Software Studio

軟體設計與實驗

JavaScript – Part I

Hung-Kuo Chu

Department of Computer Science

National Tsing Hua University

CS2410

What is JavaScript?

- A high-level, interpreted programming language that enables you to create **dynamically updating content** and **user interaction**
  - control multimedia
  - animate images
  - etc.
- Client-side JavaScript
  - Interact with web application
- Server-side JavaScript
  - Communicate with database
- **JavaScript and Java are distinct and differ**

What is JavaScript? (Cont'd)

- JavaScript implements **ECMAScript (ES)** standardization
  - ES5 (2009)
  - **ES6 (2015)**
  - ES7, ES8 …
- Lots of useful frameworks and libraries
  - jQuery
  - Firebase, Node.js
  - **WebGL**

Why Study JavaScript?

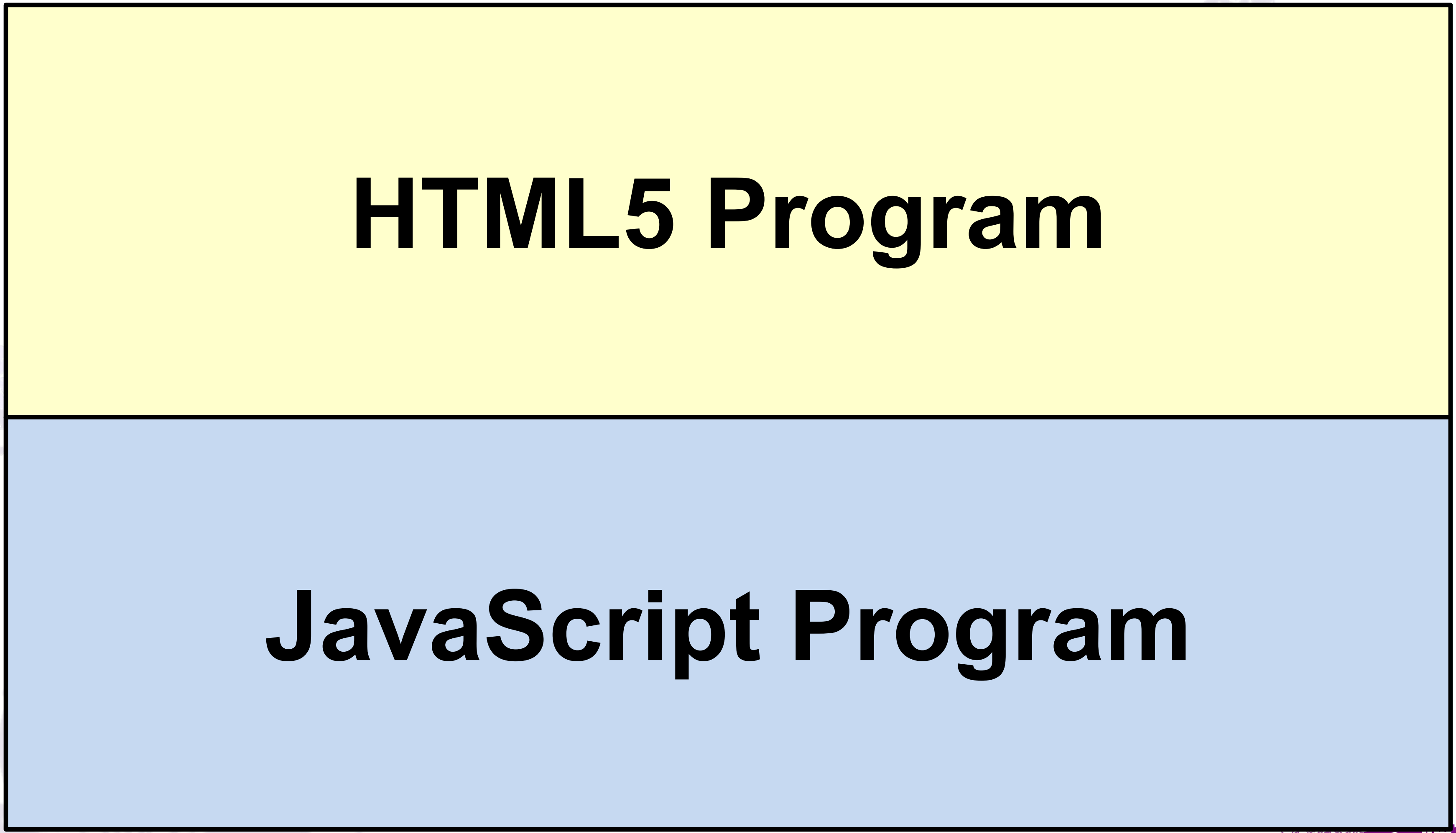
- JavaScript is one of the 3 languages all web developers must learn:
  - – 1. HTML to define the content of web pages
  - – 2. CSS to specify the layout of web pages
  - – 3. JavaScript to program the behavior of web pages

HTML
HTML + CSS
@TopDevVietNam
HTML + CSS + JAVASCRIPT

Why Use JavaScript ?

- All modern web browsers support JavaScript without the need for plug-ins by means of a built-in JavaScript engine
- In other words, JavaScript is a **cross-platform** programming language that runs on all machines with browser software

Codeblock Conventions



HTML5 Program

JavaScript Program

JavaScript(JS): Basics

- Every statement **ends with a semicolon**, although none is required by JS.
- Unlike HTML5, JS is **case sensitive**, so you should use the proper uppercase and lowercase letters when coding.
- JavaScript statements can be grouped together in **code blocks**, inside curly brackets {...} (Life-time scope).

JavaScript(JS): Usage

- Use the **<script></script>** tag in HTML and write JS within the tag's scope, and the browser will execute the code

```
<script>
    document.write("Hello World!!");
</script>
```

