INT201 Client-side

JavaScript History

- 1995 - JavaScript is a programming language that was created by Brendan Eich who was working for Netscape.
- 1997 - JavaScript 1.1 proposal was submitted to the European Computer Manufacturers Association (ECMA).

ECMAScript

- The formal specification of the JavaScript language specified in the document ECMA-262
- ES1, ES2, ES3,...ESX are a different version of the ECMAScript specification

* Started from ES6, version of the ECMAScript start naming the versions based on the year of published specification, for example, ES2015 (ES6),ES2016 (ES7), ...

JavaScript

ES5 (2009) is fully supported by most modern browser in early 2016

- Higher order iteration functions (map, reduce, filter, foreach);
- JSON support;
- Better reflection and object properties;

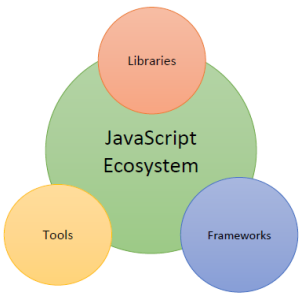
ES6 (ES2015) provide a greatly improved developer experience

- Classes
- Modules
- Iterators
- Generators
- Promises
- Arrow functions

From 2016 to 2019, a new edition of the ECMAScript standard was published each year, but the scope of changes was much smaller than the 5th or 6th editions

ES11 (ES2020), officially known as ECMAScript 2020, was published in June 2020

JavaScript EcoSystem



The different aspects of JavaScript

- Front-End: React, Angular, Vue.js, svelte, jQuery
- Back-End: node.js Deno
- Web Framework: Express
- Mobile: React Native, Apache Cordova Ionic
- Desktop: Electron
- Database: MongoDB

Introduction to JavaScript

- JavaScript is the programming language of the web.
- The overwhelming majority of websites use JavaScript, and all modern web browsers on desktops, tablets, and phones

ทำให้ JS มี power มากขึ้น ไม่ต้องมี browser ก็ลอง ประมวลผลได้

- Over the last decade, Node.js has enabled JavaScript programming outside of web browsers, and the dramatic success of Node means that JavaScript is now also the most used programming language among software developers.
- JavaScript is completely different from the Java programming language.

มีรวมมาจาก ECMAScript

JavaScript

ECMAScript	DOM	BOM
Scripting lan. คำสั่งที่ตกลงกันภาษา	can model document (html), สร้าง model tree ได้ ไม่ใช่ของ JS	

run ที่ browser

DOM: The Document Object Model. Map out an entire page as a hierarchy of nodes JS can เข้าไปจัดการ obj. ของเอกสารได้

BOM: The Browser Object Model. Deals with the browser window and frames can เข้าถึง obj. ตัวๆ บน browser ได้ ex. history, location bar etc.

Web Browser

Chromium
open source browser project
เป็น project ที่มี partner หลาย partner มาใช้ open source มาพัฒนา code base ของ browser ร่วมกัน

Chromium-based browser: Google Chrome, Microsoft Edge, Opera

พัฒนา browser เอง

- **Safari** is a graphical web browser developed by Apple, based on the WebKit engine.
- **Mozilla Firefox**, or simply **Firefox**, is a free and open-source web browser developed by the Mozilla Foundation and its subsidiary, the Mozilla Corporation. Firefox uses the Gecko layout engine to render web pages.

JavaScript Engine

Chrome V8 ใช้ JS ตัวเดียวกัน
open source JavaScript engine project

Chrome V8: Google Chrome, Microsoft Edge, Opera

- **JavaScriptCore**: A JavaScript interpreter and JIT originally derived from KJS. It is used in the WebKit project and applications such as Safari.
- **SpiderMonkey**: A JavaScript engine in Mozilla Gecko applications, including Firefox.

high lv. lan. ต้องไปแปลเป็นภาษาเครื่อง

JavaScript Development Environment

① run ใน
In Web Brower

- Google Chrome
- Microsoft Edge
- Safari
- Firefox
- Opera

② run ใน
Outside Web Browser (based on Chrome V8 JavaScript Engine)

server side scripting เหมือนเรา run ใน browser ประมวลผล high lv. lan.

Node.js: a JavaScript runtime built on Chrome's V8 JavaScript engine.

พัฒนาต่อ

Deno: a simple, modern and secure runtime for JavaScript and TypeScript that uses Chrome's V8 and is built in Rust.

MyFirstScript.js

```
console.log("I am JavaScript.");
```

Demo JavaScript In and Outside Web Browser

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="MyFirstScript.js"></script>
</head>
<body>
    <h1>Hello, This is my HTML page with JavaScript.</h1>
</body>
</html>
```

node ... ใน vs คือ run นอก browser but if อ้างถึง DOM & BOM ต้อง run ที่ browser bec. มีการเข้าถึง obj.

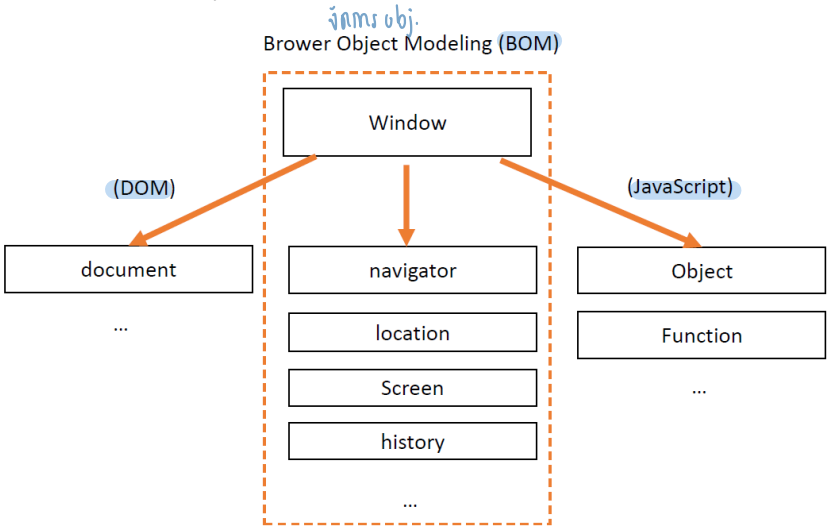
"terminal"
node
num
let num ประกาศตัวแปร
num = 3
num = 'INT201'
typeof (num) 'string'
num = 5
typeof (num) 'number'
num = {id: 1, name: 'Pomalee'}
num.email = 'pomalee.fonn@mail.kmutt.ac.th

: คั่นระหว่าง key กับ value
{} = obj.

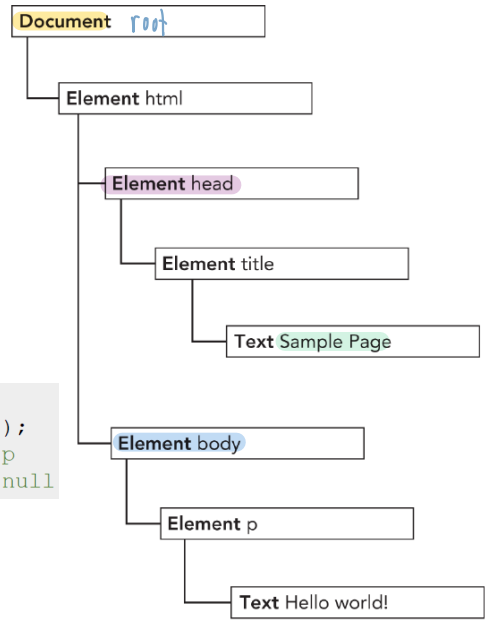
JavaScript Language Features

- Interpreted Language ใช้ translate ทีเดียว (แปลทีละบรรทัด if มี error ก็จะหยุด ทุกๆ ครั้งที่)
- Single Threaded, do one operation at one time ทำได้ทีละ 1 คำสั่ง
- Dynamically and weakly typed language //02_TypesValuesVariables/script1.js
- Support Object Oriented Programming (Prototyped based) (ตรงเปลี่ยนแปลงค่า type ของตัวแปรได้อิสระ)

The Window interface represents a window containing a DOM document. In a tabbed browser, each tab is represented by its own Window object.



tree structure ทำให้เห็น node ต่างๆ ได้ง่าย



DOM: The Document Object Model

```
<html>
    <head>
        <title>Sample Page</title>
    </head>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

```
const paragraphs =
document.getElementsByTagName("p");
alert(paragraphs[0].nodeName); //p
alert(paragraphs[0].nodeValue);//null
```

//01_BasicJS/script3.js

Asynchronous vs. Synchronous Programming

ทำ task ให้เสร็จก่อนค่อยไป

- **Synchronous** tasks are performed one at a time and only **when one is completed, the following is unblocked**. In other words, you need to wait for a task to finish to move to the next one.
- **Asynchronous** software design expands upon the concept by building code that allows a program to ask that a task be performed alongside the original task (or tasks), **without stopping to wait for the task to complete**. When the secondary task is completed, the original task is notified using an agreed-upon mechanism so that it knows the work is done, and that the result, if any, is available.

ยังทำไม่เสร็จก็ไปทำอันต่อไปก่อน

Asynchronous Callback Functions

Js ให้สิทธิ์ function เทียบเท่ากับ Data

In JavaScript, a callback function is a function that is passed into another function as an argument.

function ที่เป็น parameter

This function can then be invoked during the execution of that higher order function. รับ function เป็น parameter

// 01_BasicJS/script4.js

```
console.log('Hello');
setTimeout(function () {
  console.log('JS');
}, 5000);
console.log('Bye bye');
```

anonimus function
callback fun/fun

```
//Console

Hello
Bye bye

//until 5 seconds
JS
```

delay

Since, in JavaScript, functions are objects, functions can be passed as arguments.

setTimeout() executes a particular block of code once after a specified time has elapsed.

