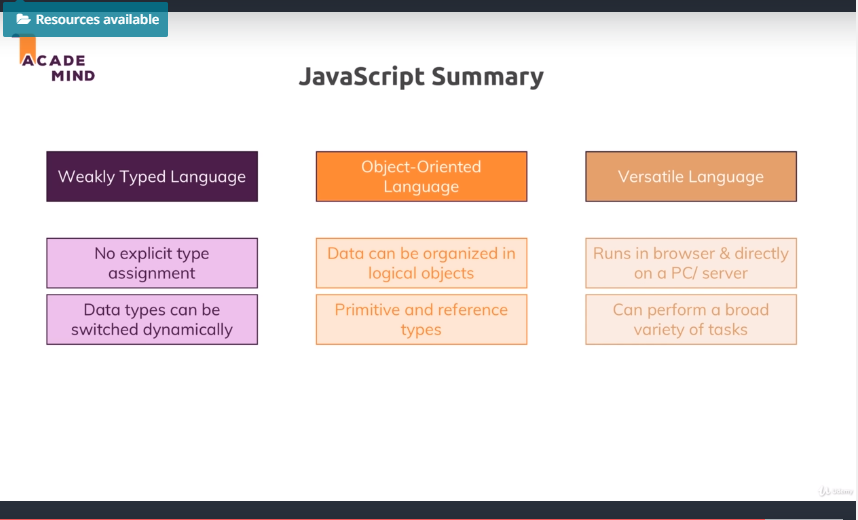
1. JavaScript



Main Reference : MDN (https://developer.mozilla.org/en-US/docs/Learn/JavaScript) and  <https://academind.com/learn/javascript>

1.Premitive VS Reference (Array, and Object)

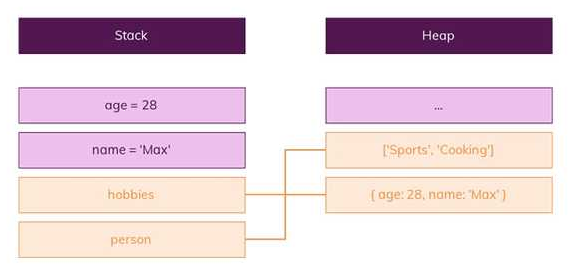
**Primitive -> Stack (in memory) : value stored in memory**

**Reference -> Heap (in memory) : pointer stored in memory**

1. Stack VS. Heap

The stack is essentially an easy-to-access memory that simply manages its items as a - well - stack. Only items for which the size is known in advance can go onto the stack. This is the case for numbers, strings, boolean.

The heap is a memory for items of which you can't pre-determine the exact size and structure. Since objects and arrays can be mutated and change at runtime, they have to go into the heap therefore. Hobbies and persons are pointers. The pointers get stored in the stack. And fields, values and elements are stored in heap.



What’s actually stored in the person variable in the following snippet?

**var person = { name: 'Max' }**

Is it: a) The object ({ name: 'Max' })

b) The pointer to the object

c) A pointer to the name property?

It’s **b)**. A pointer to the person object is stored in the variable. The same would be the case for the hobbies array.

# What are Reference Types Then?

So we learned what “Primitives” (or “primitive types”) are.

What are “reference types” then?

Objects and Arrays!

**var person = {**

**name: 'Max',**

**age: 28,**

**}**

**var hobbies = ['Sports', 'Cooking']**

Here, person is an object and therefore a so-called reference type. Please note that it holds properties that in turn have primitive values. This doesn’t affect the object being a reference type though. And you could of course also have nested objects or arrays inside the person object.

The hobbies array is also a reference type - in this case, it holds a list of strings. A string is a primitive value/ type as you learned but this doesn’t affect the array. Arrays are **always** reference types.

-----------------------------------------------------------------------------------------------------------

var person = { name: 'Max' };

// After copying person, we can update the field inside the object.

var newPerson = person;

newPerson.name = 'Anna';

console.log(person.name); // What does this line print? 🡺 Anna

Because you never copied the person object itself to newPerson. You only copied the pointer! It still points at the same address in memory though. Hence changing newPerson.name also changes person.name because newPerson points at the exactly same object! (그럼으로 이름만 달라지고 memory address는 여전히 같다.)

1. The way of copy for reference type
2. Slice for an array

var hobbies = ['Sports', 'Cooking'];

var copiedHobbies = hobbies.slice();

It basically returns a new array which contains all elements of the old element, starting at the starting index you passed (and then up to the max number of elements you defined). If you just call slice(), without arguments, you get a new array with all elements of the old array.

1. spread

var hobbies = ['Sports', 'Cooking'];

var copiedHobbies = [...hobbies];

1. Object.assgin()

var person = { name: 'Max' };

var copiedPerson = Object.assign({}, person);

1. Spread

var person = { name: 'Max' };

var copiedPerson = { ...person };

2. Deep Clone about an object \*\*\*\*\*

<https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns>

Defining a new variable does *not* create a new actual object - it only creates another reference to the same object. An example of this error would be:

[Immutable, not mutable ]

This function does correctly return a shallow copy of the top-level state object, but because the nestedState variable was still pointing at the existing object, the state was directly mutated.