JavaScript(JS): Usage

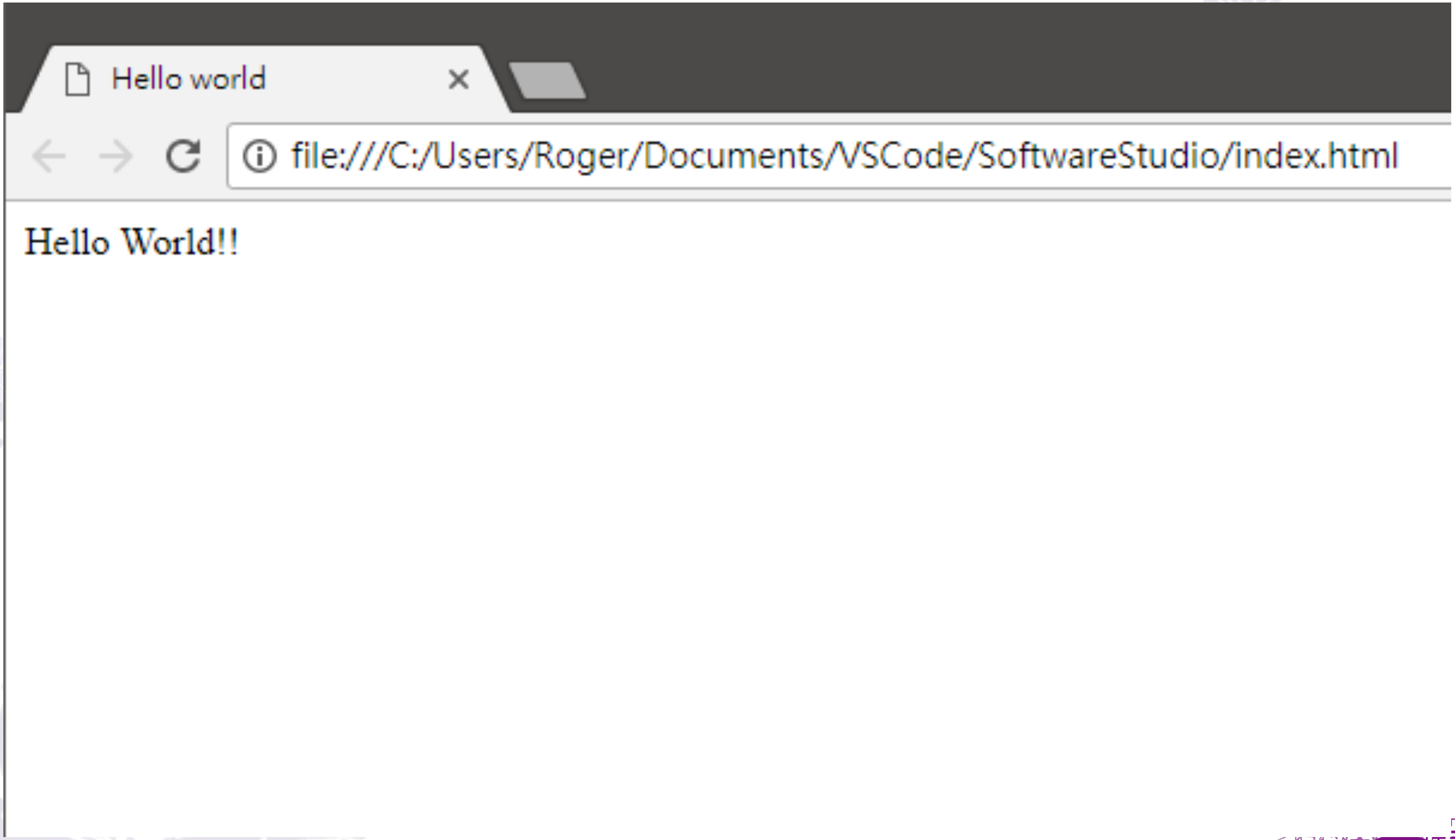
- JS code usually appears in the **<head>** section, but it can be placed **anywhere** in a HTML document.
- When using JS inside the **<body>** section, **ALWAYS** place scripts at the bottom of the <body> element.
  - script interpretation slows down the display!
- Web browser will execute the scripts sequentially, from **top to bottom**.

Hello World

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Hello world</title>

    <script>
        document.write("Hello World!!");
    </script>
</head>
<body></body>
</html>
```

Hello World



file:///C:/Users/Roger/Documents/VSCode/SoftwareStudio/index.html

```
Hello World!!
```

Link the JS File

- You can save the JS codes to a **.js** file, and then add the path to the web page

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

Modified Hello World

```
document.write("Hello World!!");
```

app.js

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Hello world</title>

    <script src = "app.js"></script>
</head>
<body></body>
</html>
```

Comments in JS

- Use “//” to comment a single line
- User “/* … */” to comment a code block

Same as C/C++!

JavaScript(JS): Debugging

- The **console.log**() method
  - – Browser -> F12 -> Console tab.
- The **alert**() method

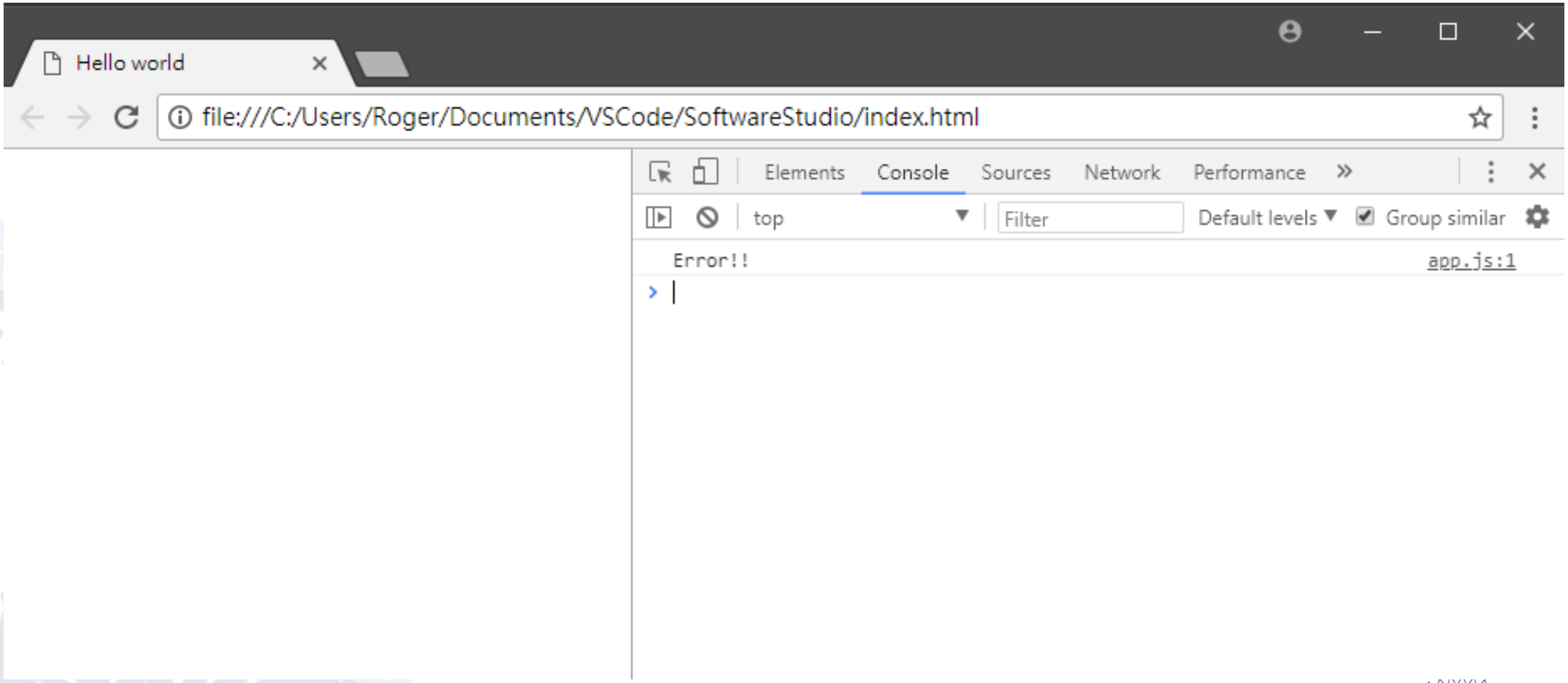
```
alert("Error!!");
```

- The “**debugger**” Keyword
  - – Setting a breakpoint in the debugger

```
var x = 15 * 5;
debugger;
document.getElementById("demo").innerHTML = x;
```