Higher-Order Functions

function ที่รับ / คืนค่าเป็น function ได้

A "higher-order function" is a function that accepts functions as parameters and/or returns a function.

JavaScript Functions are **first-class citizens** ประโยชน์ reuse ได้ เขียนแค่ครั้งเดียวใช้ได้ตลอด

- be assigned to variables (and treated as a value)
- be passed as an argument of another function
- be returned as a value from another function

```
//1. store functions in variables

function add(n1, n2) {
  return n1 + n2
}
let sum = add

let addResult1 = add(10, 20)
let addResult2 = sum(10, 20)

console.log(`add result1: ${addResult1}`)
console.log(`add result2: ${addResult2}`)
```

//01_BasicJS/script2.js

```
//2. returned as a value from another function

function operator(n1, n2, fn) {
  return fn(n1, n2)
}

//3. Passing a function to another function
function multiply(n1, n2) {
  return n1 * n2
}

let addResult3 = operator(5, 3, add)
let multiplyResult = operator(5, 3, multiply)

console.log(`add result3 : ${addResult3}`)
console.log(`multiply result: ${multiplyResult}`)
```

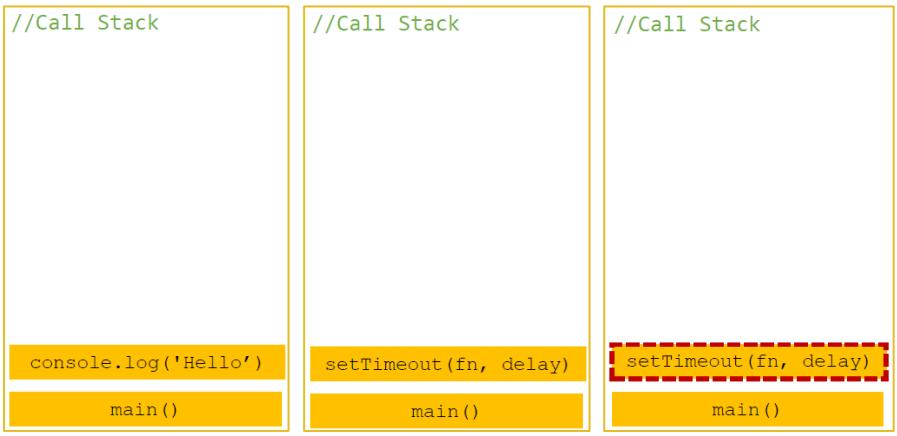
รับfunc เป็น para

Output

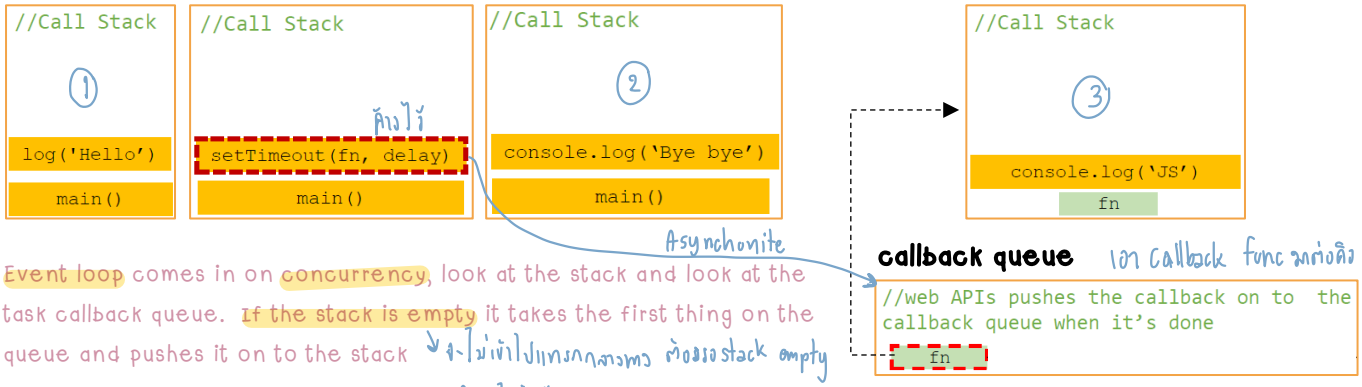
```
console.log('Hello');  (1)
setTimeout(function () {
  console.log('JS');  (3)
}, 5000);
console.log('Bye bye');  (2)
```

```
//Console
Hello
Bye bye
//until 5 seconds
JS
```

console.log('Bye bye')



with Single thread, JavaScript Runtime cannot do a setTimeout while you are doing another code



Event loop comes in on concurrency, look at the stack and look at the task callback queue. If the stack is empty it takes the first thing on the queue and pushes it on to the stack ↳ จะไม่เข้าไปแทรกกลางทาง ต้องรอ stack empty ไม่สนใจอะไรทั้งสิ้น

Vanilla JavaScript is just plain or pure JavaScript without any additional libraries or framework

Types, Values, and Variables

Basic JavaScript Statements

- Semicolon in the end of statement is an optional
  - let x=10; ใส่ / ไม่ใส่ก็ได้ but แนะนำให้ใส่

- let y=20
- Statement can take up multiple lines
- Comment
  - //Single Line Comment
  - /* ... */ Single or Multiple Lines Comment
- Console Printing
  - Console.log (*variable*); แสดงผล ตอนส่งงานให้ใส่

Reserved Words

ส่วนใหญ่จะเป็น key word ที่เป็นคำสั่ง

as	const	export	get	null	target	void
async	continue	extends	if	of	this	while
await	debugger	false	import	return	throw	with
break	default	finally	in	set	true	yield
case	delete	for	instanceof	static	try	catch
do	from	let	super	typeof	class	else
function	new	switch	var			

Types

JavaScript types can be divided into **two** categories:

1. primitive types

ได้ทั้ง ทศนิยม & ตัวเลข

- number -including integer and floating-point numbers between -2^{53} to 2^{53}
- string text มัน character , char ใส่ ' ' / " " ก็ได้
- boolean

Primitive value ค่าพื้นฐาน แตกต่อไปไม่ได้แล้ว

- number
- string
- boolean
- null (special type) แตกต่อไม่ได้แล้ว
- undefined (special type)
- symbol (special type) symbol = unistring (string ไม่ซ้ำ)

2. object types

- An object (that is, a member of the type object) is a collection of properties where each property has a name and a value (either a primitive value or another object) obj. ซ้อน obj. ได้
- a special kind of object, known as **an array**, เปรียบเทียบ address that represents an ordered collection of numbered values

JavaScript Data Types: numbers, string, boolean , undefined, symbol, object ทดลองกับเพื่อน

//02_TypesValuesVariables/script2.js

```
//output
type of myNum is number
type of myString is string
type of myBool is boolean
type of myUndefined is undefined
type of mySymbol is symbol
type of myNull is object
```

```
let myNum = 0;
console.log(`type of myNum is ${typeof myNum}`);

let myString = 'Good';
console.log(`type of myString is ${typeof myString}`);

let myBool = true;
console.log(`type of myBool is ${typeof myBool}`);

let myUndefined;
console.log(`type of myUndefined is ${typeof myUndefined}`);

let mySymbol = Symbol();
console.log(`type of mySymbol is ${typeof mySymbol}`);

let myNull = null;
console.log(`type of myNull is ${typeof myNull}`);

let myArr = [1, 2, 3];

console.log(`myArr Length: ${myArr.length}`);
console.log(`type of myArr is ${typeof myArr}`);

let myObj = {id: 1, task: 'grading exam'};

console.log(`${JSON.stringify(myObj)}`);
console.log(`type of myObj is ${typeof myObj}`);
```

ตัวไว้เฉยๆมีค่า ตัวไว้ไม่รู้ว่าจะมี/ไม่มีค่า

Null and undefined เป็น absent value คือไม่มีค่าเกิดขึ้น

- null is a language keyword that evaluates to a special value.
- null represent normal,expected absence of value and if there is no value, the value of variable can be set to null. If a variable is meant to later hold an object, it is recommended to initialize to null.
- Using the typeof operator on null returns the string "object" indicating that null can be thought of as a special object value that indicates "empty object pointer".
- JavaScript also has a second value that indicates absence of value. The undefinedvalue represents **unexpected absence of value**,a deeper kind of absence.
  - the value of variables that have not been initialized
  - the value you get when you query the value of an object property or array element that does not exist.
  - value of functions that do not explicitly return a value
  - value of function parameters for which no argument is passed
- If you apply the typeof operator to the undefined value, it returns "undefined", indicating that this value is the sole member of a special type.

The following table summarizes the possible return values of typeof

Type	Result
Undefined	"undefined"
Null	"object" (see below)
Boolean	"boolean"
Number	"number"
BigInt (new in ECMAScript 2020)	"bigint"
String	"string"
Symbol (new in ECMAScript 2015) เช่น unique string ไม่มีค่าซ้ำ	"symbol"
Function object (implements [[Call]] in ECMA-262 terms)	"function"
Any other object	"object"

Literals

```
15              // The number twelve
1.5             // The number one point two
"Hello World"   // A string of text
'Hi'            // Another string
`"I' am a student", I said` // Another string
true            // A Boolean value
false           // The other Boolean value
null            // Absence of an object
```

back tip ` ให้ค่าควรที่ บางครั้ง ตัวค่าควรที่อาจจะประกอบด้วย ' ' / " " ต่างๆ ไปรวมกันจบ ค.หมายความว่า string

Escape sequences can be used in JavaScript: \n,\t, \\, \b, ...

