Table of Contents

[**1** **SMALL THINGS TO LEARN** 2](#_Toc518046371)

[1.1 JS Skills: debugging, testing and performance analysis: 2](#_Toc518046372)

[1.2 What is Functional Programming? 2](#_Toc518046373)

[1.3 Window console method 2](#_Toc518046374)

[1.4 Events are synchronous: 2](#_Toc518046375)

[1.5 Parameters and Arguments in Function: 2](#_Toc518046376)

[1.6 `this` keyword & getter-setter ES6: 3](#_Toc518046377)

[1.7 Constructor function ES6: 3](#_Toc518046378)

[1.8 Scope and Context: 3](#_Toc518046379)

[1.9 Prototype – Prototype Chains – Inheritance: 4](#_Toc518046380)

[**2** **BIG THINGS TO LEARN** 5](#_Toc518046381)

[2.1 TESTING & DEBUGGING 5](#_Toc518046382)

[2.2 FUNCTION IS FUNDAMENTAL 5](#_Toc518046383)

[2.2.1 Functions as First-class Objects: 5](#_Toc518046384)

[2.2.2 Types of function: 6](#_Toc518046385)

[2.2.3 Invoking Functions: 7](#_Toc518046386)

[2.3 INDENTIFIER RESOLUTION 11](#_Toc518046387)

[2.3.1 Stack: 12](#_Toc518046388)

[2.3.2 Lexical Environment: 12](#_Toc518046389)

[2.4 CLOSURES 13](#_Toc518046390)

[2.5 ASYNCHRONOUS ES17 13](#_Toc518046391)

[2.5.1 Promise: 13](#_Toc518046392)

[2.5.2 Async: 13](#_Toc518046393)

[2.5.3 Await: 13](#_Toc518046394)

# **SMALL THINGS TO LEARN**

## JS Skills: debugging, testing and performance analysis:

* JS Scope and Hoist
* Functions - Closures
* Prototypes – Objects
* Features: RegEx, promises,…

## What is Functional Programming?

* A style of programming that focuses on solving problems by composing functions (instead of sequences of steps – **this style is Imperative Programming**)
* This style is easier to **test**, **modularize**, **extend(?)**

## Window console method 😊

* console.log()
* console.dir()
* console.time()
* console.timeEnd()
* console.assert

## Events are synchronous:

1. Browser events: (*e.g. page to be loading or finishes loading*)
2. Network events: (*e.g. responses from server (e.g. XMLHttpRequest)*)
3. User events: (*e.g. click, mousemove, keypress,…*)
4. Time events: (*e.g. setTimeout(), setInterval()*)

## Parameters and Arguments in Function:

* They are quite different.
* A parameter: is the **variable** listed as a part of function definition
* An argument: is the **object** value that passed against parameter in the function when function is **invoked**.

## `this` keyword & getter-setter ES6:

*const obj = {*

*title: ‘obj’,*

*method: function(){*

*return this.title;*

*}*

*}*

*const obj1 = {*

*title: ‘obj1’,*

***method()****{*

*return this.title;*

*}*

*}*

*const obj2 = {*

*title: ‘obj2’,*

***get*** *getTitle(){*

*return this.title;*

*}*

*}*

*const obj3 = {*

*title: ‘obj3’,*

***set*** *setTitle(newTitle){*

*if(typeof newTitle == ‘string’){*

*this.title = newTitle;*

*}*

*}*

## Constructor function ES6:

* **Constructor function ES6:**

**1st way:**

*function* ***Person(name, age)*** *{*

***this.****name = name;*

***this.****age = age;*

*}*

*const myMom =* ***new*** *Person(‘Khanh’, 21);*

**2nd way (not always optimal):**

*const newObj =* ***Object.create(****null****)***

**3rd way:**

*const newObj =* ***{}***

## Scope and Context:

* Scope is function-based while Context is object-based.
* Scope related to the variable access of a function when invoked and it is unique
* Context is **always** the value of **`this`** keyword which is a reference to the object that **owns** executing function.