Debugging: console.log()

```
console.log("Error!!")
```

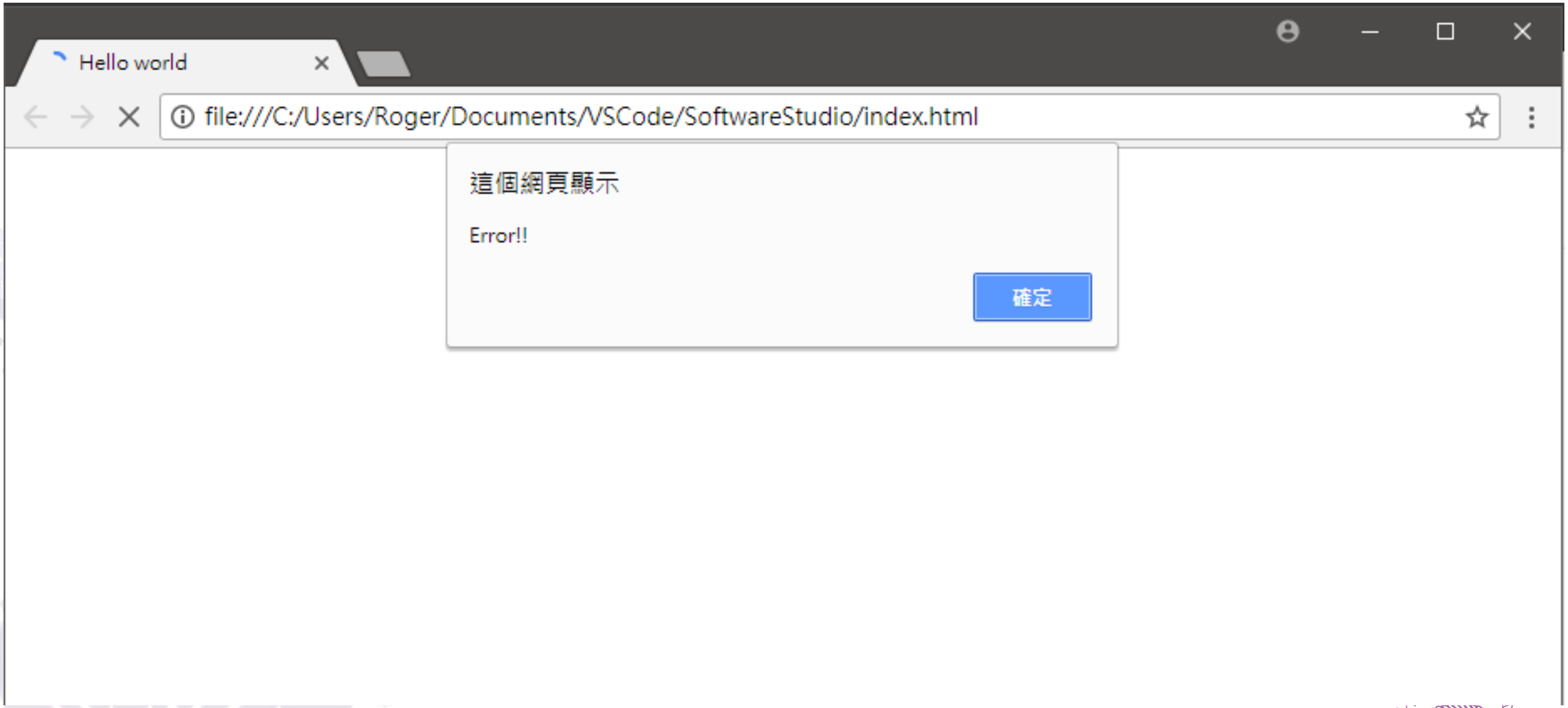


file:///C:/Users/Roger/Documents/VSCode/SoftwareStudio/index.html

```
Error!!                                                   app.js:1
>
```

Debugging: alert()

```
alert("Error!!");
```



Hello world

file:///C:/Users/Roger/Documents/VSCode/SoftwareStudio/index.html

這個網頁顯示

Error!!

Debugging: Try Catch

- The **try** statement allows you to define a block of code for debugging during the run time
- The **catch** statement allows you to define a block of code to catch the error occurred in the try block
- The **finally** statement lets you execute code, after try and catch, regardless of the result

Debugging: Try Catch (Cont’d)

```
try {
    // do something that may go wrong
}
catch (e) {
    // handle the error
}
finally {
    // finally do something
}
```

Take a Break!

Variable

- JS uses three keywords to declare variables: **var, let, const**
- If a variable declared without using any of the keywords, it will be regarded as a **global** variable

‘var’ Variable

- If a var variable is declared **within** the function, its life scope is bounded by the function.
- If a var variable is declared **outside** the function, it is a global variable.
- You can use a variable before declaring it!

```
console.log(a)                     // output: undefined

var a = 3;
console.log(a)                     // output: 3
```

‘let’ Variable

- Not supported before ES6
- **Block-scoped local** variable

let: Examples

```
let x = 1;

if (x === 1) {
  let x = 2;

  console.log(x); // expected output: 2
}

console.log(x);  // expected output: 1
```

‘const’ Variables

- Not supported before ES6
- Need to be initialized and cannot be changed afterward.
- **Block-scoped local** variable, like let.

const: Examples

```
const number = 42;

try {
  number = 99;
} catch(err) {
  console.log(err);
// output: TypeError: invalid assignment to const `number'
// Note: error messages will vary depending on browser
}

console.log(number);
// output: 42
```

Comparisons: var , let, const

	var	let	const
Declaration	global-scoped or function-scoped	block-scoped	block-scoped
Updated?	✓	✓	×
Redeclared?	✓	×	×

Ref: https://dev.to/sarah_chima/var-let-and-const--whats-the-difference-69e

Variable Type

- Basic types of JS variables:
  - Primitive Data
    - String
    - Number
    - Boolean
    - Undefined (when the variable is not assigned any value)
  - Complex Data
    - Object (include Array and Null)
    - Function (can be treated as an object)

Variable Type

- JS automatically determines the variable type according to the assigned value.
- JS automatically converts the variable types when performing numeric operations.

```
var a = 4;
var b = true;
var c = 'A';

console.log(a + b);                 // output: 5
console.log(a + c);                 // output: 4A
```

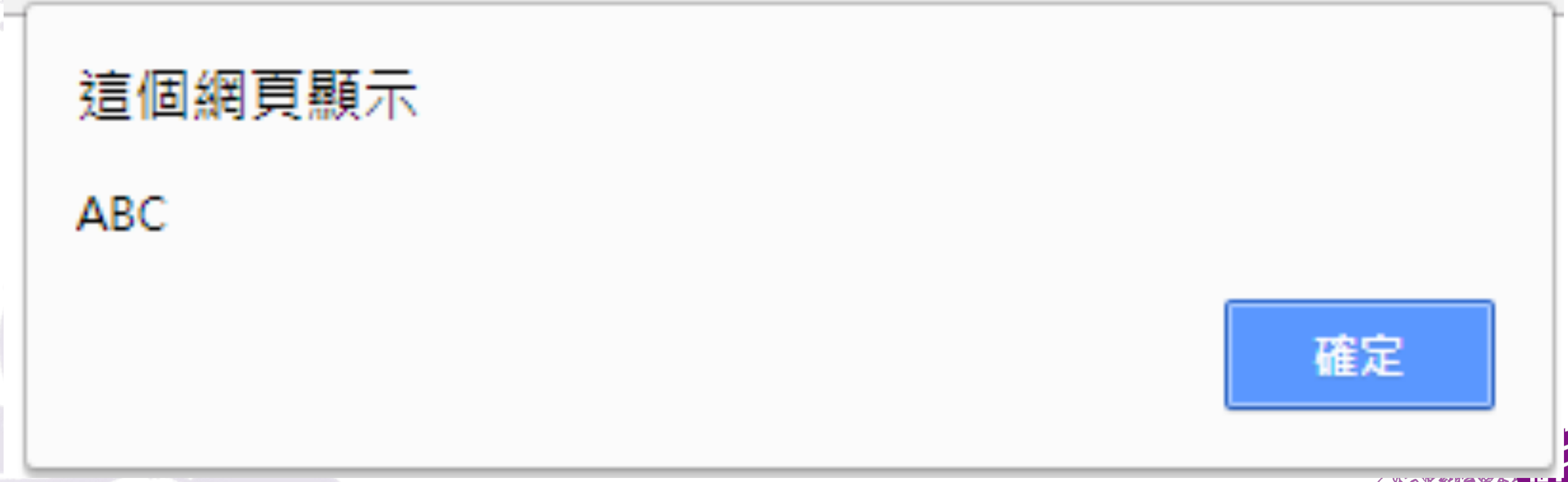
String Type

- Characters enclosed by single (‘) /double (") quotation
- JS uses UTF-16 as default encoding
- Use (+) operator to concatenate strings
- Use ([]) operator to access individual elements