Identifiers ชื่อใดๆที่ตั้งชื่อใน program

- **Identifiers** are used to name constants, variables, properties, functions, and classes and to provide labels for certain loops in JavaScript code.

- A JavaScript identifier **must begin with a letter, an underscore (_), or a dollar sign ($).** Subsequent characters can be letters, digits, underscores, or dollar signs. (Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.)
- JavaScript is a **case-sensitive language**. This means that language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

let, var, const variables

- One of the features that came with ES 6 is the addition of let and const , which can be used for variable
- var declarations are **globally scoped or function/locally scoped.**
- The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window.
- All variables and functions declared globally with var **become properties and methods of the window object**.
- var is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

var variables

// 01_BasicJS/ script5.js

```
//greeting is globally scope, it exists outside a function
var greeting = 'Hey';
//var variables can be re-declared and updated
var greeting = 'Ho Ho';
function greeter() {
  //msg is function scoped, we cannot access the variable msg outside of a function
  var msg = 'hello';
}
// console.log(msg); //error: msg is not defined
console.log(greeting);
```

var variables can be re-declared and updated

This means that we can do this within the same scope and won't get an error.

```
var year = 'leap';
if (year === 'leap')
   var greeting = 'Hey 366 days'; //re-declared
console.log(greeting);
```

It becomes a problem when you do not realize that a variable greeting has already been defined before.

let variables

- let is now preferred for variable declaration.
- JavaScript block of code is bounded by {}. A block lives in curly braces.Anything within curly braces is a block.
- let is **block scoped**, a variable declared in a block withletis only available for use within that block.
- Let **can be updated but not re-declared.**

let can be updated but not re-declared. //01 _BasicJS/ script 6 .js

if the same variable is defined in different scopes, there will be no error. This is because both instances are treated as different variables since they have different scopes.

```
let greeting = 'Hey';
greeting = 'Ho Ho';
function greeter() {
  let greeting = 'Good morning';
  console.log(`greeting in function is ${greeting}`);
}

greeter(); console.log(greeting); //Ho Ho
```

```
/*let variables*/
//greeting is block scope
let greeting = 'Hey';
//let variables cannot be re-
declared, only can be updated
greeting = 'Ho Ho';
function greeter() {
  //msg is function scoped, we cannot access the
 variable msg outside of a function
  let msg = 'hello';
}
// console.log(msg); //error: msg is not defined
console.log(greeting);
let year = 'leap';
if (year === 'leap')
   greeting = 'Hey 366 days';
console.log(greeting);
```

const

- Variables declared with the const maintain constant values.
- const declarations share some similarities with let declarations.
- Like let declarations, const declarations can only be accessed within the block they were declared.
- const **cannot be updated or re-declared**
- Every const declaration, therefore, **must be initialized at the time of declaration.**

```
/*const variables*/
const greeting = 'Hey';
//const variables cannot be re-declared
// const greeting = 'Ho Ho';
//const variables cannot be updated
// greeting = 'Hi Hi';
```

//01_BasicJS/ script 7 .js

ControlStructures file P. 10

JavaScript String

- The JavaScript type for representing text is the string.
- A string is an immutable ordered sequence of 16-bit values.
- JavaScript's strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1, and so on.
- The empty string is the string of length 0.
- JavaScript does not have a special type that represents a single element of a string. To represent a single 16-bit value, simply use a string that has a length of 1.

let msg = 'JS'
msg JS
msg.length 2
msg.charAt(msg.length-1) S
msg.toLowerCase() js
msg JS
msg.substring(1,2) S

Template Literals

```
let name = `Umaporn`;
let greeting = `Hello ${ name }`;
```

console.log("Hello" + msg) Hello js

- This is more than just another string literal syntax, however, because these template literals can include arbitrary JavaScript expressions.
- Everything between the **${}** is interpreted as **a JavaScript expression**
- Everything **outside the curly braces** is **normal string literal text**
- The final value of a string literal in backticks is computed by
  - evaluating any included expressions,
  - converting the values of those expressions to strings and
  - combining those computed strings with the literal characters within the backticks

- ${x+y} operation (arithmetic/logical) and operands (x,y)
- Simple expression -> complicated exp
- (Boolean exp) ? (true) expression : (false) expression

Explicit Conversions

- Although JavaScript performs many type conversions automatically, you may sometimes need to perform an explicit conversion, or you may prefer to make the conversions explicit to keep your code clearer.
- The simplest way to perform an explicit type conversion is to use the Boolean(), Number(), and String() functions:

```
//Explicit Conversions
Number('3'); //   3
String(false); //  "false"
Boolean([]); //   true
```

How values convert from one type to another in JavaScript?

```
1 + ' objects'; //"1 objects": Number 1 converts to a string
'5' * '4'; //20: both strings convert to numbers
let n = 'y' + 1; // n == NaN; string "y" can't convert to a number
```

JavaScript implicit type conversions

"2" - 1 → Number("2") - 1

explicit ≈ manual
implicit ≈ auto

The primitive-to-primitive conversions shown in the table are relatively straightforward but Object-to-primitive conversion is somewhat more complicated

```
//examples of implicit type conversions

x + ""      // String(x)

+x          // Number(x)

x-0         // Number(x)

!!x         // Boolean(x)
```

//02_TypesValuesVariables/script5.js

Value	to String	to Number	to Boolean
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	`0`	`false`
`true`	`"true"`	`1`	
`false`	`"false"`	`0`	
`""` (empty string)		`0`	**`false`**
`"1.2"` (nonempty, numeric)		`1.2`	`true`
`"one"` (nonempty, non-numeric)		`NaN`	`true`
`0`	`"0"`		**`false`**
`-0`	`"0"`		**`false`**
`1` (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`[]` (empty array)	`""`	`0`	`true`

Control Structures

JavaScript Operators

//02_TypesValuesVariables/script4.js

Operator precedence and associativity specify the order in which operations are performed in a complex expression.

ส.สำคัญ มาก ทำก่อน

- Increment and Decrement
  ++ --
- Invert Boolean value
  !
- Type of operand
  typeof
- Arithmetic operators
  * / % + -
- Relational operators
  < <= > >=
- Equality operators
  == != (non strict equality) → ไม่ check conversion
  === !== (strict equality) → มี conversion ไม่ทำก็นับนี่
- Logical operators
  && ||
- Conditional operators
  ?:
- Assignment operators
  += -= *= /= %=

น้อย ทำหลัง

//02_TypesValuesVariables/script5.js

```
null == undefined // true: These two values are treated as equal.
"0" == 0          // true: String converts to a number before comparing.
0 == false        // true: Boolean converts to number before comparing.
"0" == false      // true: Both operands convert to 0 before comparing.


//if change from  == to strict equality ===, the results are all FALSE!
```

Conversions and Equality

- JavaScript has two operators that test whether two values are equal.
- The "strict equality operator,"===, does not consider its operands to be equal if they are not of the same type. return false ได้เลย
- But because JavaScript is so flexible with type conversions, it also defines the==operator with a flexible definition of equality.

Equality with type conversion ระวัง obj. type

The equality operator == is like the strict equality operator, but it is less strict. If the values of the two operands are not the same type, it attempts some type conversions and tries the comparison again:

- If the **two values have the same type**, test them for strict equality as described previously. **If they are strictly equal, they are equal. If they are not strictly equal, they are not equal.**

- If the **two values do not have the same type**, the **== operator may still consider them equal**. It uses the following rules and type conversions to check for equality:
  - If **one value is null and the other is undefined**, they are **equal.**
  - If **one value is a number and the other is a string**, **convert the string to a number** and try the comparison again, using the converted value.
  - **If either value is true, convert it to 1** and try the comparison again. **If either value is false, convert it to 0** and try the comparison again.
  - Any other combinations of values are not equal.
- If **one of the operands is an object** and **the other is a number or a string try to convert the object to a primitive using the object's valueOf() and toString() methods.**

str → Num
true → 1
false → 0

//02_TypesValuesVariables/script4.js

```
//Arithmetic operators
console.log(5 + 2); // => 7: addition
console.log(5 - 2); // => 3: subtraction
console.log(5 * 2); // => 10: multiplication
console.log(5 / 2); // => 2.5: division

// JavaScript defines some shorthand arithmetic operators
let count = 0; // Define a variable
count++; // Increment the variable
count--; // Decrement the variable
count += 3; // Add 3: same as count = count + 3;
count *= 2; // Multiply by 2: same as count = count * 2;
console.log(`count = ${count}`); // => 6: variable names are expressions

//conditional operator
let result = count > 5 ? 'count > 5' : ' count<=5';
console.log(`result = ${result}"`);

//== and != non-strict equality
//If the two operands are different types, interpreter attempts to convert them to suitable type.
console.log(`15 == '15' ${15 == '15'}`); //true

//=== and!=== strict equality without type conversion
console.log(`15 === '15' ${15 === '15'}`); //false

//logical operators
// && (and), || (or), ! (not)

console.log(`5 < '10' && '1' > 5 is  ${5 < '10' && '1' > 5}`); //false

console.log(`5 < '10' || '1' > 5 is  ${5 < '10' || '1' > 5}`); //true
console.log(`!(0) is ${!0}`); //true
```

Null & undefined are special types

array เป็น obj.

อย่าไม่แน่ใจ

คำตอบคือ ?, ใช้ implicit function ?