## Prototype – Prototype Chains – Inheritance:

* **Object:** all objects have ***a*** ***constructor***and ***a prototype***is a built-in property of constructor that contains constructor as its property by default. *(e.g. const anObj = {}; anObj.constructor)*
* **Constructor:** is a function
* **Prototype:** is an object, has properties and methods

+ Many objects may share the same prototype

*function* ***Person(name, age)*** *{*

***this.****name = name;*

***this.****age = age;*

*}*

*const myMom =* ***new*** *Person(‘Khanh’, 21);*

*Object.getPrototypeof(obj);*

*Object.setPrototypeof(obj);*

# **BIG THINGS TO LEARN**

## TESTING & DEBUGGING

* Good tests have 3 things: **Repeatability, Simplicity** and **Independence** (break tests into smallest units)
* Type of tests:

**+** **Deconstructive:** deconstruct existing code into smaller case to reproduce the bug.

**+ Constructive:** start from small case and build up until reproduce the bug.

* Some popular testing frameworks:

**+ QUnit:** testing for jQuery, API-based unit-testing framework

**+ Jasmine:**

* + *describe()*: describe test case
  + *it()*: specify individual test
  + *expect()*: check individual assertion
* A test suite has 80% code coverage means: 80% code does not have bugs

## FUNCTION IS FUNDAMENTAL

* In JS, function is primary modular unit of execution.

### Functions as First-class Objects:

|  |  |  |
| --- | --- | --- |
| Characteristics | Object | Function |
| 0. An instance of Object | true | true |
| 1. can be created via literals | var ninja = {} | function ninjaFunc(){} |
| 2. Store as a data (*e.g. variable, array, object property,…*) | var ninja = [] OR var ninja = {}  ninjaArr.push({})  ninja.person = {  name: ‘Kiko’,  age: 19} | var ninja = function () {}  ninjaArr.push(function(){})  ninjaArr.sayHi = function(){  return “Hi”;} |
| 3. Can be passed as argument to a function | function sayHiNinja(ninja){  return “Hi, “ + ninja.name;}  sayHiNinja({name: ‘Kiko’, age: 19}) | function callbackSayHi(**sayHi**){  return **sayHi()**;}  function sayHi(){  console.log(“Hi :D”);}  callbackSayHi(sayHi); |
| 4. Can be returned as value from a function. | function newNinja(name, age){  return {name, age};}  newNinja(“Min”, 28); | [example above]  …  function callbackSayHi(**sayHi**){  return **sayHi**;}  callbackSayHi(sayHi); |
| 5. Have properties and a link back to its construction method(?) | var ninja = {};  ninja.name = “Kiko”; | var sayHi = function(){  return “Hi :D”;}  sayHi.name = “Khanh”; |

### Types of function:

* **Function declarations** and **function expressions** (1)
* **Self-invoked functions** (2)
* **Arrow functions** (3)
* **Generator functions** (5)
* **Function constructors:** create a new function by strings
* **Constructor function:** create object instances

Definitions:

(1a) Function Declarations:

* Define a function by not assigning to a variable but declare “function” itself
* Declare a function in another function
* Function Declaration is interpreted by JS program even **before** we execute JS program. It brings this type of function to the top of its current scope.
* This type of function **wastes** RAM memory and badly affects mobile.

(1b) Function Expression:

* Define a function by assigning it to a **larger expression syntax** (e.g. variable)

**!HUOM: Hoisting**

[To quote Ben Cherry’s excellent article: “Function declarations and function variables are always moved (‘hoisted’) to the top of their JavaScript scope by the JavaScript interpreter”.](https://javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions/?blogsub=confirming#subscribe-blog)

**Applied** for:

* Function Declarations
* Variable Declarations

But **not**: Assignment Expression of its Function Definition or any other Variables

(2)



(3) aka **Lambda function** (anonymous function + arrow function)

\* When not to use arrow function instead of function expression:

<https://wesbos.com/arrow-function-no-no/>

**!HUOM: Hoisting**

* Argument Object is an **alias** with Function Parameter.

