** can be divided into **function scope**, **block scope**, and **lexical scope**.
- **Function Scope**: A variable declared inside a function has function scope. It can be accessed and modified only within the function where it is declared.

Variables and Scope

- **Block Scope**: A variable declared inside **a block statement** (within curly braces) has block scope. It can be accessed only within the block where it is declared.

by default global

```
//Global Scope
function setGlobalVar() {
  globalVar = 30; // globalVar is declared without the 'var' keyword
}
setGlobalVar();
console.log(globalVar); // Output: 30


//Function Scope
function addNumbers() {
  var num1 = 5; // num1 has function scope
  var num2 = 10; // num2 has function scope
  var sum = num1 + num2;
  console.log(sum);
}
addNumbers(); // Output: 15
console.log(num1); // Output: ReferenceError: num1 is not defined

//Block Scope
if (true) {
  let x = 10; // x has block scope
  console.log(x); // Output: 10
}
console.log(x); // Output: ReferenceError: x is not defined
```

Variables and Scope

- scope difference var vs. let and const:

can be access outside the {}

```
var x = 10;
if (x > 0) {
  var i = 10;//accessible outside
}
console.log('Value of i is ' + i);
```

Output:

```
Value of i is 10
>
```

```
let x = 10;
if (x > 0) {
 let i = 10;//accessible only here
}
console.log('Value of i is ' + i);
```

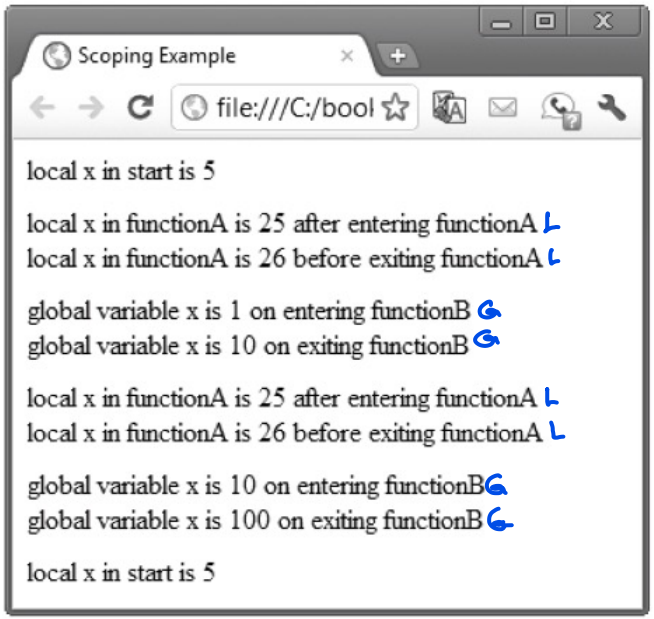
Output:

```
Uncaught ReferenceError: i is not defined
    at script.js:58:47
>
```

- Variables declared with "var" do not go out of scope at the end of blocks; only at the end of functions.
- Variables declared with "let", "const" have block-scope, so accessing the variable outside the block results in an error.

Variables and Scope

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.9: scoping.html -->
4  <!-- Scoping example. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Scoping Example</title>
9        <style type = "text/css">
10          p       { margin: 0px; }
11          p.space { margin-top: 10px; }
12       </style>
13
14       <script>
15          var output; // stores the string to display
16          var x = 1; // global variable
17
18          function start()
19          {
20             var x = 5; // variable local to function start
21
22             output = "<p>local x in start is " + x + "</p>";
23
24             functionA(); // functionA has local x
25             functionB(); // functionB uses global variable x
26             functionA(); // functionA reinitializes local x
27             functionB(); // global variable x retains its value
28
29             output += "<p class='space'>local x in start is " + x +
30                "</p>";
31             document.getElementById( "results" ).innerHTML = output;
32          } // end function start
33
34          function functionA()
35          {
36             var x = 25; // initialized each time functionA is called
37
38             output += "<p class='space'>local x in functionA is " + x +
39                " after entering functionA</p>";
40             ++x;
41             output += "<p>local x in functionA is " + x +
42                " before exiting functionA</p>";
43          } // end functionA
44
45          function functionB()
46          {
47             output += "<p class='space'>global variable x is " + x +
48                " on entering functionB";
49             x *= 10;
50             output += "<p>global variable x is " + x +
51                " on exiting functionB</p>";
52          } // end functionB
53
54          window.addEventListener( "load", start, false );
55       </script>
56    </head>
57
58    <body>
59       <div id = "results"></div>
60    </body>
61 </html>
```

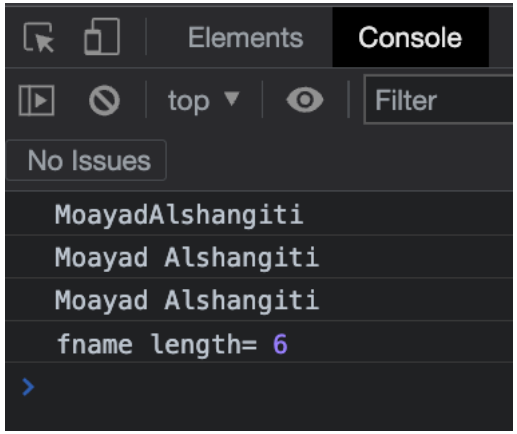


Datatypes and Operators

Working with string.

- Can be defined with single or double quotes.
- Can use plus for string concatenation.
- Can check size via length property (not function)