- 1 == true true, Number() convert true -> 1
- * "1" == [] false, ~~Number([]) == Number("1")~~, "1" == ""
- "1" == 1 true, Number("1")
- * 1 == null false, ~~Number(null)~~, => null & number => false, null & false => false
- * "1" == undefined false, ~~Number(undefined) = NaN~~
- typeof (null) 'object'
- null == 0 false
- typeof ([]) 'object'
- "0" == [] false
- 0 == [] true

JavaScript String

เปรียบเทียบ string 'B' < 'a' อิงตาม ชาติ code 65 < 97

'A' - 'Z' = 65-90
'a' - 'z' = 97-122

- Strings can be compared with the standard === equality and !== inequality operators
- two strings are equal if and only if they consist of exactly the samesequence of 16-bit values.
- Strings can also be compared with the <, <=, >, and >= operators. String comparison is done simply by comparing the 16-bit values.
- To determine the length of a string—the number of 16-bit values it contains—use the length property of the string: str.length

//02_TypesValuesVariables/script3.js

```
let str1 = 'Hello';
let str2 = 'hello';
console.log(`str1 === str2 is ${str1 === str2}`);
console.log(`str1 < str2 is ${str1 < str2}`);
console.log(`str1 > str2 is ${str1 > str2}`);
console.log(`str1.length = ${str1.length}`);
console.log(
  `str1.toLowerCase === str2.toLowerCase is ${
    str1.toLowerCase === str2.toLowerCase
  }`
);
console.log(`str1.charAt(str1.length-1) = ${str1.charAt(str1.length - 1)}`);
```

```
//output
str1 === str2 is false
str1 < str2 is true
str1 > str2 is false
str1.length = 5
str1.toLowerCase === str2.toLowerCase is true
str1.charAt(str1.length-1) = o
```

เปรียบเทียบ ค่า value เปรียบเทียบ address ของ obj.

Comparing Primitives vs Objects

- **Primitives are also compared by value**: two values are the same only if they have the same value.
- Objects are not compared by value: two distinct objects are not equal even if they have the same properties and values.
- **Objects** are sometimes called **reference types** to distinguish them from JavaScript's primitive types
- we say that objects are compared by reference: two object values are the same if and only if they refer to the same underlying object.

//02_TypesValuesVariables/script2.js

```
let myObj = {
  id: 1,
  task: 'grading exam'
};

let myObj2 = {
  id: 1,
  task: 'grading exam'
};

newObj = myObj;
console.log(`newObj === myObj is ${newObj === myObj}`);
console.log(`myObj1 === myObj2 is ${myObj === myObj2}`);
```

myObj. myObj2 newObj. id = 1 task = xxx myobj2

ชี้ที่ address เดียวกัน

ชี้คนละ obj.

```
//output
newObj === myObj is true
myObj1 === myObj2 is false
```

And two distinct arrays are not equal even if they have the same elements in the same order:

```
let a = [];
let b = a;
b[0] = 1;
let c = [1];
console.log(`a === b is ${a === b}`);
console.log(`b == c is ${b == c}`);
```

//02_TypesValuesVariables/script3.js

```
//output
a === b is true
b == c is false
```

obj. เดียวกัน อยู่ที่เดียวกัน

a b 1 c 1

คนละ array

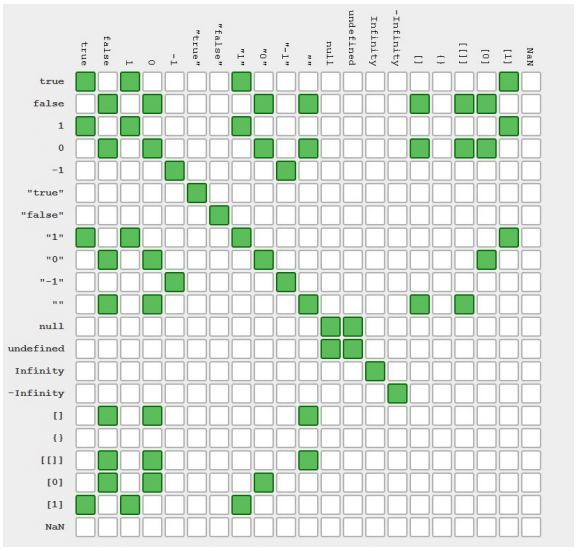
Conditionals - *if/else*

use a statement block { } to combine multiple statements into one

```
if (expression)
  statement
```

```
if (expression)
    statement1
else
    statement2
```

```
if (expression1) {
    // Execute code block #1
}
else if (expression2) {
    // Execute code block #2
}
else if (expression3) {
    // Execute code block #3
}
else {
    // If all else fails, execute block #4
}
```



Conditionals - *switch*

The matching case is determined using the === identity operator, not the==equality operator, so the expressions must match without any type conversion.

```
switch(n) {
   case 1: // Start here if n === 1
   // Execute code block #1.
   break; // Stop here
   case 2: // Start here if n === 2
   // Execute code block #2.
   break; // Stop here
   case 3:
   // Start here if n === 3 // Execute code block #3.
   break; // Stop here
   default:
   // If all else fails... // Execute code block #4.
   break; // Stop here
}
```

```
switch(expression){
    statements
}
```

Loop - *while/do while*

```
while (expression)
   statement
```

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

```
do
   statement
while (expression);
```

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

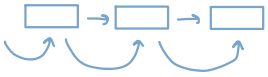
Loop - *for*

The **for** statement simplifies loops that follow a common pattern.

```
for(initialize ; test ; increment)
   statement
```

```
for(let i = 0, len = data.length; i < len; i++)
   console.log(data[i]);
```

The **for/of** loopworks with *iterable* objects, arrays, strings, sets, and maps are iterable:



```
for(variable of iterableObject)
   statement
```

```
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9]
let sum = 0;
for(let element of data) {
    sum += element;
});
console.log(`sum = ${sum}`); //sum=45
```

เข้าถึงแต่ละ property

The **for/in** statement loops through the property names of a specified object

```
for (variable in object)
    statement
```

```
for(let property in object) {
   console.log(property); //print property name
   console.log (object[property]); //print value of each property
}
```

แบบฝึกหัด 1 เขียนโปรแกรมเพื่อแสดงราศีที่ตรงกับปีที่กำหนดไว้ โดยมีทั้งหมด 12 ราศี ซึ่งแทนโดยสัตว์ประเภทต่าง ๆ ตัวอย่างเช่น ปี 1900 % 12 จะมีค่า 4 ซึ่งจะแทนด้วยราศีหนู

0: monkey
1: rooster
2: dog
3: pig
4: rat
5: ox
6: tiger
7: rabbit
8: dragon
9: snake
10: horse
11: sheep

https://www.hisgo.com/us/destination-japan/blog/japanese_horoscope.html

แบบฝึกหัดที่ 4 ให้เขียนโปรแกรมเพื่อทำเมนูให้เลือกกับการจัดการ Text String ให้ทดสอบโดยใช้ String อย่างน้อย 3 กรณีที่แตกต่างกัน

- ให้เขียน Function เพื่อแสดงเมนูให้เลือกในการจัดการ String
  - 1: Reverse String
  - 2: Replace Vowels with '*'
  - 3: Count Vowels in String
- ตัวอย่างเช่น "Hello World"
  - กด 1 ได้ "dlroW olleH"
  - กด 2 ได้ "H*ll* W*rld"
  - กด 3 ได้ 3

W4

JavaScript Objects

let obj = { ... } ชื่อที่เก็บคือ obj.

- ECMAScript **objects as hash tables**: nothing more than a grouping of name-value pairs where the value may be **data or a function**.

  set ของ key กับ value
  เข้าถึง / ค้นหา data ได้เร็ว bec. จัดเก็บด้วย hashcode big O 1
  เป็นได้ทั้ง data & เก็บ function ได้
- **An object** is an **unordered collection of properties**

  object ใน object
- An object is a **composite value**: it aggregates multiple values (primitive values or other objects) and allows you to store and retrieve those values by name.

  ซ้อนกันได้
- Property names are usually Strings or can also be Symbols.

  unique string
  book.pages = 200 same as book["pages"] = 200
- No object may have two properties with the same name.

  if ชื่อซ้ำกัน จะใช้ key value ตัวสุดท้าย / ล่าสุด
- JavaScript objects are dynamic—properties can usually be added and deleted

  เปลี่ยนแปลงค่าได้
- It is possible to create an instance of an "implicit" class without the need to actually create the class.