*function person(name){*

*console.log(name == arguments[0]);*

*name = “Kiko”;*

*console.log(arguments[0] === “Kiko”);*

*arguments[0] = “Messi”;*

*console.log(name === “Messi”);*

*}*

* To **avoid** alias, “use strict”:
* **Avoid** using **.length** as it is not JS array although can be looped.

### Invoking Functions:

4 different ways:

* Function is invoked explicitly (1)
* A method of an object, in Object-Oriented Programming (2)
* A constructor generates new object (3)
* Via its ***apply***, ***call*** or ***bind*** (most complicated) (4)

***function sayHi(name){ return “Hi, “ + name;}***

(1) Aka­: Invocation as a function

*sayHi(“Khanh”)* ***🡺 Standard function call***

*(function(name){return “Hi, “ + name;})(“Khanh”)* ***🡺 Self-invoked function***

* **`this` refers to: *window* in strict mode, *undefined* in non-strict mode**

(2) Aka: Invocation as a method

*const myMom = {* ***🡺 myMom.sayHi(“Joonas”)***

*name: “Nga”,*

*favoriteSaying: sayHi(name)*

*}*

***More advance example:***

*const obj = {*

*names: [“a”, “b”,”c”],*

*num: 5,*

*logger() {*

*this.names.forEach(function(name){*

*console.log(****this****); //window 🡺 this.num === undefined*

*//print each name with num*

*})*

*}*

*}*

* **`this` refers to: *the closest-outer object***

(3) Aka: Invocation as a constructor

*function Person(){ return this;};*

***new*** *Person();*

* ***`new` keyword invokes constructor function. A new empty object instance will be created and passed to the function as this.***
* **`this` refers to: *the newly-created object***



**!HUOM: Constructor function returns values**

**a) Constructors return primitive values:**

*function Ninja(){*

*this.method = function() { return true; };*

*return 1;*

*}*

*Ninja() 🡺* ***Output: 1***

*const ninja = new Ninja();* ***🡺 Output: Ninja {f}***

*typeof ninja === “****object****”*

*typeof ninja.method === “****function****”*

*typeof ninja.method() === “****boolean****”*

**b) Constructors explicitly return object values:**

*let pupu = {*

*eat: “carrots”*

*}*

*function Animal(){*

*this.drink =”water”;*

*return pupu;*

*}*

*Animal() 🡺* ***Output: {eat: “carrots”}***

*const dog = new Animal();* ***🡺 Output: {eat: “carrots”}***

*dog.drink* ***🡺 Output: undefined***

**Conclusion: if constructor returns an object, later will be returned as the value of `new` while newly-constructed object passed to `this` is discarded.**

**However, if constructor returns non-object, it will be ignored.**

(4) Aka: Invocation with the **apply** and **call**

- Invoke a function by explicitly specifying an object that we want as a function context.

- Use `apply` or `call`, set that new object as a function context.

*function countSum(){*

*let result = 0;*

*for(let item of arguments){*

*result += item;*

*}*

*this.result = result;*

*}*

*const gradeKhanh = {};*

*countSum.call(gradeKhanh, 1, 2, 3);*

*const gradeJoonas = {};*

*countSum.apply(gradeJoonas, [5, 10]);*

*assert(gradeKhanh.result === 6, "countSum via `call`");*

*assert(gradeJoonas.result === 15, "countSum via `apply`");*

* **`this` refers to: *first parameter of apply/call/bind***

(5) Generator function

- used for ID GENERATOR

- it generates a sequence of values, which can only be explicitly called once per request. As a result, it returns **individual value**

- after its execution, it doesn’t end as other normal function but **suspended**.