```
let fname = "Moayad";
let lname = "Alshangiti";

console.log(fname+lname);
console.log(fname+" "+lname);
console.log(fname,lname);
console.log("fname length=", fname.length);
```



Datatypes and Operators

Working with string.

- **Escape Sequences**: When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence. The escape sequence \n is the newline character. It causes the cursor in the HTML5 document to move to the beginning of the next line.

Escape sequence	Description
`\n`	*New line*—position the screen cursor at the beginning of the next line.
`\t`	*Horizontal tab*—move the screen cursor to the next tab stop.
`\\`	*Backslash*—used to represent a backslash character in a string.
`\"`	*Double quote*—used to represent a double-quote character in a string contained in double quotes. For example, `window.alert( "\"in double quotes\"" );` displays `"in double quotes"` in an `alert` dialog.
`\'`	*Single quote*—used to represent a single-quote character in a string. For example, `window.alert( '\'in single quotes\'' );` displays `'in single quotes'` in an `alert` dialog.

Fig. 6.4 | Some common escape sequences.

Datatypes and Operators

Working with number.

- All numbers are of type double.
- A few special values: NaN (not-a-number), +Infinity, -Infinity.
- The Math class can be handy.

Math Constant examples	
`Math.E`	`Math.PI`
`Math.SQRT2`	`Math.LOG10E`

Math method examples	
`max(a, b)`	`min(a, b)`
`random()`	`round(x)`
`sqrt(x)`	`pow(a,b)`

Datatypes and Operators

Working with boolean.

- There are two literal values for boolean: true and false that behave as you would expect.
- We can use the usual Boolean operators (&&,||,!, etc).

```
if(fname=="Moayad" && lname=="Alshangiti"){…}
```

- Non-boolean values can be used in control statements, which get converted to their "truthy" or "falsy" value:
  - null, undefined, 0, NaN, '', "" will evaluate to false.
  - Everything else will evaluate to true.

Datatypes and Operators

Working with boolean.

- Be aware that JS equality operators do implicit type conversion before checks:

```
'' == '0'  // false
'' == 0  // true
0 == '0'  // true
NaN == NaN  // false
[''] == ''  // true
false == undefined  // false
false == null  // false
null == undefined  // true
```

How can we fix this issue?

Datatypes and Operators

Working with boolean.

- Instead of fixing == and != , the ECMAScript standard kept the existing behavior but added === and !==

This better

```
'' === '0'  // false
'' === 0  // false
0 === '0'  // false
NaN == NaN  // still weirdly false
[''] === ''  // false
false === undefined  // false
false === null  // false
null === undefined  // false
```

Datatypes and Operators

Working with null vs. undefined.

- null is a value representing the absence of a value, similar to null in Java and nullptr in C++
- undefined is the value given to a variable that has not been a value.