Learn > L2.js

obj. can แก้ไขได้ตลอด
เพิ่ม, แก้ไข, ลบ

delete book.version ก็จะลบ version ใน book ได้เลย

JavaScript Object Examples

ใช้ { } ประกาศ property list ทั้งหมด → object แบบ literal
└ กำหนด key กับ value ไว้ชัดเจน

```
//Simple Object
let student = {
name: 'Bob',
age: 32,
gender: 'male',
```

```
//Object Value is array
let profile = {
 id: 123,
 interests: [ 'music', 'skiing']
}
```

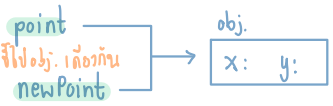
```
//Aggregated Object  object ซ้อน object
let book  = { isbn: 123456789,
              title: "JavaScript",
              author: {
                 firstname: "Umaporn",
                 lastname: "Sup"
              }
              };
```

Object Passing to functions by reference

เปลี่ยนแปลง obj. ทันที

Objects are **mutable** and manipulated by reference rather than by value.

```
//04_Objects/script2.js
let point = { x:10, y: 20 };
let newPoint = point;
newPoint.x = 30;
console.log (point) //{x:30, y:20);
```



เกิดการเปลี่ยนแปลงใดๆที่ property key จะเกิดการเปลี่ยนแปลงที่ obj. นั้นเลย
ไม่มีการสร้าง value ใหม่ / property ใหม่

Understanding Objects

```
//create object without class
//The function distance does not care
//whether the arguments are an instance of the class Point
function distance(p1, p2) {
  console.log(typeof p1); //object
  console.log(typeof p2); //object
  // ** - The exponentiation assignment operator
  return Math.sqrt((p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2);
}

console.log(distance({ x: 1, y: 1 }, { x: 2, y: 2 })); //1.4142135623730951
```

formal parameter ชื่อ parameter ที่ไม่ได้บอก data type

actual parameter ไม่เหมือนกันในแต่ละครั้ง

Shorthand Object Methods

- When function is defined as a property of an object, we call that function a method
- Prior to ES6 (ก่อน)

```
let square ={
    area: function(){ return this.side * this.side;},
    side: 10
};
square.area() //=>100
```

- In ES6, the object literal syntax has been extended to allow a shortcut where the function keyword and the colon are omitted,

```
let square ={
    area (){ return this.side * this.side;},
    side: 10
};
square.area() //=>100
```

ยุบ

Understanding Object Creation

1. Simplest form with **object literals**, object literal is a comma-separated list of {name: value} pairs.

```
let point = {x:10, y: 20};
```

2. with the **new** operator. Objects created using the new keyword and a constructor invocation use the value of the prototype property of the constructor function as their prototype.

```
let person = new Object(); // let person = {}
let a = new Array(); // let a = []
let p = new Point();
```

3. with the **Object.create()** function. Creates a new object, with specified prototypes.

```
let o = Object.create({x: 1, y: 2});
let p = Object.create(o);
console.log(p.x); //1
console.log(p.y); //2
```

Understanding Object Creation

```
//new Object()
let person = new Object();
person.name = "Adam";
person.age = 29;
person.greeting = function(){
    console.log("Hello" + person.name);
};

//{ name: 'Adam', age: 29, greeting: [Function] }
```

```
//Object Literal
let person={}; //no property
console.log(typeof person) //object
```

```
let person = {
    name: "Adam", //property key: value
    age: 29,
    greeting(){ //property function
        console.log("Hello" + person.name);
    }
};
```

```
//both can call greeting() function
person.greeting();
```

```
Hello, Adam
```

Object literals

- The easiest way to create an object is to include an object literal in your JavaScript code.
- In its simplest form, an object literal is a comma-separated list of colon-separated **name: value** pairs, enclosed within curly braces {}.
- A **property name** is a JavaScript identifier or a string literal
- A **property value** is any JavaScript expression; the value of the expression (it may be a primitive value or an object value) becomes the value of the property

Object literal Examples

```
let empty={};
let point = {x:0, y:0};
let p2={x: point.x, y: point.y+1};
let book ={
    "main title": "JavaScript", //These property names include spaces
    "sub-title": "The Definitive Guide", //and hyphens, so use string literals
    for: "all audiences",
    author:{
        firstname: "David",
        Surname: "Flanagan"
    }
};
```

Getting, Setting , Creating Object Properties

- To obtain the value of a property, use the dot(.) or square bracket([]) operators

```
let book = {  isbn: 123456789,
              title: "JavaScript",
              author:{
                firstname: "Umaporn",
                lastname: "Sup"
              }
};
```

```
object.property
object["property"]
```

```
//getting object property
console.log (book.isbn);
console.log(book["title"]);
Console.log(book['author']['firstname'])
//setting object property
book.author.firstname = "Uma";
//create new object property
book["publishedYear"]=2000;
//or book.publishedYear=2000;
```

```
//console.log(book)
{
 isbn: 123456789,
 title: 'JavaScript',
 author: { firstname: 'Uma', lastname: 'Sup' },
 publishedYear: 2000
}
```

- with the[] array notation, the name of the property is expressed as a string.
- Strings are JavaScript data types, so they can be manipulated and created while a program is running.

Create class and constructor functions (ES6)

```
//Recommendation to wrapping property and function within a single unit
//A class can be composed of the class's constructor method, instance methods, getters,
setters, and static class methods.
class Rectangle{
  constructor(width, height){ //invoke by new operator
    // Everything added to 'this' will exist on each individual instance
    this._width=width;
    this._height=height;
  }
  // Everything defined in the class body is defined on the class prototype object and
sharing between instances.
  area(){ // method
    return this._width*this._height;
  }
}
```

```
let rec1=new Rectangle (2, 3);
console.log(rec1.area()); //6
```

```
class Rectangle {
  constructor(width, height) {
    this._width = width;
    this._height = height;
  }
  area() {
    return this._width * this._height;
  }
  get width() {
    return this._width;
  }
  set width(newWidth) {
    this._width = newWidth;
  }
  get height() {
    return this._height;
  }
  set height(newHeight) {
    this._height = newHeight;
  }
  toString(){
    return "width = "+this._width + ", height = "+this._height +
    ", area = " + this.area();
  }
}
```

//04_Objects/script1.js

```
//Get and Set
//ES6 classes brings a new syntax for getters and
setters on object properties.
//Get and set allows us to run code on the reading or
writing of a property.
//use _ convention to create a backing field to store
our name property. Without this every time get or set is
called it would cause a stack overflow.
```

```
let rec1=new Rectangle (2.5, 3.5);
console.log(rec1.width);  //2.5
console.log(rec1.height); //3.5
console.log(rec1.area()); //8.75
console.log(rec1.toString());
//width = 2.5, height = 3.5, area = 8.75
```

```
//create object with Function Constructor Pattern (ES5)
//It is also possible to define custom constructors, in the form of a function,
//that define properties and methods for your own type of object.
//By convention, constructor functions always begin with an uppercase letter,
//whereas non constructor functions begin with a lowercase letter.
```

//04_Objects/script4.js

```
function Person(name, age, job){
 this.name = name;
 this.age = age;
 this.job = job;
 this.sayName = function() {
  console.log(this.name);
 };
}

let person1 = new Person("Pot", 40, "Tester");
let person2 = new Person("Joe", 20, "Doctor");

person1.sayName(); //"Pot"
person2.sayName(); //"Joe
```

Any function that is called with the **new** operator acts as a constructor, whereas any function called without it acts just as you would expect a normal function call to act.

```
Person.prototype.greeting = function () {
  return `Hello, ${this.name}`;
};

console.log(person1.greeting());
```

P.12 - 18

Object Prototypes

- **Prototypes** are the mechanism by which JavaScript objects inherit features from one another.
- **JavaScript** is often described as a prototype-based language— to provide inheritance, objects can have a prototype object, which acts as a template object that it inherits methods and properties from.

Prototype Chaining

การสืบทอด object

Object > Rectangle > Square

- ECMA-262 describes **prototype chaining** as the primary method of **inheritance** in ECMAScript.
- The object created by **new** Object() or **object literal** inherit from **Object.prototype**
- Similarly, the object created by new Array()uses **Array.prototype** as its prototype, and the object created by new Date() uses **Date.prototype** as its prototype.
- **Date.prototype** inherits properties from **Object.prototype**, so a **Date** object created by **new Date()** inherits properties from both **Date.prototype** and **Object.prototype**.
- This linked series of prototype objects is known as a **prototype chain**.
- JavaScript objects have a set of "**own properties**" and they also inherit a set of properties from their prototype object.

```
let o = {};                 // o inherits object methods from Object.prototype
o.x = 1;                    // and it now has an own property x.
let p = Object.create(o);   // p inherits properties from o and Object.prototype
p.y = 2;                    // and has an own property y.
let q = Object.create(p);   // q inherits properties from p, o, and Object.prototype
q.z = 3;                    // and has an own property z.
let f = q.toString();       // toString is inherited from Object.prototype
q.x + q.y                   // => 3; x and y are inherited from o and p
```

`prototypeObj.isPrototypeOf(object)`

object - the object whose prototype chain will be searched. Return a Boolean indicating whether the calling object lies in the prototype chain of the specified object.

```
//define our own class and
//constructor functions
class Rectangle{
  constructor(width, height){
    this._width=width;
    this._height=height;
  }
  area(){
    return this._width*this._height;
  }
}
let rec1=new Rectangle (2, 3);
console.log(rec1.area()); //6
```

```
//create object with Object.create()
let square = Object.create(rec1);
square.perimeter = function() {
  return 4 * this.width;
}
console.log(square.width); //2
console.log(square.height);//3
console.log(square.area());//6
console.log(square.perimeter());//8
console.log(Object.prototype.isPrototypeOf(rec1));//true
console.log(Rectangle.prototype.isPrototypeOf(square));//true
console.log(Object.prototype.isPrototypeOf(square));//true
```

Rectangle

WG

How to Compare Objects in JavaScript

1. Referential equality: `==, ===, Object.is()`
2. Manual comparison of properties' values.
3. Shallow Equality check the properties' values for equality.