String: Examples

```
var a = "Apple";
var b = 'Banana';
var c = 'Coconut';

alert(a[0] + b[0] + c[0]);
```



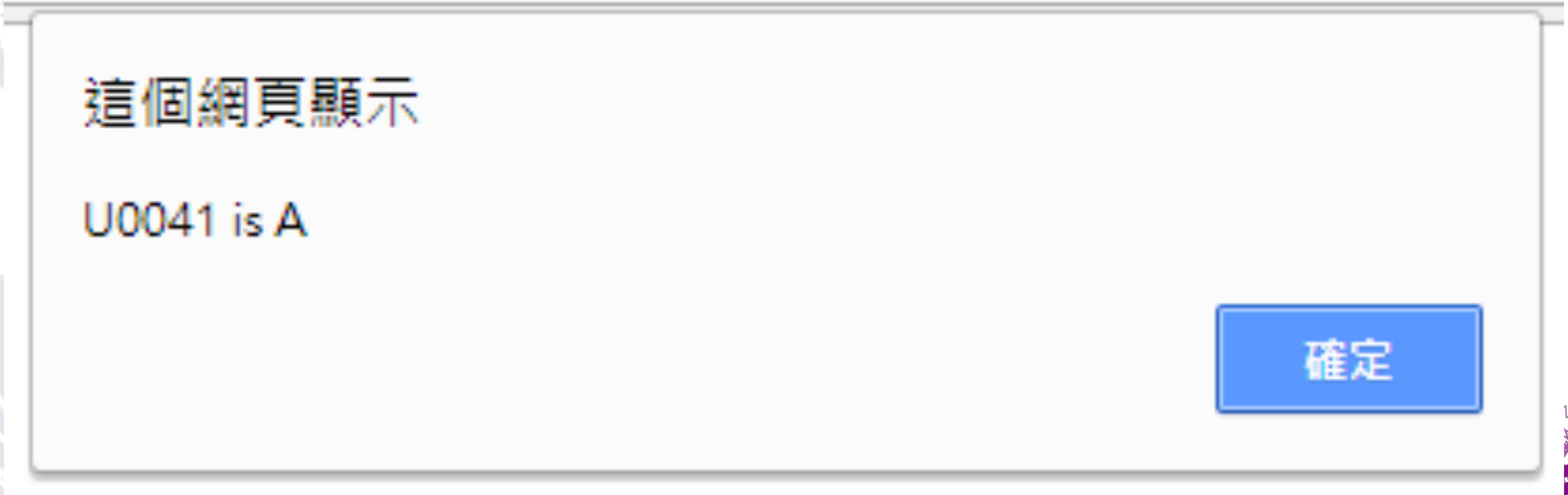
這個網頁顯示

ABC

Unicode in String

- Represent a character using its corresponding Unicode
- Syntax: \u[Unicode]

```
alert("U0041 is \u0041");
```



這個網頁顯示

U0041 is A

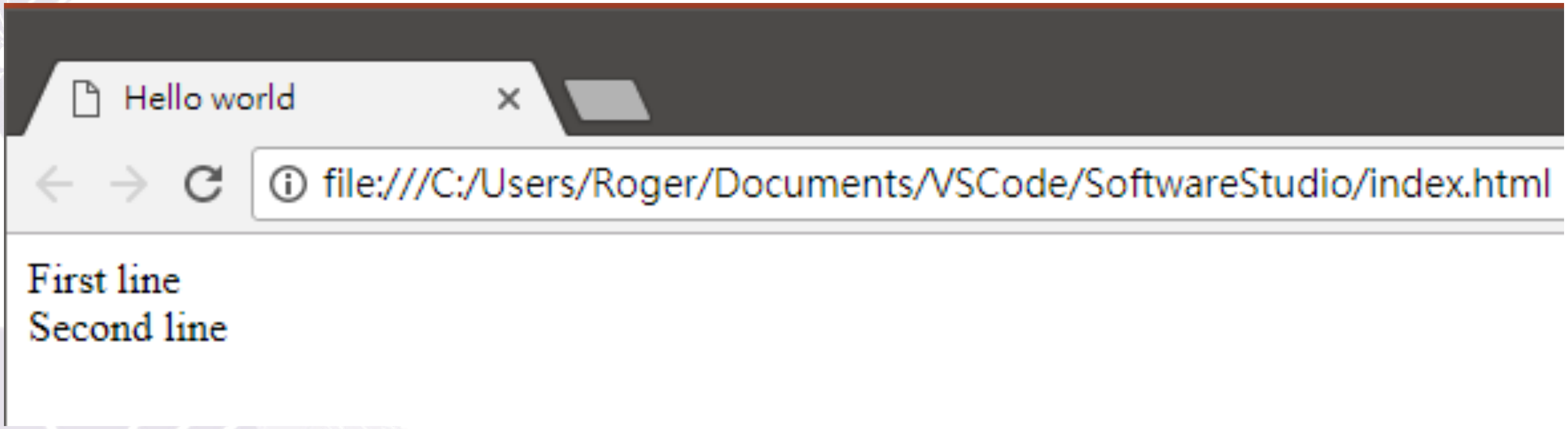
Ref: Unicode codepoint lookup/search tool

HTML Tag in String

- Compatible to HTML element!

HTML line break element

```
document.write("First line<br>Second line");
```


Hello world
file:///C:/Users/Roger/Documents/VSCode/SoftwareStudio/index.html

First line
Second line

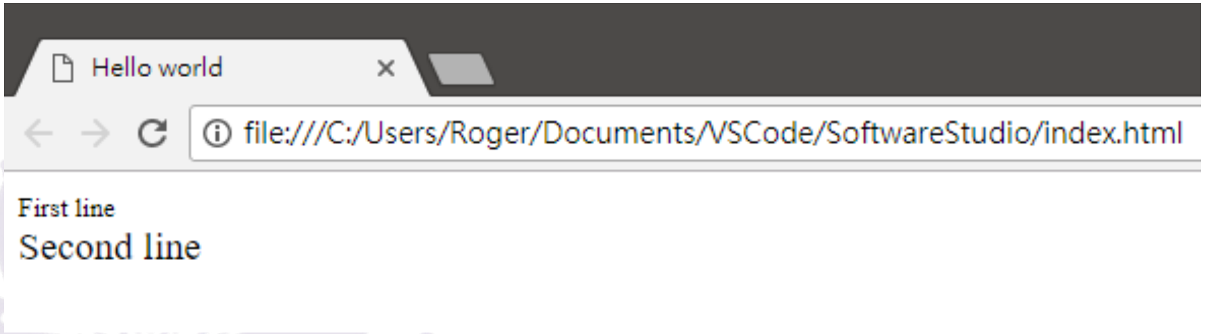
JS HTML Wrapper Functions

- JS HTML Wrapper functions can wrapper up the string parameters for HTML context.

```
var firstLine = "First line";
var secondLine = "Second line";
document.write(firstLine.small() + "<br>" +
                    secondLine.big());
```

small font size

big font size


Hello world
file:///C:/Users/Roger/Documents/VSCode/SoftwareStudio/index.html

```
First line
Second line
```

More examples!