```
let x = null;
let y;
console.log(x);
console.log(y);
```

```
null
undefined
>
```

Datatypes and Operators

- Working with operators.

Standard algebraic equality operator or relational operator	JavaScript equality or relational operator	Sample JavaScript condition	Meaning of JavaScript condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Fig. 6.13 | Equality and relational operators.

Datatypes and Operators

- Working with operators.

Operator	Associativity	Type
++ -- !	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== != === !==	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 8.16 | Precedence and associativity of the operators discussed so far.

Datatypes and Operators

- Working with operators.

Assignment operator	Initial value of variable	Sample expression	Explanation	Assigns
+=	c = 3	c += 7	c = c + 7	10 to c
-=	d = 5	d -= 4	d = d - 4	1 to d
*=	e = 4	e *= 5	e = e * 5	20 to e
/=	f = 6	f /= 3	f = f / 3	2 to f
%=	g = 12	g %= 9	g = g % 9	3 to g

Fig. 7.12 | Arithmetic assignment operators.

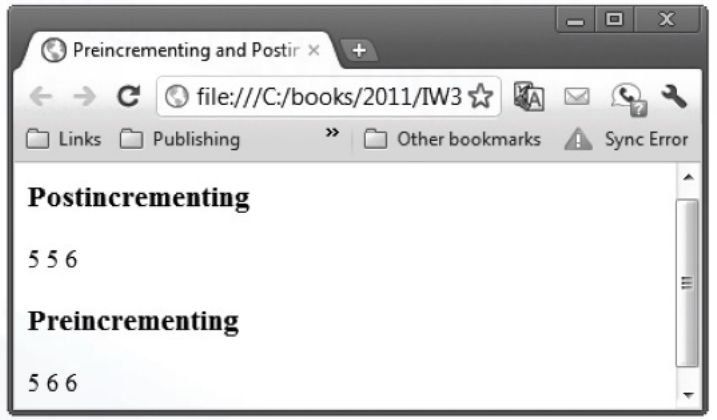
Operator	Example	Called	Explanation
++	++a	preincrement	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	postincrement	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	predecrement	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	postdecrement	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 7.13 | Increment and decrement operators.

Datatypes and Operators

- Working with operators.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 7.14: increment.html -->
4  <!-- Preincrementing and Postincrementing. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Preincrementing and Postincrementing</title>
9        <script>
10
11          var c;
12
13          c = 5;
14          document.writeln( "<h3>Postincrementing</h3>" );
15          document.writeln( "<p>" + c ); // prints 5
16          // prints 5 then increments
17          document.writeln( " " + c++ );
18          document.writeln( " " + c + "</p>" ); // prints 6
19
20          c = 5;
21          document.writeln( "<h3>Preincrementing</h3>" );
22          document.writeln( "<p>" + c ); // prints 5
23          // increments then prints 6
24          document.writeln( " " + ++c );
25          document.writeln( " " + c + "</p>" ); // prints 6
26
27       </script>
28    </head><body></body>
29 </html>
```



Good Programming Practice 7.5

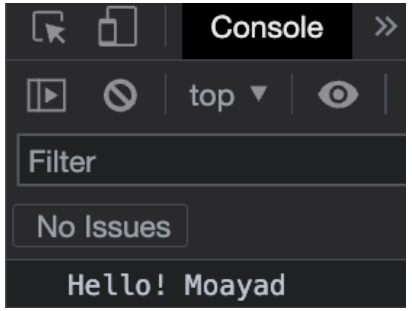
For readability, unary operators should be placed next to their operands, with no intervening spaces.

Functions and Objects

- **Functions** are blocks of code that can be defined and then called repeatedly throughout your code. They can take parameters as input and can return values as output. Functions are used to organize code, reduce repetition, and make code more modular.

- Functions can be **defined** in several ways:
  - **Function Declaration**: This is the most common way to define a function. It involves using the "function" keyword followed by the name of the function, followed by parentheses containing any parameters the function may have, and finally the code block containing the function's code.

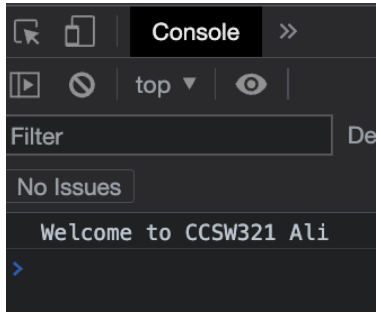
```
function hello(name) {
  console.log('Hello!',name);
}
//invoke
hello('Moayad');
```



Functions and Objects

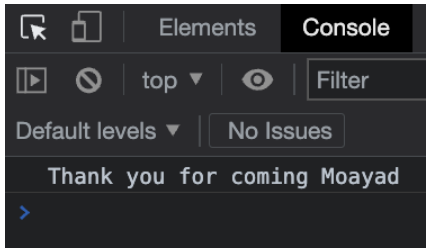
- **Functions** can be **defined** in several ways:
  - **Function Expression**: This involves assigning a function to a variable. This is useful when you want to pass a function as an argument to another function.

```
const welcome = function(name){
    console.log('Welcome to CCSW321', name);
} //invoke
welcome('Ali');
```