Memory Address	Variable Value
1100 (num1)	10
1101	
1102	
1103 (num2)	10
...	
11EE (num)	20

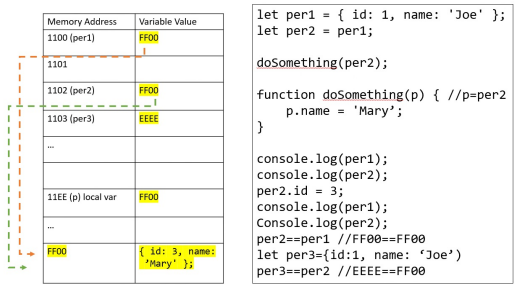
```
let num1 = 10;
let num2 = num1;
doSomething(num2)
function doSomething(num) {
//num=num2
  num = 20;
}
num1==num2 // 10==10
```

Referential equality

- Both are the same object means both object point to the same object instances.
- Three ways to compare objects:

Learn > L4.js

- The strict equality operator ===
- The loose equality operator ==
- Object.is() function



//04_Objects/script3.js

```
//Object Comparing
let student = { id: 1, name: "Joe" };
let newStudent = { id: 2, name: "Joe" };
let oldStudent = { id: 1, name: "Joe" };
let alumniStudent = student;
```

```
if (student == alumniStudent) { //true
  console.log("student equals to alumni student by ==");
  //student equals to alumni student by ==
}
if (student == newStudent) { //false
  console.log("student equals alumni student by ==");
}
if (student === alumniStudent) { //true
  console.log("student strictly equals to alumni student");
  //student strictly equals to alumni student.
}

if (student === newStudent) { //false
  console.log("student strictly equals to new student by ==");
}
```

//04_Objects/script3.js

```
//Object Comparing
let student = { id: 1, name: "Joe" };
let newStudent = { id: 2, name: "Joe" };
let oldStudent = { id: 1, name: "Joe" };
let alumniStudent = student;
```

```
/*The Object.is()method determines whether two values are the same value without
type conversion.  Both the same object means both object have same reference*/

  if (Object.is(student, alumniStudent)) { //true
    console.log("student equals to alumni student by Object.is()");
    //student equals to alumni student by Object.is()
  }
  if (Object.is(student, newStudent)) { //false
    console.log("student equals to new student by Object.is()");
  }

  if (Object.is(student, oldStudent)) { //false
    console.log("student equals to old student by Object.is()");
  }
```

Manual Comparison

A manual comparison of properties' values.

//04_Objects/script3.js

```
//compare properties manually
function isStudentEqual(object1, object2) {
  return object1.id === object2.id;
}
```

id เบอร์โทรศัพท์

เปรียบเทียบ ของใน object

```
console.log(isStudentEqual(student, oldStudent)); //true
console.log(isStudentEqual(student, alumniStudent));//true
```

Shallow Equality

เช็คค่าเท่ากันแบบไม่ลงไปลึกสุด

Deep Equality

เปรียบเทียบแบบลึกลึก

return เป็น string ของ key property

`Object.keys(obj)` //04_Objects/script3.js

obj - the object of which the enumerable's own properties are to be returned. Return an array of strings that represent all the enumerable properties of the given object.

```
//3. Shallow Equality
let book1 = {
  isbn: 123456789,
  title: "JavaScript",
};

let book2 = {
  isbn: 123456789,
  title: "JavaScript",
};
```

```
function shallowEquality(object1, object2){
  const keys1=Object.keys(object1);
  const keys2=Object.keys(object2);

  if(keys1.length !== keys2.length){
    return false;
  }
  for(let key of keys1){
    if(object1[key] !== object2[key] ){
      return false;
    }

  }
  return true;
}
```

ดูว่า book1 กับ book2 มี key properties เหมือนกัน? แต่ไม่ได้ดูค่าข้างใน properties

```
console.log("shallow equality: " + shallowEquality(book1, book2)); //true
```

JSON – JavaScript Object Notation

https://www.borntodev.com/2020/02/28/what-is-json/

ตั้งต้นมาจาก ภาษา JS แต่ไม่ใช่ JS

- JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax.
- It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa).
- Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript, and many programming environments feature the ability to read (parse) and generate JSON.
- A JSON string can be stored in its own file, which is basically just a text file with an extension of .json, and a MIME type of application/json.

JSON structure

JSON.stringify()
JSON.parse()

- **JSON is a string** whose format very much resembles JavaScript object literal format.
- JSON **requires double quotes** to be used around strings and property names. **Single quotes are not valid** other than surrounding the entire JSON string.
- You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals.
- JSON is purely a string with a specified data format — **it contains only properties, no methods.**
- We can also convert arrays to/from JSON.

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

```
[
  {
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": [
      "Radiation resistance",
      "Turning tiny",
      "Radiation blast"
    ]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
  }
]
```

JavaScript Arrays

Learn >Ls.js

Arrays

- An array is an ordered collection of values. JavaScript arrays are object
- Each value is called an element, and each element has a numeric position in the array, known as its index (zero based index

เก็บ collection ที่ต่าง type กันได้

- JavaScript arrays are untyped : an array element may be of any type , and different elements of the same array may be of different types
- Array elements may even be objects or other arrays , which allows you to create complex data structures such as arrays of objects and arrays of arrays.
- JavaScript arrays are dynamic : they grow or shrink as needed , and there is no need to declare a fixed size for the array when you create it or to reallocate it when the size changes.
- Every JavaScript array has a length property.

Creating Arrays //05_Arrays/script2.js

1. Array literals
2. The ... spread operator on an iterable object
3. The Array() constructor
4. The Array.of () and Array.from () factory methods

1. Array literals

- The simplest way to create an array is with an array literal, which is simply a comma separated list of array elements within square brackets

//05_Arrays/script1.js

```
let arr1 = [10, 'in progress', true];

let arr2=  [15 ,30 ,42];

let students = [
  { id: 1, name: 'Ann'  },
  { id: 2, name: 'Peter' },
  { id: 3, name: 'Mary' }
];

let colors = [
  ['pink', 'red'],
  ['yellow', 'orange', 'brown']
];
```

2. The ... spread operator on an iterable object

- In ES6 and later, you can use the "spread operator," ..., to include the elements of one array within an array literal:

```
let a = [1, 2, 3];
let b = [0, ...a, 4];   // b == [0, 1, 2, 3, 4]
```

- The three dots "spread" the array so that its elements become elements within the array literal that is being created.
- The spread operator is a convenient way to create a (shallow) copy of an array:

//05_Arrays/script2.js

```
let c = [5, 10, 15];
let d = [...c];
d[0] = 10;
console.log(`d: ${d}`); //d: 10,10,15
console.log(`c[0]: ${c[0]}`); //5
console.log(`d[0]: ${d[0]}`); //10
```

- Modifying the copy does not change the original

3. The Array() Constructor

- Call it with **no arguments**:

  `let a = new Array();`

- Call it with a single numeric argument, which **specifies a length**:

  `let a = new Array(10);`

- Explicitly **specify two or more array elements** or **a single non-numeric element for the array**:

  `let a = new Array(3, 2, 1, "testing");`

4. The Array.of() and Array.from() factory methods

- The Array() constructor cannot be used to create an array with a single numeric element.
- In ES6, the Array.of() function addresses this problem: it is a factory method that creates and returns a new array, using its argument values (regardless of how many of them there are) as the array elements:

//05_Arrays/script2.js

```
Array.of()        // => []; returns empty array with no arguments
Array.of(5)       // => [5]; create arrays with a single numeric argument
Array.of(1,2,3)   // => [1, 2, 3]
```

- Array.from is another array factory method introduced in ES 6.
- It expects an iterable or array like object as its first argument and returns a new array that contains the elements of that object.
- With an iterable argument, Array.from (iterable) works like the spread operator [... iterable] does. It is also a simple way to make a copy of an array:

//05_Arrays/script2.js

```
let j = Array.of(1, 2, 3);

let k = Array.from(j); //k: 1,2,3
```

Reading and Writing Array Elements

- You access an element of an array using the [ ] operator.
- An arbitrary expression that has a non negative integer value should be inside the brackets.
- You can use this syntax to both read and write the value of an element of an array. Thus, the following are all legal JavaScript statements:

```
let a = ["hello"];
let value = a[0];       // Read element 0
a[1] = 3.5;             // Write element 1
let i = 2;
a[i] = 3;               // Write element 2
a[i + 1] = "world";     // Write element 3
a[a[i]] = a[0];         // Read elements 0 and 3, write element 3
```

Adding and Deleting Array Elements

//05_Arrays/script4.js

```
let arrList = [];     // Start with an empty array.
arrList[0] = 10;   // add elements to it.
arrList[1] = 20; // add elements to it.
arrList[2] = 'ten'; // add elements to it.
delete arrList[1]; // delete element at index 1
arrList.length //length=3
```

```
//result
[ 10, <1 empty item>, 'ten' ]
```

Note that using delete on an array element does **not alter the length property** and **does not shift elements with higher indexes down to fill in the gap** that is left by the deleted property.

Iterating Arrays

- As of ES 6, the easiest way to loop through each of the elements of an array (or any iterable object) is with the for/of loop

//05_Arrays/script3.js

```
let letters = [...'Hello world']; //spread array of characters
let msg = '';
for (let ch of letters) {
  msg += ch + ', ';
}
console.log(msg);
```

//result

```
H, e, l, l, o,  , w, o, r, l, d,
```

destructuring assignment

การใช้ตัวแปร > 1 ตัว รับค่าหลายๆ ค่า

Learn > L6.js

- The **destructuring assignment** syntax is a JavaScript expression that makes it possible to **unpack values from arrays, or properties from objects, into distinct variables.**

Array

```
let a, b, rest;
[a, b] = [5, 10];

console.log(a); //5
console.log(b); //10

[a, b, ...rest] = [5, 10, 15, 20, 25];
console.log(rest); // [15,20,25]
```

object

//05_Arrays/script5.js

ชื่อตัวแปรต้องเหมือนกับ property

```
({ a, b } = { a: 10, b: 20 });
console.log(a); // 10
console.log(b); // 20

({ a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 });
console.log(a); // 10
console.log(b); // 20
console.log(rest); // {c: 30, d: 40}
```

W7

Iterating Arrays (with index of each array element)

- If you want to use **a for/of loop** for an array and **need to know the index** of each array element, use the entries() method of the array, along with destructuring assignment.

spread operator กลายเป็น string array

//05_Arrays/script3.js

```
let letters = [...'Hello world'];
let value = '';
for (let [index, letter] of letters.entries()) {
  if (index % 2 === 0) value += letter; // letters at even indexes
}
console.log(`value: ${value}`); // "Hlowrd"
```

return key, value

1 ตัวแรก 1 ตัว

The **entries()** method returns a new **Array Iterator** object that contains the key/value pairs for each index in the array

Array Methods

Array Iterator Methods: Iterator methods loop over the elements of an array

เป็นกลุ่ม method ที่ทำอะไรสักอย่างกับ element แต่ละ element ใน array ที่ส่งเข้าไป visite

- **forEach()** iterates through an array, invoking a function you specify for each element — no return
- **map()** passes each element of the array on which it is invoked to the function you specify and returns an array containing the values returned by your function. — return array
- **filter()** returns an array containing a subset of the elements of the array on which it is invoked.
- **find()** returns the matching element, If no matching element is found, find() returns undefined
- **findIndex()** returns the index of the matching element. If no matching element is found, If no matching element is found, find() returns -1
- **every() and some()** they apply a predicate function you specify to the elements of the array, then return true or false.
- **reduce()** combine the elements of an array, using the function you specify, to produce a single value.