*function\* produceFood(){*

*yield “Egg”;*

*yield “Fish”;*

*yield “Potato”;*

*}*

*for(let food of* ***produceFood()****){*

*console.log(food);}*

*const foodGenerator = produceFood();*

*foodGenerator.next()*

*foodGenerator.next()*

*foodGenerator.next()*

|  |
| --- |
| **Purposes** |
| 1. Yield to another generator, ***yield\**** 2. For Infinite Loops, whenever generator encounters ***yield***, it will ***suspend*** until ***next()*** 3. For Traversing DOM, |

|  |  |
| --- | --- |
| **STATE PHASES** | **Code Example** |
| 1. Create a new generator 🡺 *Suspended starts*  *const persons = PersonGenerator()*  2. Activate generator, returns a new object 🡺 *Executing, Suspended yield*  *persons.next()*  🡪 *{value:* ***‘Khanh’****, done:* ***false****}*  3. Reactivate generator, returns a new object 🡺 *Reactivate. Executing. Suspended yield*  *persons.next()*  🡪 *{value:* ***‘Dang****, done:* ***false****}*  4. Reactivate generator but returns a new object with empty value property (in default return mode) 🡺 *Reactivate. Executing. Completed*  🡪 (default) *{value:* ***undefined****, done:* ***true****}*  🡪 (without *comment*) *{value: ‘****Hihihihih****’, done:* ***true****}* | *function PersonGen() {*  *yield “Khanh”;*  *yield “Dang”;*  *//return “Hihihihihih”;*  *}* |

## INDENTIFIER RESOLUTION

* Two types of JS code: **global code** and **function code**
* Two types of execution context: **global execution context** and **function execution context**

### Stack:

* Stack keeps track of program execution.
* Fundamental data structure, putting new items/”outer function” on the top. When want to get existing item, need to go from the top.
* For example, applied for function in another function.

### Lexical Environment:

* Lexical Environment is a JS internal implementation of **keeping track** of the **mapping** from **identifiers** to **specific values**. (in other words, it works for JS scope: *function, block of code [if-else statement, try-catch statement],…*)
* Lexical Environment needs to keep track of its typical LE:
  + *Global*
  + *Function (outer, local)*
  + *Block*

… and type of variable (*let, const, var* and *function declaration* **(NOT *function expression and Lambdas*)***)*

* Program often access to identifiers in *outer LE*



var: defined in global LE and function scope (code of block is ignored)

const, let: defined in closest LE.

## CLOSURES

* Closure is a mechanism that allows a function to access to all variables

## ASYNCHRONOUS ES17

### Promise:

* Promise is a constructor with a built-in function (aka **executor** function). This function has **two** parameters (resolve, reject)

*const generatePromise = new Promise((resolve, reject) => resolve(“hi, Khanh”))*

* To use the promise, built-in method **.then** on Promise object will help. Pass **two** call-back functions (success, failure)

*generatePromise*

*.then(person => console.log(person),*

*err => fail(‘Oops’));*

### Async:

* Special syntax works with Promise.
* It returns a promise. If it returns <non-promise>, it automatically wraps the returned value into a **resolved promise**.

### Await:

* This keyword makes JS wait until the promises settles and returns its values.
* It only works with **async function**, not callback function.

|  |  |  |  |
| --- | --- | --- | --- |
| Code example | **CALLBACK** | **PROMISE** | **ASYNC-AWAIT** |
| ***Fetch API*** | $.getJSON(‘https://randomuser.me/api/’, function(data) {  console.log(data);  }) | fetch(‘https://randomuser.me/api/’)  .then(res => res.json())  .then(data => console.log(data))  .catch(err => console.log(err)) | async function fetchAsync() {  const res = await fetch(‘https://randomuser.me/api/’);  const data = await res.json();  **//(1) or (2)**  }  ***Method 1:***  console.log(data);  ***Method 2:***  return data;  fetchAsync()  .then (data => console.log(data))  .catch(err => console.log(err)) |
| ***Error handling*** | try {  $.getJSON(…);  }  catch (err) {  console.log(err);  } |  |  |