  - **Arrow Function**: This is a newer syntax introduced in ES6. It is a shorthand way of writing a function that does not have its own "this" keyword.

```
const bye = (name) => {
  console.log('Thank you for coming', name)
}//invoke
bye(Moayad');
```



Functions and Objects

- In addition to user-defined functions, JavaScript offer many useful global functions that we can use:
  - **alert():** Displays an alert box with a message and an OK button.
  - **confirm():** Displays a dialog box with a message and OK and Cancel buttons.
  - **parseInt():** Parses a string and returns an integer.
  - **parseFloat():** Parses a string and returns a floating-point number.
  - **isNaN():** Determines whether a value is NaN (Not a Number).
  - **setTimeout():** Calls a function or executes a code snippet after a specified delay.
  - **setInterval():** Calls a function or executes a code snippet repeatedly at a specified interval.

Functions and Objects

- **Objects** are used to represent complex data structures consisting of properties and values in JavaScript. An object can have any number of **properties**, each of which has a **name** and a **value**.
- In JavaScript, objects are created using either the **object literal notation** or the **constructor notation**.
- **The object literal notation** is a shorthand way of creating an object by enclosing a comma-separated list of property-value pairs within curly braces { }.

```
let person = {
  name: 'John',
  age: 30,
  occupation: 'Web Developer'
};
```

object with three properties: name, age, and occupation.

Functions and Objects

- **The constructor notation** is a way of creating an object using a constructor function. A constructor function is a special type of function that is used to create objects with specific properties and methods

```
function Person(name, age, occupation)
{
  this.name = name;
  this.age = age;
  this.occupation = occupation;
}

let person1 = new Person('John', 30,
'Web Developer’);

let person2 = new Person('Jane', 25,
'Graphic Designer');
```

Person is a **constructor function** that takes three parameters: name, age, and occupation.

The **new keyword** is used to create a new instance of the Person object.

Functions and Objects

- **To access** the properties of an object, we can use the **dot** or **bracket notation**:

```
console.log(person.name); // Output: John
console.log(person['age']); // Output: 30
```

- **Objects can also have methods, which are functions that are defined as properties of an object.**

```
let person = {
  name: 'John',
  age: 30,
  occupation: 'Web Developer',
  sayHello: function() {
    console.log('Hello, my name is ' + this.name);
  }
};

person.sayHello(); // Output: Hello, my name is John
```

Functions and Objects

- You can also use the **this keyword** to refer to the current object within a method.

```
let person = {
  name: 'John',
  age: 30,
  occupation: 'Web Developer',
  sayHello: function() {
    console.log('Hello, my name is ' + this.name);
  },
  getAge: function() {
    return this.age;
  },
  setAge: function(newAge) {
    this.age = newAge;
  }
};

console.log(person.getAge()); // Output: 30
person.setAge(35);
console.log(person.getAge()); // Output: 35
```

Arrays

- **Arrays** are Object types used to create lists of data.
- 0-based indexing.
- Can check size via length property (not function)

```
// creating an array
Let emptylist = []; //empty
let mylist = [1,2,3]; //has 3 elements.
// elements can be of different types
let mixed = ['milk',10,3,'cookie’]; //has 4 elements
// accessing elements with index
console.log(mixed[0]); //will print milk
```

Arrays

- Iterating over elements:
  - **groceries.length:** returns how many elements are in the array.

Iterate using **for loop**:

```
let groceries = ['milk', 'cocoa puffs', 'tea'];

for (let i = 0; i < groceries.length; i++) {

console.log(groceries[i]);}
```

Iterate use **.forEach**:

```
let groceries = ['milk', 'cocoa puffs', 'tea'];

groceries.forEach(function(element) {{

console.log(element);}
```

Arrays

- Useful methods to know:

```
// sort method sorts arrays
   groceries.sort();
```
▶ (3) ['cocoa puffs', 'milk', 'tea']

```
// join method joins all array elements into a string
   groceries.join(",");
```
cocoa puffs,milk,tea

```
// push method adds a new element to array at the end
   groceries.push('apple');
```
▶ (4) ['cocoa puffs', 'milk', 'tea', 'apple']

```
// pop method removes last element from array
   groceries.pop();
```
▶ (3) ['cocoa puffs', 'milk', 'tea']

```
// concat method creates a new array by merging existing arrays
   let sports = ["football", "basketball"];
   groceries.concat(sports);
```
at end

▶ (5) ['cocoa puffs', 'milk', 'tea', 'football', 'basketball']

Arrays

- Useful methods to know:

```
// Get index of value in groceries (-1 if not found):
  groceries.indexOf('milk’); //will return 1
// Return a subarray (also works for strings)
  groceries.slice(1,2);
// Replaces 1 element at index 0
  groceries.splice(0,1, 'pepsi’);
```

ما يطبع tea بس بوقف عند 2

▶ ['milk']

▶ (3) ['pepsi', 'milk', 'tea']

if 0,2 → pepsi , pepsi , tea

Software Engineering Observation 10.2

JavaScript automatically reallocates an array when a value is assigned to an element that’s outside the bounds of the array. Elements between the last element of the original array and the new element are `undefined`.