เพิ่ม/ลดที่ท้าย array เข้าก่อนออกทีหลัง

Stack and queue methods add and remove array elements to and from the beginning and the end of an array.

- **push()** appends one or more new elements to the end of an array and returns the new length of the array.
  pop() deletes the last element of an array, decrements the array length, and returns the value that it removed.
- **unshift()** adds an element or elements to the beginning of the array, shifts the existing array elements up to higher indexes to make room, and returns the new length of the array.
- **shift()** removes and returns the first element of the array, shifting all subsequent elements down one place to occupy the newly vacant space at the start of the array.

ตัดบางส่วน/ช่วง/ตอน ของ array เพิ่ม ลด แก้ไข บางส่วน

Subarray methods are for extracting, deleting, inserting, filling, and copying contiguous regions of a larger array.

- **slice()** returns a slice, or subarray, of the specified array. Its two arguments specify the start and end of the slice to be returned.
- **splice()** a general-purpose method for inserting or removing elements from an array.
- **fill()** sets the elements of an array, or a slice of an array, to a specified value. It mutates the array it is called on, and also returns the modified array:

หา index

Searching and sorting methods are for locating elements within an array and for sorting the elements of an array.

- **indexOf()** search an array for an element with a specified value and return the index of the first such element found, or -1 if none is found.
- **includes()** takes a single argument and returns true if the array contains that value or false otherwise. It does not tell you the index of the value, only whether it exists.
- **sort()** sorts the elements of an array in place and returns the sorted array.
- **reverse()** reverses the order of the elements of an array and returns the reversed array.

Array to String Conversions

แยบ element ใน array เป็น string

- **join()** converts all the elements of an array to strings and concatenates them, returning the resulting string.

Function Expressions

Function expression

```
const getRectangleArea = function(width, height) {
        return width * height;
};
```

Named function expression

```
let fact = function factorial(n) {
                console.log(n) ;
                if (n <= 1) {
                  return 1;
                }
               return n * factorial(n - 1);
          };

fact (5); //120
```

Arrow Function Expressions - is a compact alternative to a traditional function expression but is limited and can't be used in all situations.

```
// Traditional Function (no arguments)
let a = 4;
let b = 2;
function (){
    return a + b + 100;
}

// Arrow Function
let a = 4;
let b = 2;
() => a + b + 100;
```

```
//No param
() => expression

// One param
param => expression

// Multiple param
(param1, paramN) => expression

// Multiline statements
param1 => {
        statement1;
        …
        statementN;
}

(param1, paramN) => {
        statement1;
        …
        statementN;
}
```

Comparing traditional functions to arrow functions

```
// Traditional Function (one argument)
function (a){
    return a + 100;
}
// Arrow Function Break Down
// 1. Remove the word "function" and place arrow
between the argument and opening body bracket
(a)=> {
    return a + 100;
}

// 2. Remove the body braces and word "return" --
the return is implied.
(a)=> a + 100;

// 3. Remove the argument parentheses
a => a + 100;
```

```
// Traditional Function (multiple arguments)
function (a, b){
        return a + b + 100;
}

// Arrow Function
(a, b) => a + b + 100;
```

```
// Traditional Function (multiline statements)
function (a, b){
        let chuck=42;
        return a + b + chuck;
}

// Arrow Function
(a, b) => {
        let chuck=42;
        a + b + chuck;
}
```

Functions - object

└ sub program ไม่ต้องเขียน code ซ้ำ เรียกใช้ได้เลย

- A function is a **block of JavaScript code that is defined once** but may be **executed**, or invoked, **any number of times**.
- **JavaScript functions are parameterized:** a function definition may include a list of identifiers, known as parameters, that work as local variables for the body of the function. local parameter
- If a **function** is assigned to **a property of an object**, it is **known as a method** of that object.
- When a function is invoked on or through an object, that object is the invocation context or thisvalue for the function.
- Functions designed to initialize a **newly created object** are called **constructors**.
- In JavaScript, **functions are objects**, and they can be manipulated by programs. JavaScript can **assign functions to variables and pass them to other functions** function = data => first class sitizen

Function Declarations = function มีชื่อ

└ ประกาศ function

① เรียกใช้ได้กับตัวแปรใดๆก็ได้ ต่างจากตัวกำหนดค่าใช้ได้กับตัวแปร
② เรียกถูกส่งไปเป็น parameter ได้
③ เรียก return จาก function ใดๆก็ได้

- the function keyword
- the name of the
- a list of parameters to the function
- the JavaScript statements that define the function, enclosed in curly brackets, {...}.

ใส่/ไม่ใส่ก็ได้

```
function name([param1[, param2[, ..., paramN]]]) {
              statements

}
```

Function Expressions = function ที่ไม่มีชื่อ but เรียกใช้ผ่านตัวแปรที่เก็บfunctionนั้น

└ function ถูกเขียนเป็นนิพจน์ได้

Function expressions look a lot like function declarations, but they appear within the context of a larger expression or statement, and the name is optional. However, a name can be provided with a function expression.

Function expression: function expression defines a function and assign it to a variable

```
//functions as Values
const getRectangleArea = function(width, height) {
        return width * height;
};
```

นิพจน์

assignment statement variable = expression

Named function expression: Function expressions can include names, which is useful for recursion.

└ function ที่มีชื่อ ประโยชน์การเรียกตัวเอง

optional ใส่/ไม่ใส่ก็ได้

```
//functions as Values
let fact = function factorial(n) {
                    console.log(n) ;
                    if (n <= 1) {
                      return 1;
                    }
                    return n * factorial(n - 1);
                } ;
fact (5); //120
```

Calling Functions

- การกำหนดฟังก์ชันไม่ดำเนินการ กำหนดชื่อฟังก์ชันและระบุว่าต้องทำอย่างไรเมื่อเรียกใช้ฟังก์ชัน
- การเรียกใช้ฟังก์ชันจะเป็นการดำเนินการตามที่ระบุจริงโดยใช้พารามิเตอร์ที่ระบุ

- *Defining* a function does not *execute* it. Defining it names the function and specifies what to do when the function is called.
- Calling the function actually performs the specified actions with the indicated parameters.

//06_Functions/script1.js

```
//function declaration
function square (side) {
      return side * side;
}

square(3); //calling functions
```

```
//function expression
let area=function square(){
      return side* side;
}
area(3); //calling functions
```

Functions must be *in scope* when they are called, but the function declaration can be hoisted:

เรียกก่อนหรือหลังก็ได้

เรียกใช้ function ก่อน but สร้าง function ทีหลัง

```
square(3); //hoisting
function square (side) {
      return side * side;
}
```

แม้ว่าจะไม่ได้เขียนเป็นตัวแปร var

* let const ใช้ hoisted ไม่ได้

function hoisting only works with function *declarations* —not with function *expressions*.

ใช้ syntax ไหนก็ได้ แล้วแต่

Arrow Function expressions is a compact alternative to a traditional function expression but is limited and can't be used in all situations.

```
// Traditional Function (no arguments)
let a = 4;
let b = 2;
function (){
    return a + b + 100;
}

// Arrow Function
let a = 4;
let b = 2;
() => a + b + 100;
```

One param ไม่จำเป็นต้องใส่ค่าว่า return

เขียนย่อ function

เหมือนกัน

param => exp

```
//No param ไม่มี parameter
() => expression
                ได้ค่าเดียวในการประมวลผล func
// One param มี 1 parameter เอา () ออกได้เลย
param => expression   ไม่จำเป็นต้องใส่ return

// Multiple param มีหลาย parameter ต้องมี ()
(param1, paramN) => expression

// Multiline statements มีหลายคำสั่งให้ทำ
param1 => {                เพิ่ม {} ครอบด้วย
        statement1;
        …                  มี return
        statementN;
}

(param1, paramN) => {
        statement1;
        …
        statementN;
}
```

Comparing traditional functions to arrow functions

```
// Traditional Function (one argument)
function  (a){
    return a + 100;
}
// Arrow Function Break Down
// 1. Remove the word "function" and place arrow
between the argument and opening body bracket
(a)=> {
    return a + 100;
}

// 2. Remove the body braces and word "return" --
the return is implied.
(a)=> a + 100;

// 3. Remove the argument parentheses
a => a + 100;
```

parameter return

```
// Traditional Function (multiple arguments)
function (a, b){
        return a + b + 100;
}

// Arrow Function
(a, b) => a + b + 100;
```

```
// Traditional Function (multiline statements)
function (a, b){
        let chuck=42;
        return a + b + chuck;
}

// Arrow Function   มีหลาย statement ต้องระบุ return ด้วย
(a, b) => {                function จะได้รู้ว่าต้อง return ค่าไหน
        let chuck=42;
        return a + b + chuck;
}
```

ต้องใส่ return

Primitive Parameter Passing

→ ให้รู้ว่าเวลาส่ง primitive / ส่ง obj. ไป bec. ดูเรื่อง size effect

assign แอกซ์ parameter ไปเป็น formal parameter

Primitive parameters are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.

```
function square (side) {
      return side * side;
}
```

Object Parameter Passing

Object parameter (i.e., a non-primitive value, such as Array or a user-defined object) are passed to function and the function changes the object's properties, **that change is visible** outside the function.

```
function myFunc(theObject) {
  theObject.model = "A9999";
}
let product = {model: "A1001", price: 199};
console.log(mycar.model); // "A1001"

myFunc(mycar);
console.log(mycar.model); // "A9999"
```

Higher-Order Functions

function ที่รับ function มาเป็น parameter / return เป็น function

A "higher-order function" is a function that accepts functions as parameters and/or returns a function.