Ref: JS HTML Wrapper Functions

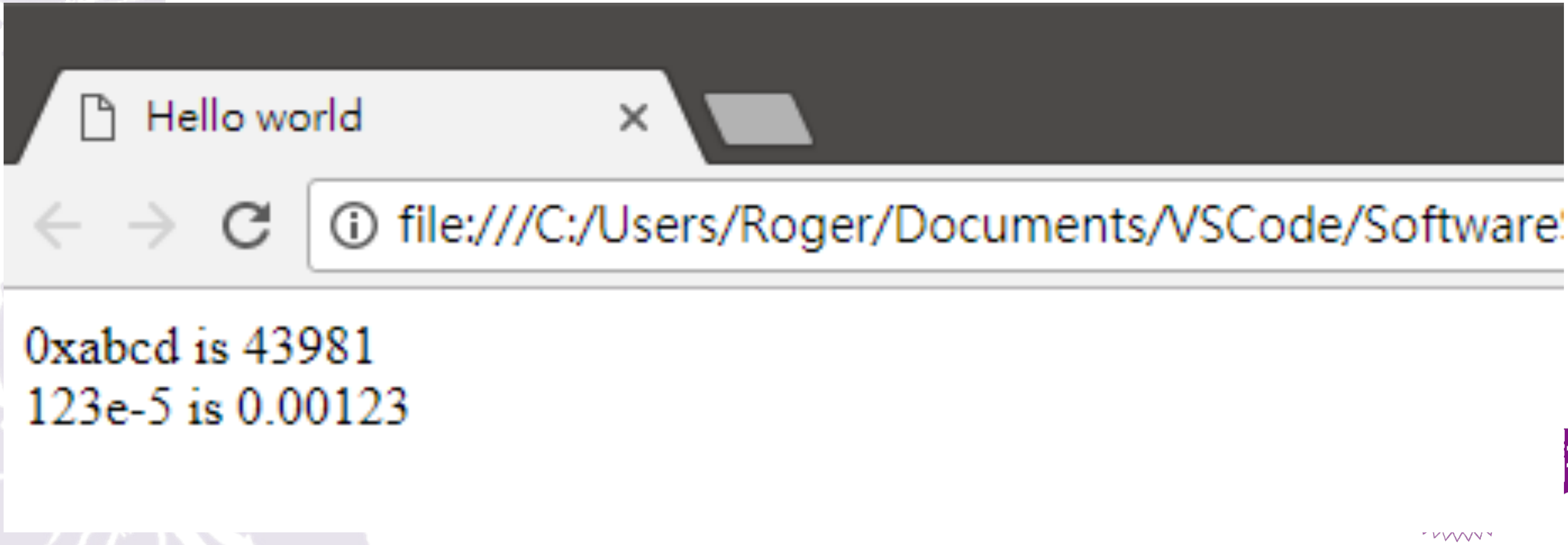
Number Type

- All numbers are stored in **floating-point** format
- You can use a decimal, hexadecimal, or scientific notation to indicate a number

Number: Examples

```
var numHex = 0xabcd;        // hex format
var numExp = 123e-5;        // scientific format

document.write("0xabcd is " + numHex + "<br>");
document.write("123e-5 is " + numExp);
```



```
0xabcd is 43981
123e-5 is 0.00123
```

Boolean Type

- Boolean represents two kinds of values: **true** or **false**
- Each data type can be converted to a boolean value:
  - **false**: 0, -0, null, false, NaN, undefined, empty string ("")
  - **true**: else

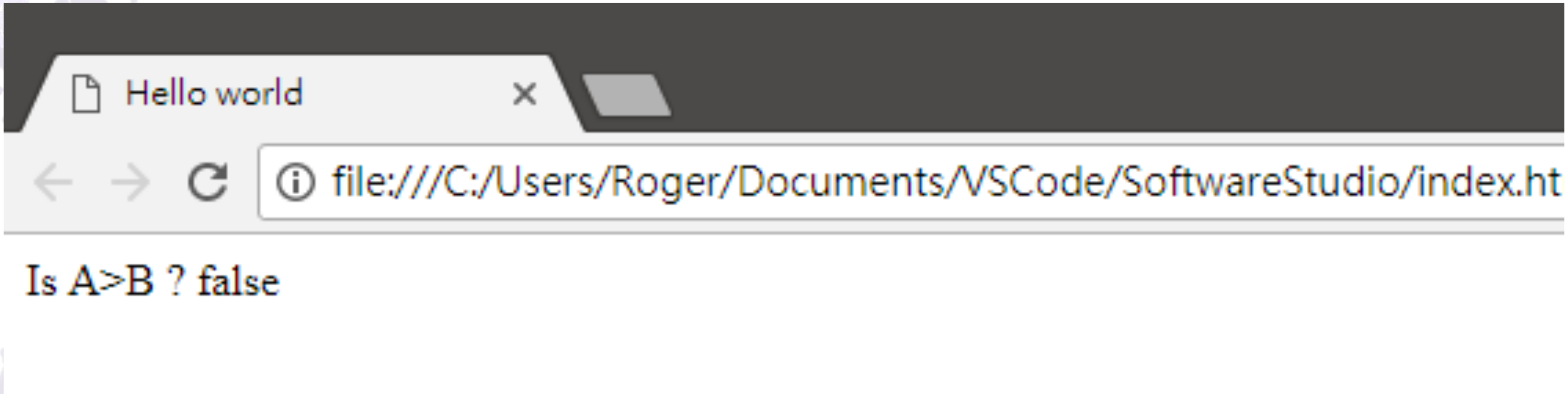
Comparison

- Compare operation always returns a boolean value
- JS uses both **strict** and **type-converting** comparisons
- Strings are compared based on standard lexicographical ordering, using Unicode values

Comparison

```
var a = 'A’; // 0041
var b = 'B’; // 0042

document.write("Is A>B ?\n" + (a > b));
```


Hello world
file:///C:/Users/Roger/Documents/VSCode/SoftwareStudio/index.ht

```
Is A>B ? false
```

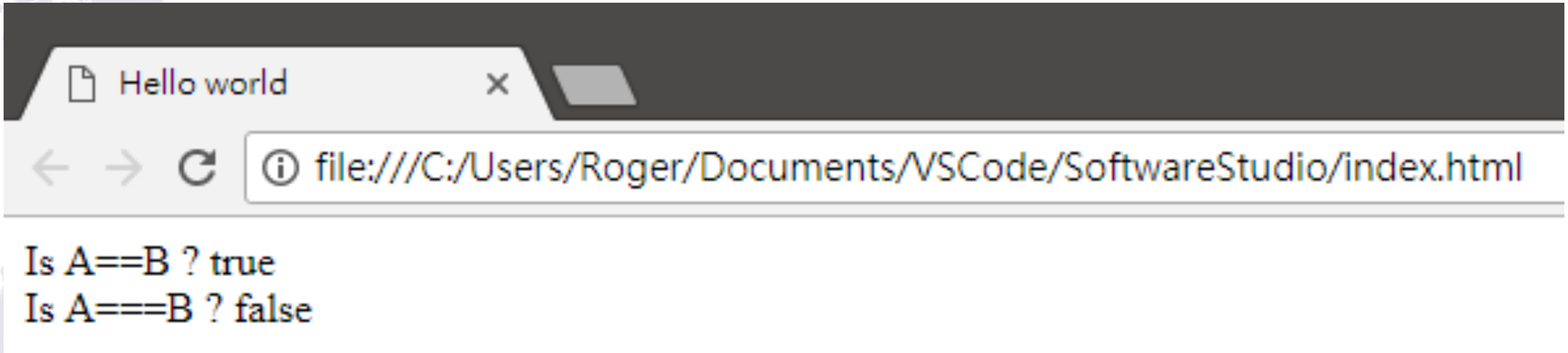
Strict Comparison

- A **strict** comparison (e.g., ===) is only true if the operands are of the **same type** and the **contents match**
- An **abstract** comparison (e.g., ==) converts the operands to the same type before making the comparison
- Reference (MDN)