Exception Handling

- **Exception handling** is a mechanism that allows you to handle errors or unexpected situations in your code gracefully, rather than letting the program crash or produce incorrect results. In JavaScript, you can use a try-catch statement to handle exceptions.

- Here is **how the try-catch statement works**:
  - The code that you want to monitor for exceptions is placed inside a try block.
  - If an exception occurs inside the try block, the code execution is immediately transferred to the catch block.
  - The catch block contains code that specifies what to do in case of an exception. It takes one parameter, which is the exception object containing information about the error.

Exception Handling

- The try/catch block:
  - The **try block** contains code that throws an exception using the throw statement. The **catch block** contains code that logs the error object to the console.
  - There are different types of errors in JavaScript, and you can use different types of catch blocks to handle them.

```
try {
  // code that might throw an exception
  throw new Error("Something went wrong");
} catch (error) {
  // code to handle the exception
  console.error(error);
}
```

JavaScript Example

Can you create this simple app?

Step1. A JavaScript code creates an **array** of student **objects** with their names, grades, and date of birth.

- "Moayad",[85, 90, 75], "2001-05-20”
- "Ali",[70, 80, 90], "2002-03-15”
- "Osama",[80, 75, 85], "2000-11-10"

Step2. The **calculateGPA** function takes in a student object and calculates their GPA based on their grades

Step3. The for **loop** at the bottom prints out each **student's information** and their **calculated GPA**.

JavaScript Example

Step1

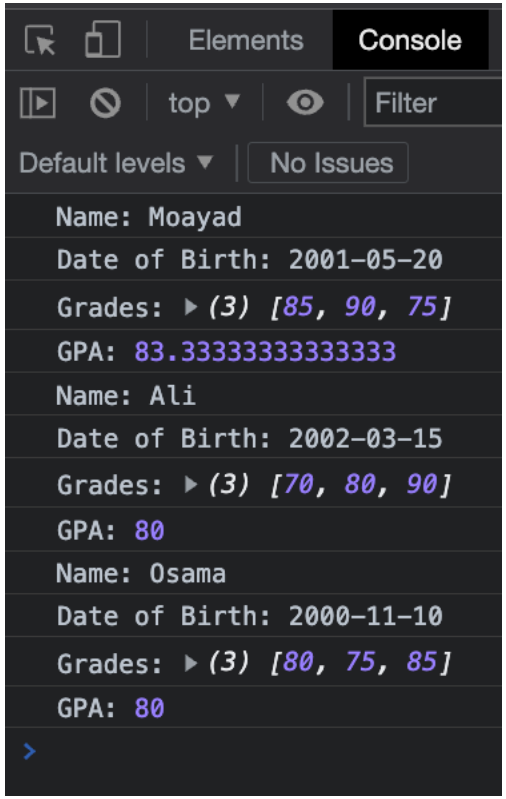
```
// Create the object template
function student(name,grades,dob){
    this.name = name;
    this.grades = grades;
    this.dob = dob;
}

// Create an array of student objects
let students = [new student("Moayad",[85, 90, 75],  "2001-05-20"),
                new student("Ali",[70, 80, 90],  "2002-03-15"),
                new student("Osama",[80, 75, 85],  "2000-11-10")];
```

Step2

```
// Calculate the GPA of each student
function calculateGPA(student) {
  let total = 0;
  for (let i = 0; i < student.grades.length; i++) {
    total += student.grades[i];
  }
  let gpa = total / student.grades.length;
  return gpa;
}
```

Step3

```
// Print out each student's information and GPA
for (let i = 0; i < students.length; i++) {
  console.log('Name:', students[i].name)
  console.log('Date of Birth:', students[i].dob)
  console.log('Grades:', students[i].grades)
  console.log('GPA:', calculateGPA(students[i]))
}
```



Any questions?
Please feel free to raise your hands and ask.