Web Application Development for Industrial Engineers

การพัฒนาเวปแอปพลิเคชันสำหรับวิศวกรอุตสาหการ

Looping code

- Print 10 random numbers

```
for (let i = 0; i < 10; i++) {
  console.log(Math.random()); // 10 randoms numbers
}
```

Looping through an array

```
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal', 'Lion'];

for (const cat of cats) {
  console.log(cat);
}
```

Looping through an object

```
const person = { age: 10, weight: 30 };

for (const entry of Object.entries(person)) {
  console.log(entry); // ['age', 10], ['weight', 30]
}
```

- Note that `Object.entries()` return an array.

Array methods: `forEach`

```
var numbers = [28, 77, 45, 99, 27];

numbers.forEach((item) => {
  console.log(item); // 28, 77, 45, 99, 27
});

numbers.slice(0, 3).forEach((item) => {
  console.log(item); // 28, 77, 45
});
```

Array methods: `map`

```
const names = ['Taylor', 'Donald', 'Don', 'Natasha', 'Bobby'];

const nameModified = names.map((name) => 'This is ' + name + '.');

console.log(nameModified);
/* [ 'This is Taylor.',
  'This is Donald.',
  'This is Don.',
  'This is Natasha.',
  'This is Bobby.' ]
 */
```

Array methods: `filter`

```
const numbers = [4, 11, 42, 14, 39];

const numberFilter = numbers.filter((item) => item > 12);

console.log(numberFilter); // [ 42, 14, 39 ]
```

- You can use this technique to *delete* items from array.

Array methods: `find`

```
const numbers = [4, 11, 42, 14, 39];

const numberFilter = numbers.find((item) => item > 12);

console.log(numberFilter); // 42
```

- Note that, in these examples, `filter` return an array while `find` return a number.

Destructuring

- Destructuring assignment is a syntax that allows you to assign object properties or array items as variables.
- This can greatly reduce the lines of code necessary to manipulate data in these structures.
- There are two types of destructuring: Object destructuring and Array destructuring.

Array destructuring

```
const foo = ['one', 'two', 'three'];

const [red, yellow, green] = foo;

console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
```

Object destructuring

```
const note = {
  id: 1,
  title: 'My first note',
  date: '01/01/1970',
};
// Destructure properties into variables
const { id, title, date } = note;

console.log(id); // 1
console.log(title); // "My first note"
console.log(date); // "01/01/1970"
```

Spread

Spread can simplify common tasks with arrays.

```
// Create an Array
const tools = ['hammer', 'screwdriver'];
const otherTools = ['wrench', 'saw'];

// Concatenate tools and otherTools together
const allTools1 = tools.concat(otherTools);

// Unpack the tools Array into the allTools Array
const allTools2 = [...tools, ...otherTools];

console.log(allTools1, allTools2);
// [ 'hammer', 'screwdriver', 'wrench', 'saw' ]
```

Spread with objects

```
const user = {
  id: 3,
  name: 'Ron',
};

const updatedUser = { ...user, isLoggedIn: true };

console.log(updatedUser); // { id: 3, name: 'Ron', isLoggedIn: true }
```

Spread with function call

```
function multiply(a, b, c) {
  return a * b * c;
}

const numbers = [1, 2, 3];

multiply(...numbers); // 6
```

Rest parameter

- Rest parameter syntax appears the same as spread (`...`) but has the opposite effect.
- Instead of unpacking an array or object into individual values, the rest syntax will create an array of an indefinite number of arguments.

```
function restTest(...args) {
  console.log(args); // [ 1, 2, 3, 4, 5, 6 ]
}

restTest(1, 2, 3, 4, 5, 6);
```

Rest parameters assignment (array)

```
const foo = ['one', 'two', 'three'];

const [red, ...rest] = foo;

console.log(red); // "one"
console.log(rest); // ["two", "three"]
```

Rest parameters assignment (object)

```
const note = {
  id: 1,
  title: 'My first note',
  date: '01/01/1970',
};

const { id, ...rest } = note;

console.log(id); // 1
console.log(rest); //{ title: 'My first note', date: '01/01/1970' }
```

Immutability (safe)

```
let a = 1;

function myFunc(a) {
  a = 2;
  console.log(a); // 2
}

myFunc(a);

console.log(a); // 1
```

Mutability (be careful)

```
let a = [1, 2, 3];

function myFunc(a) {
  a.push(4);
  console.log(a); // [1, 2, 3, 4]
}

myFunc(a);

console.log(a); // [1, 2, 3, 4]
```

Mutability (be careful)

```
let a = [1, 2, 3];

function myFunc(a) {
  let b = a;
  b.push(4);
  console.log(a); // [1, 2, 3, 4]
}

myFunc(a);

console.log(a); // [1, 2, 3, 4]
```

Mutability (be careful)

```
let a = [1, 2, 3];

function myFunc(a) {
  let b = a;
  return b;
}

let b = myFunc(a);
b.push(4);

console.log(b); // [1, 2, 3, 4]
console.log(a); // [1, 2, 3, 4]
```

Enforce "immutability" (safer)

```
let a = [1, 2, 3];

function myFunc(a) {
  let b = a.slice(0);
  b.push(4);
  console.log(b); // [1, 2, 3, 4]
}

myFunc(a);
console.log(a); // [1, 2, 3]
```

How to enforce "immutability"? (array)

- Try to use expression that gives a new copy of `a`.

```
let b = a.slice(0);
let b = [...a];
```

Mutable array methods (unsafe)

```
pop();
push();
shift();
unshift();
reverse();
sort();
splice();
```

How to enforce "immutability"? (object)

```
let b = { ...a };
let b = Object.assign({}, a);
```

Form

- https://ie-software-dev.netlify.app/Codes/T10_js/t01_form/