Strict Comparison

```
var a = '5';
var b = 5;

document.write("Is A==B ?\n" + (a == b) + "<br>");
document.write("Is A===B ?\n" + (a === b));
```


Hello world
file:///C:/Users/Roger/Documents/VSCode/SoftwareStudio/index.html

```
Is A==B ? true
Is A===B ? false
```

Conditional Statements

- if…else
- For loop
- While loop
- Switch

if / else / else if

```
// Randomly create a number between 0 and 1
var a = Math.random();

// Use if…else to determine which block should be executed
if (a < 0.3) {
    console.log("a < 0.3");
}
else if (a < 0.6) {
    console.log("0.3 =< a < 0.6");
}
else {
    console.log("a >= 0.6");
}
```

if / else / else if (Cont’d)

- You can use ternary operators in JS

```
var a = Math.random();

a < 0.3 ? console.log("a < 0.3") :
    a < 0.6 ? console.log("0.3 =< a < 0.6") :
        console.log("a >= 0.6")
```

For Loop

- Loops are handy, if you want to run the same code repeatedly, each time with a different value

```
var num = [0, 1, 2, 3, 4, 5];
var sum = 0;

for (i = 0; i < num.length; i++) {
    sum +=  num[i];
}
```

While Loop

- The while loop loops through a block of code until the specified condition is true.

```
while (i < 5) {
    sum += i;
    i++;
}
```

```
do {
    sum += i;
    i++;
}
while (i < 4);
```

Switch

- The switch statement is used to perform different actions based on different conditions
- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.

Switch (Cont’d)

```
switch (expression) {
    case value1:
        // do something
        break;
    case value2:
        // do something
        break;
   default:
        // do something
        break;
}
```

JavaScript Object

- In JavaScript, almost "everything" is an object.
- In JavaScript, objects are king. If you understand objects, you understand JavaScript.

JavaScript Primitives

- **Primitive value:**
  - a value that has no properties or methods.
- **Primitive data type:**
  - data that has a primitive value.
- JS has 5 primitive data types:
  - string
  - number
  - boolean
  - null
  - undefined
- Primitive values are **immutable**
  - if x = 3.14, you can change the value of x. But you cannot change the value of 3.14.

Creating a JS Object

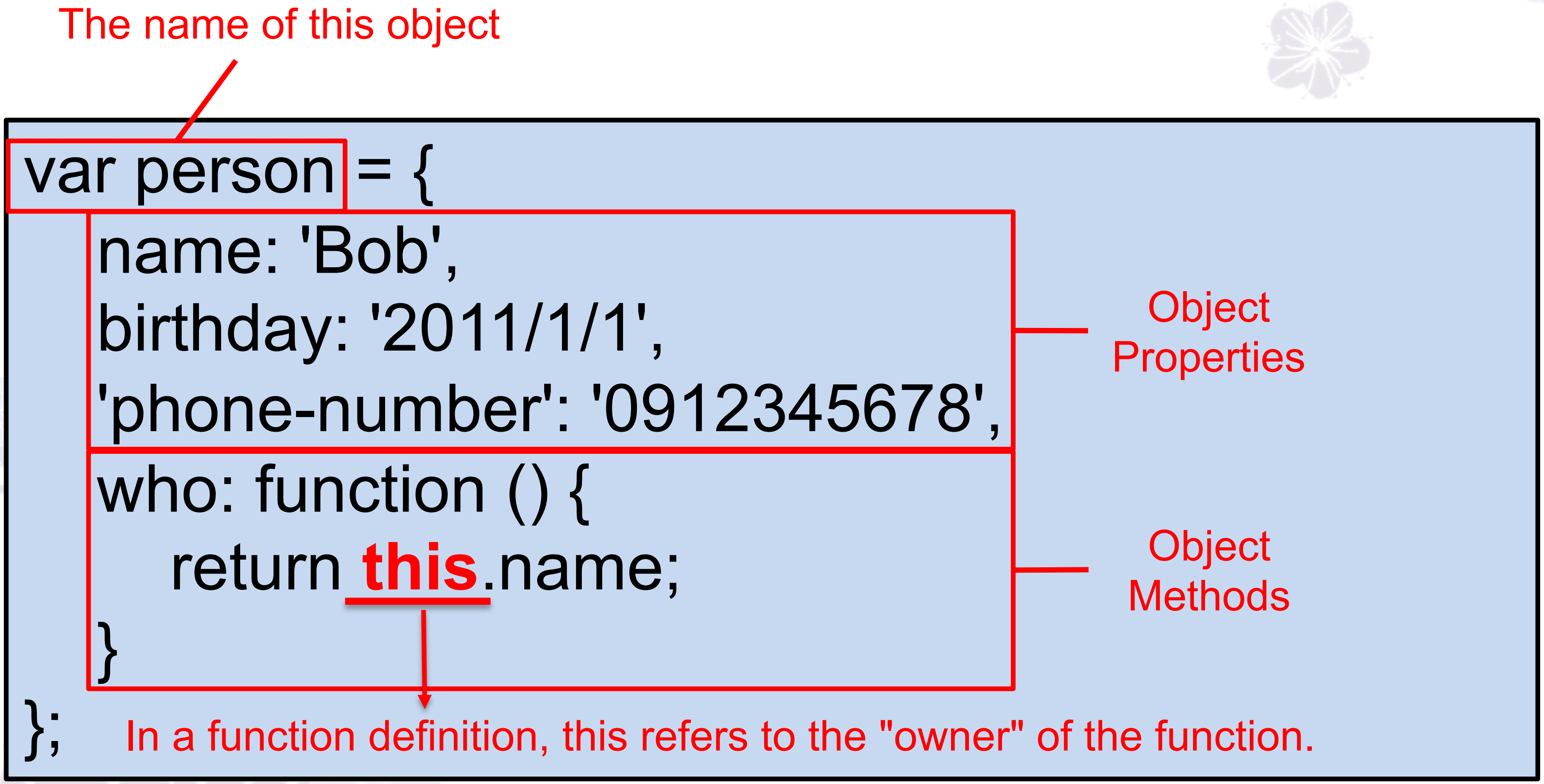
- JavaScript provides three ways for creating your own objects
  - **Object initializer**
  - Constructor function
  - Class declaration (ES6 or later)

Object Initializer

- A simple way to create an object in JS
- Write properties and methods in closed curly brackets(**{..}**)

```
var person = {
    name: 'Bob',
    birthday: '2011/1/1',
    'phone-number': '0912345678',
    who: function () {
        return this.name;
    }
};
```

Object Initializer



```
var person = {
    name: 'Bob',
    birthday: '2011/1/1',
    'phone-number': '0912345678',
    who: function () {
        return this.name;
    }
};
```

The name of this object

Object Properties

Object Methods

In a function definition, this refers to the "owner" of the function.

Object Properties and Methods

- Object properties:
  - Properties are the named values in an object.
  - Properties can be primitive values, other objects, and functions.
- Object methods:
  - An **object method** is an object property containing a **function definition**.
  - Methods are **actions** that can be performed on objects.