JavaScript Functions are **first-class citizens**

- be assigned to variables (and treated as a value)
- be passed as an argument of another function
- be returned as a value from another function

```
//1. store functions in variables

function add(n1, n2) {
  return n1 + n2
}
let sum = add

let addResult1 = add(10, 20)
let addResult2 = sum(10, 20)

console.log(`add result1: ${addResult1}`)
console.log(`add result2: ${addResult2}`)
```

//01_BasicJS/ script2.js

```
//2. returned as a value from another function

function operator(n1, n2, fn) {
  return fn(n1, n2)
}

//3. Passing a function to another function
function multiply(n1, n2) {
  return n1 * n2
}

let addResult3 = operator(5, 3, add)
let multiplyResult = operator(5, 3, multiply)

console.log(`add result3 : ${addResult3}`)
console.log(`multiply result: ${multiplyResult}`)
```

final

Function scope

- **Variables defined inside a function** cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.
- However, **a function can access all variables and functions defined inside the scope in which it is defined.**
- In other words, **a function defined in the global scope can access all variables defined in the global scope.**
- **A function defined inside another function** can also **access all variables defined in its parent function,** and **any other variables to which the parent function has access.**

Function scope and Nested Functions

//06_Functions/script2.js

```
// The following let variables are defined in the global scope
let mid = 20;
let final = 5;

let fname = 'Ada';

// sum function is defined in the global scope
function sum() {
  return mid + final;
}

console.log(`#1 sum: ${sum()}`); // Returns 25
mid = 10;
console.log(`#2 sum: ${sum()}`); // Returns 15
```

```
function getScore() {
  let mid = 10;
  let final = 30;

  //yourScore is nested function
  function yourScore() {
    return fname + ' scored ' + (mid + final);
  }
  return yourScore();
}
console.log(getScore()); // Returns "Ada scored 40"
```

Nested Functions and closures

- You may nest a function within another function. The nested (inner) function is private to its containing (outer) function.
- **A closure is an expression (most commonly, a function)** that can have free variables together with an environment that binds those variables (that "close" the expression).
- Since a **nested function is a closure**, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

Closures

- Closures are one of the most powerful features of JavaScript.
- JavaScript allows for **the nesting of functions** and grants the inner function **full access to all the variables and functions defined inside the outer function** (and all other variables and functions that the outer function has access to).
- However, **the outer function does not have access** to the variables and functions defined **inside the inner function**. This provides a sort of encapsulation for the variables of the inner function.

//06_Functions/script3.js

```
let getScoringPass = function (scores) {
  //bind and store "scores" argument to use in the nested "cuttingPoint" function
  function cuttingPoint(cuttingScore) {
    return scores.filter((score) => score >= cuttingScore);
  }
  return cuttingPoint;
};
//fn_cuttingPoint1 and fn_cuttingPoint2 are instance closure functions
//that bind to each their outer parameter "scores"
let fn_cuttingPoint1 = getScoringPass([50, 15, 32, 80, 100]);
console.log(fn_cuttingPoint1(50)); //[ 50, 80, 100 ]
let fn_cuttingPoint2 = getScoringPass([-10, -15, -53, -97, -32]);
console.log(fn_cuttingPoint2(-30)); //[ -10, -15 ]
```

ไม่ใช้ array แต่ มีคุณสมบัติเหมือน array

Using the arguments object

: เป็นการเรียก array ของ parameter ที่ส่งเข้ามาใน function ทั้งหมด

ไม่ใช่ array

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows: `arguments[i]`

where i is the ordinal number of the argument, starting at arguments [0]. The total number of arguments is indicated by arguments.length.

Array-like" means that arguments has a length property and properties indexed from zero, but it doesn't have Array's built-in methods like forEach() or map().

//06_Functions/script4.js

```
function printStudents(students) {
  let result = '';
  // iterate through arguments
  let separator = arguments[0];
  for (i = 1; i < arguments.length; i++) {
    result += arguments[i] + separator;
  }
  return result;
}
console.log(printStudents('.', 'Adam', 'John', 'Danai'));
//Adam.John.Danai.
```

Function Parameters

Starting with ECMAScript 2015, there are two new kinds of parameters:

- default parameters
- rest parameters

Default Parameters : กำหนดค่า default parameter (if ไม่ได้ใส่ใน parameter ตรงนี้ ก.ใช้ค่า default)

- In JavaScript, **parameters of functions default to undefined**.
- In the past, the general strategy for setting defaults was to test parameter values in the body of the function and assign a value if they are undefined.
- With **default parameters**, a manual check in the function body is no longer necessary. You can put the default value for any parameters in the function head

//06_Functions/script5.js

```
//default parameter
function who(name = 'unknown') {
  return name;
}
console.log(who()); //unknown
console.log(who('Umaporn')); //Umaporn
```

Rest Parameters

- **Rest parameters** allow us to write functions that can be invoked with an indefinite number of arguments as **an array**
- Rest parameters are Array instances 1 ตรงใช้ได้แค่ตัวเดียวเท่านั้น
- **Only the last parameter** in a function definition can be a **rest parameter**

```
//rest parameters          รับค่าทั้งหมดที่อยู่ในวงเล็บ
function sum(opName, ...theNumbers) {
  console.log(opsName); //'sum'
  return theNumbers.reduce((previous, current) => {
    return previous + current;
  });
}

console.log(sum('sum', 1, 2, 3)); //6
console.log(sum('sum', 1, 2, 3, 4, 5)); //15
```

Spread Parameters

Spread operator takes the array of parameters and spreads them across the arguments in the function call.

```
function sum(num1, num2, num3) {
  return num1 + num2 + num3;
}
let nums = [5, 20, 15];
//spread parameter
console.log(sum(...nums)); //40
```

Destructuring Function Arguments into Parameters

- If you define a function that has parameter names **within square brackets**, you are telling the **function to expect an array value** to be passed for each pair of square brackets.
- As part of the invocation process, **the array arguments will be unpacked into the individually named parameters.**

```
//destructuring function arguments into parameters
function arrayAdd1(v1, v2) {
  return [v1[0] + v2[0], v1[1] + v2[1]];
}
console.log(arrayAdd1([1, 2], [3, 4])); // [4,6]

function arrayAdd2([x1, y1], [x2, y2]) {
  // Unpack 2 arguments into 4 parameters
  return [x1 + x2, y1 + y2];
}
console.log(arrayAdd2([1, 2], [3, 4])); // [4,6]
```

final

JavaScript Export

- The **export** statement is used when creating JavaScript modules to export live bindings to *functions*, *objects*, or *primitive values* from the module so *they can be used by other programs with the* ***import*** *statement.*
- There are **two types** of exports:
  - Named Exports (Zero or more exports per module)
  - Default Exports (One per module)

Module Export

```
// Exporting individual features
export let name1, name2, …, nameN; // also var, const
export let name1 = …, name2 = …, …, nameN; // also var, const
export function functionName(){...}
export class ClassName {...}
```

```
// Export list export { name1, name2, …, nameN };
// Renaming exports export { variable1 as name1, variable2 as name2, …, nameN };
```

```
// Default exports
export default expression;
export default function (…) { … } // also class, function
export default function name1(…) { … } // also class, function
export { name1 as default, … };
```

Module Import

The import statement cannot be used in embedded scripts unless such script has a type="module".

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
import { export1 , export2 } from "module-name";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export1 [ , [...] ] } from "module-name";
```

module-name: The module to import from. This is often a relative or absolute path name to the **.js** file containing the module.

```
//dataFuncExport.js
//named export
export const frontEndFramework = ['Vuejs', 'React', 'Angular']
//or
const frontEndFramework = ['Vuejs', 'React', 'Angular']
export { frontEndFramework }

export function greeting() {
  return 'Hello, function from another module'
}
//default export
export default function getInstructor() {
  return `Umaporn Supasitthimethee`;
}
```

```
<!--index.html -- > //run on browser
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>

</head>
<body>
  <script src="./main.js" type="module"></script>

</body>
</html>
```

//06_Functions/ExportImportModules/script.js

```
//subjectExport.js
const subject = 'INT201'
export {subject}
```

```
//main.js
import defaultExport, {greeting, frontEndFramework as frontEnd} from './dataFuncExport.js';

import {subject} from './subjectExport.js'

console.log(`Frontend Framework: ${frontEnd}`)
console.log(greeting())
console.log(defaultExport);
console.log(subject)
```