**function** **updateNestedState**(state, action) {

**let** nestedState = state.nestedState

// ERROR: this directly modifies the existing object reference - don't do this!

nestedState.nestedField = action.data

**return** {

...state,

nestedState

}

}

Another common version of this error looks like this: Doing a shallow copy of the top level is not sufficient - the nestedState object should be copied as well. (object아의 object, 즉, nestedState는 여전히 old object의 같은 주소를 가리키고 있다.) 그럼으로 Spread를 써서 카피를 했더라도 nested object의 메모리 주소는 여전히 같다. Filed value가 변경 된다.

**function** **updateNestedState**(state, action) {

// Problem: this only does a shallow copy!

**let** newState = { ...state }

// ERROR: nestedState is still the same object!

newState.nestedState.nestedField = action.data

**return** newState

}

##### Correct Approach: Copying All Levels of Nested Data

Unfortunately, the process of correctly applying immutable updates to deeply nested state can easily become verbose and hard to read. Here's what an example of updating state.first.second[someId].fourth might look like:

**function** **updateVeryNestedField**(state, action) {

**return** {

...state,

first: {

...state.first,

second: {

...state.first.second,

[action.someId]: {

...state.first.second[action.someId],

fourth: action.someValue

}

}

}

}

}

## Inserting and Removing Items in Arrays

Normally, a Javascript array's contents are modified using mutative functions like push, unshift, and splice. Since we don't want to mutate state directly in reducers, those should normally be avoided. Because of that, you might see "insert" or "remove" behavior written like this:

**function** **insertItem**(array, action) {

**return** [

...array.slice(0, action.index),

action.item,

...array.slice(action.index)

]

}

**function** **removeItem**(array, action) {

**return** [...array.slice(0, action.index), ...array.slice(action.index + 1)]

}

**Copy**

However, remember that the key is that the original in-memory reference is not modified. **As long as we make a copy first, we can safely mutate the copy**. Note that this is true for both arrays and objects, but nested values still must be updated using the same rules.

This means that we could also write the insert and remove functions like this:

**function** **insertItem**(array, action) {

**let** newArray = array.slice()

newArray.splice(action.index, 0, action.item)

**return** newArray

}

**function** **removeItem**(array, action) {

**let** newArray = array.slice()

newArray.splice(action.index, 1)

**return** newArray

}

**Copy**

The remove function could also be implemented as:

**function** **removeItem**(array, action) {

**return** array.filter((item, index) => index !== action.index)

}

1. Arrow Function의 장점

function Prefixer(prefix) {

this.prefix = prefix;

}

Prefixer.prototype.prefixArray = function (arr) {

return arr.map(function (x) {

return this.prefix + ' ' + x;;

}.bind(this)); // this: Prefixer 생성자 함수의 인스턴스

};

var pre = new Prefixer('Hi');

console.log(pre.prefixArray(['Lee', 'Kim']));

* ["Hi Lee", "Hi Kim"]

1. When the arrow function is used

function Prefixer(prefix) {

this.prefix = prefix;

}

Prefixer.prototype.prefixArray = function (arr) {

return arr.map(x => `${this.prefix} ${x}`);

};

const pre2 = new Prefixer('Hi');

console.log(pre2.prefixArray(['Lee', 'Kim']));

* ["Hi Lee", "Hi Kim"]

1. Error example of Arrow Function
2. const obj = {
3. name: "han",
4. hi: () => console.log(`Hi ${this.name}`)
5. };
6. obj.hi(); // Hi undefined

const obj = {

name: "han",

hi(){

console.log(`Hi ${this.name}`);

}

};

obj.hi(); // Hi han

* Error (Undefined

const obj = {

name: 'Lee',

};

Object.prototype.hi = () => console.log(`Hi ${this.name}`);

obj.hi(); // Hi undefined

// Working case

Object.prototype.sayHi = function() {

console.log(`Hi ${this.name}`);

};

1. adEventListener 이 경우 this가 상위 컨택스트를 가리 킨다고 합니다. 고로 이전처럼 function(){}으로 사용하는 것이 옳은 표현입니다.

button.addEventListener('click', () => {

console.log(this === window); // => true

console.log(this === button); // => false

});

button.addEventListener('click', function() {

console.log(this === window); // => false

console.log(this === button); // => true

});

1. ‘this’ usage
2. Class

class MyClass {

constructor() {

const button = ...;

button.addEventListener('click', this.myMethod);

}

myMethod() { ... }

}

The this keyword is important here though. It basically points at the object that is created based on the class. And since myMethod is a method of the class/ object, it can only be accessed via this.

That's how you pass a reference to a function instead of calling it immediately. And that's why you would use this syntax without the parentheses.

1. addEventListener + class
2. class NameGenerator {
3. constructor() {
4. const btn = document.querySelector('button');
5. this.names = ['Max', 'Manu', 'Anna'];
6. this.currentName = 0;
7. btn.addEventListener('click', this.addName.bind(this));
8. }
9. addName() {

Console.log(this) // This is not //"whoever called the code in which it's being used" anymore because of ‘bind’. The reference is ‘btn’, now

1. const name = new NameField(this.names[this.currentName]);
2. this.currentName++;
3. if (this.currentName >= this.names.length) {
4. this.currentName = 0;
5. }
6. }
7. }