Accessing Property/Method

- When you want to use object's properties or methods, use **object key**
- Object key:
  - – name, birthday, phone-number, who

```
var person = {
    name: 'Bob',
    birthday: '2011/1/1',
    'phone-number': '0912345678',
    who: function () {
        return this.name;
    }
};
```

If object key contains non-standard character like "-", use quotation(' or ") to define it

Accessing Property/Method

```
var person = {
    name: 'Bob',
    birthday: '2011/1/1',
    'phone-number': '091234',
    who: function () {
        return this.name;
    }
};

console.log(person.name);                    // output: Bob
console.log(person['phone-number’]); // output: 091234
console.log(person.who());                   // output: Bob
```

Adding Property/Method

```
var person = {
    name: 'Bob',
    birthday: '2011/1/1',
    'phone-number': '0912345678',
    who: function () {
        return this.name;
    }
};

person.nationality = "English";   // adding a new property
person.InfoAll = function () {      // adding a new method
  return this. name  + " " + this. birthday; };
```

Can the properties/methods be deleted ?

Object Accessors

- ECMAScript 5 (2009) introduced **Getter** and **Setters**.
- Getters and setters allow you to define **Object Accessors** (Computed Properties).

Getter Accessors

- Using the **get** keyword, which binds an object property to a function that will be called when that property is looked up

```
var person = {
    name: 'Bob',
    birthday: '2011/1/1’,
    height: 170,
    'phone-number': '0912345678',
    who: function () {
        return this.name;
    }
    get myHeight() {return this.height; }
};

var height = person.myHeight;    // use it without "()"
```

Setter Accessors

- Using the **set** keyword, which binds an object property to a function that can change the values of properties.

```
var person = {
    name: 'Bob',
    birthday: '2011/1/1’,
    height: 170,
    'phone-number': '0912345678',
    who: function () {
        return this.name;
    }
    set changeHeight(newHeight) {    this.height = newHeight * 1.05;    }
};

person.changeHeight = 180;
```

Why Using Getters and Setters ?

- It gives simpler syntax.
- It allows equal syntax for properties and methods.
- It can secure better data quality.
- It is useful for doing things behind-the-scenes.

Object.create()

- Objects can also be created using the Object.create() method
- It allows you to choose the **prototype object** for the object you want to create, without having to define a new object

Object.create()

```
var person = {
    name: 'Bob',
    birthday: '2011/1/1',
    'phone-number': '0912345678',
    who: function () {
        return this.name;
    }
};

var person2 = Object.create(person);
person2.name = 'John';
person2.who();                                    // output: John
```

JavaScript Objects are Mutable

- Objects are mutable: They are addressed by **reference**, not by value.
- Primitive variables are not mutable.

```
var person = {
    name: 'Bob’,
    …
};

var person2 = person;     // person2 is a reference of person
person2.name = 'John’;
person2.who();            // output: John
person.who();             // output: John
```

Creating a JS Object

- JavaScript provides three ways for creating your own objects
  - Object initializer
  - **Constructor function**
  - Class declaration (ES6 or later)

Constructor Function

- Create a function for the object type that specifies its name, properties, and methods.
- Create an object instance with **new** keyword

```
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.info = function() {
        return this.make  + " " + this. Model + " " + this. year;
    };
}
var mycar = new Car('Eagle', 'Talon TSi', 1993);
var kenscar = new Car('Nissan', '300ZX', 1992);
var vpgscar = new Car('Mazda', 'Miata', 1990);
```

Adding Property/Method

```
var mycar = new Car('Eagle', 'Talon TSi', 1993);
var kenscar = new Car('Nissan', '300ZX', 1992);

// add a property and method that belong ONLY to mycar,
// not another object kenscar
mycar.price = 500;
mycar.salePrice = function () { this.price * 3};

// add a property that belongs ONLY to kenscar
kenscar.owner = “Peter”;
```

Adding Property/Method

- You **cannot** add a new property to an object constructor !
- You **cannot** add a new method to an object constructor !
- All properties and methods must be declared in the constructor function.
- But what if we want to add new properties/methods to **ALL** existing objects of a given type ?

Object Prototype

- The JavaScript **prototype** property allows you to add new properties/methods to object constructors.

```
function Car(make, model) {
    this.make = make;
    this.model = model;
}
Car.prototype.year = 2019;
Car.prototype.info = function()
    {return this.make  + " " + this.model + " " + this.year;};
```

Creating a JS Object

- JavaScript provides three ways for creating your own objects
  - Object initializer
  - Constructor function
  - **Class declaration (ES6 or later)**

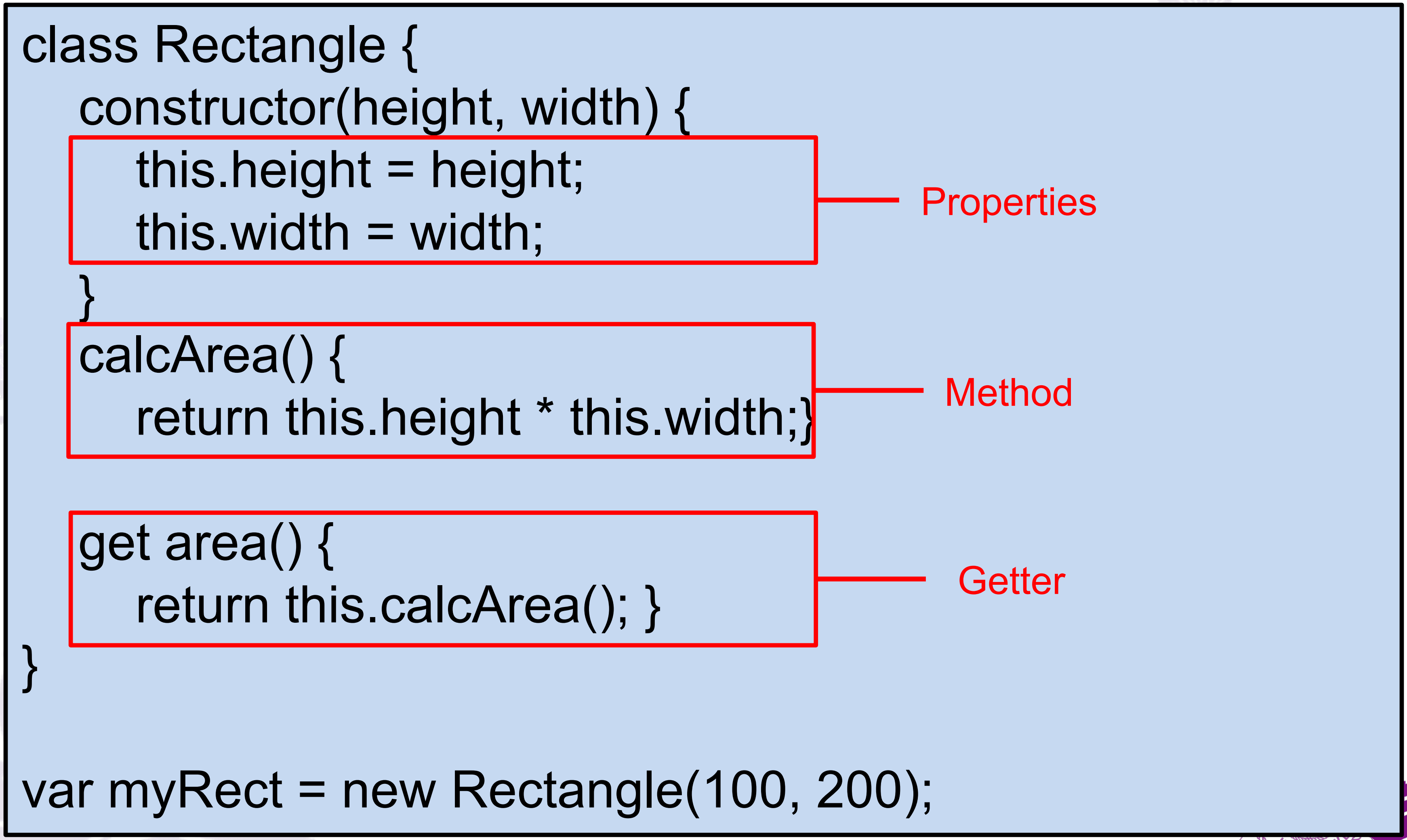
Class Declaration

- Define the properties and methods in class declaration
- Instantiate an object using **new** operator

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}

var myRect = new Rectangle(100, 200);
```

Class Declaration



```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  calcArea() {
    return this.height * this.width;}

  get area() {
    return this.calcArea(); }
}

var myRect = new Rectangle(100, 200);
```

Constructor

- Each class has **ONLY ONE** constructor
- Use the constructor to initialize an object
- Define the properties **ONLY** within the constructor

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Constructor

Static Method

- The **static** keyword defines a static method for a class
- Static methods are called without instantiating their class and cannot be called through a class instance

Static Method

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.sqrt(dx * dx + dy * dy);
    }
}

var p1 = new Point(5, 5);
var p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));            // output: 7.0710678118654755
```

Extends

- The **extends** keyword is used in class declarations or class expressions to create a class as a child of another class
- Use the **super** keyword to call the constructor of the parent class

Extends

```
class Person {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log('My name is ' + this.name);
    }
}

class Student extends Person {
    constructor(name, id) {
        super(name);
        this.id = id;
    }
}

var studentA = new Student('Bob', 's1234567890');
studentA.speak();                               // output: My name is Bob
```

subclass of Person

call the parent constructor

Array Object

- Array is a special type of object.
- Arrays use **numbers** to index its "elements"
- Objects use **strings(text)** to index its "elements"
- Reference

Array Object

- You can put different types of data in an array

```
var person = ["John", "Doe", 46];
                   String            Number

console.log(person[0]);                // output: John
console.log(person[1]);                // output: Doe
console.log(person[2]);                // output: 46
```

Array Object Initialization

- These two different statements both create a new array named points

```
// Bad
var points = new Array(40, 100, 1, 5, 25, 10);

// Good
var points = [40, 100, 1, 5, 25, 10];
```

- But there is no need to use the array constructor **new** Array(), **Use [] instead**.

Array Object Initialization

- Because the **new** keyword only complicates the code. It can also produce some unexpected results

```
// Creates an array with 2 elements (40 and 100)
var points = new Array(40, 100);

// Creates an array with 40 undefined elements!
var points = new Array(40);
```

Array Property

- An array length can be accessed by **length** property

```
var person = ["John", "Doe", 46];

console.log(person.length);          // output: 3
```

Array Operations

- There are some built-in Array operations:
  - push
  - pop
  - sort
  - reverse
  - toString
  - and more

Array Operations

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.pop();
fruits.push("Papaya");
console.log(fruits.toString());
// output: Banana,Orange,Apple,Papaya

fruits.reverse();
console.log(fruits.toString());
// output: Papaya,Apple,Orange,Banana
```

Take a Break!

Function

- A JS function is declared with the keyword **function**

```
function name (parameter1, parameter2, …) {
  // code to be executed
}
```

- A function can be used before its declaration and definition!

```
fun();
function fun () { console.log("Hello"); }
```

Function

- A function can be assigned to a variable
- When you assigned a function to a variable, you **cannot** call the function by its function name

```
var funVar = function fun () {
   console.log("Hello");
};

funVar(); // Correct!
fun();      //  Wrong! The system will throw an error message
```

Function Parameter

- Don't need to declare the type of incoming parameters
- Will not check the number of parameters, the missing parameters will be set to “Undefined”
- If the parameter is an **object**, it will be passed by **reference**, otherwise passed by **value**

Function Parameter (Cont’d)

- In ES6 and later version, you can assign initial values to the parameters

```
function multiply (arg1 = 3, arg2 = 4) {
    console.log(arg1 * arg2);
}
```

Function() Constructor

- Functions can also be defined with a built-in JS function constructor called "**F**unction()“ (Notice the uppercase ‘F’)
- Syntax:
  - Function("arg1", "arg2", …, "function body")

```
// using Function() constructor
var multiply = new Function("arg1", "arg2",
 "console.log(arg1 * arg2);");
```

Anonymous Function

- An anonymous function is a function that is declared without a named identifier

```
function (arg1, arg2) {
    console.log(arg1 * arg2);
}
```

Anonymous Function

- One common use for an anonymous function is to assign it to a **variable**

```
var multiply = function (arg1, arg2) {
    console.log(arg1 * arg2);
};

multiply(3, 4);
```

Anonymous Function

- Another common use for anonymous functions is as a **closure**
- It will invoke automatically, without being called
- It is also called **self-invoking** function

```
(function (arg1, arg2) {
   console.log(arg1 * arg2);
}) ();
```

Arrow Function

- Arrow functions allows a short syntax for writing function expressions.

```
var func1 = function(x, y) {
        return x * y;
};
var func2 = (x, y) => {
        return x * y;
};
// func1 and func2 are the same.
```

Nested Function

- Define a function within the body of another function
- Nested functions have access to the scope "above" them.

```
function multiply(arg1, arg2) {
    var ans = arg1 * arg2;

    function show() { console.log(ans); }

    show();
    return ans;
}
```

The internal function can access the variables defined in the parent scope

Closure

- Recap: JS variables could be either the **local-** or **global-**scoped.
- The **closure** is a mechanism to make global variables as local (private) ones.
  - Make it possible for a function to have "**private**" variables.

A closure is a function having access to the parent scope, even after the parent function has closed.

Ref: https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Closures

A Counter Dilemma

- Objective: design a counter function and made the counter available to all functions.

Approach #1

```
// Initiate counter
var counter = 0;

// Function to increase counter by 1
function add() { counter += 1; }

// Call add() 3 times
add(); add(); add();

// The counter should now be 3
```

Approach #1: Problem

- Any code on the page can change the *counter*, without calling add().
- The ***counter*** should be **local** to the add() function, to prevent other code from changing it.

Approach #2

```
// Initiate counter
var counter = 0;

// Function to increase counter by 1
function add() {
  var counter = 0;
  counter += 1;
}

// Call add() 3 times
add(); add(); add();

// The counter should now be 3, but it is 0. Why?
```

Approach #3

```
// Function to increase counter by 1
function add() {
  var counter = 0;
  counter += 1;
  return counter;
}

// Call add() 3 times
add(); add(); add();

// The counter should now be 3, but it is 1. Why?
```

Approach #4: Closure

```
var add = (function () {
  var counter = 0;
  return function () {counter += 1; return counter}
})();

// Call add() 3 times
add(); add(); add();

// The counter is now 3!
```

Approach #4: Explained

- The variable add is assigned the return value of a self-invoking function.
- The self-invoking function only runs once.
  - It sets the counter to zero (0), and
  - returns a function expression.
- This add becomes a nested function.
  - It can access the counter in the parent scope.
- The counter variable is now:
  - private to add function.
  - protected by the anonymous function.

Closures: Another Example

```
function multiplyMaker (arg1) {
    return function (arg2) {
        return arg1 * arg2;
    }
}
var multiplier3 = multiplyMaker(3);
var multiplier6 = multiplyMaker(6);

console.log(multiplier3(4));             // output: 12
console.log(multiplier6(4));             // output: 24
```

References

- W3Schools – JavaScript
- MDN Web Docs – JavaScript
- About Array:
  - W3Schools – Array
  - MDN - Array
- About Function:
  - Function
  - JavaScript Closures

thank you!

Question?