Blockchain for Industrial Engineers: Decentralized Application Development

บล็อกเชนสำหรับวิศวกรอุตสาหการ: การพัฒนาแอปพลิเคชันแบบกระจายศูนย์

Function

Functions allow you to:

- Store a piece of code that does a single task inside a defined *block*.
- Call that code whenever you need it using a single short command.

Example

The `replace()` string function

- Takes a source string and a target string
- Replaces the source string with the target string
- Returns the newly formed string

```
const myText = "I am a string";
const newString = myText.replace("string", "sausage");
console.log(newString);
```

Invoking functions

- Including the name of the function in the code somewhere, followed by parentheses.
- This form of creating a function is also known as function **declaration**.
  - It is always **hoisted**, so you can call function above function definition and it will work fine.

```
function myFunction() {
  alert("hello");
}

myFunction();
// calls the function once
```

Function parameters

- Values that need to be included inside the function parentheses.
- Make functions more useful.

```
function myFunction(text) {
  alert(text);
}

myFunction("hello");
```

Default parameters

- If you're writing a function and want to support optional parameters, you can specify default values by adding `=` after the name of the parameter, followed by the default value:

```
function hello(name = "Chris") {
  console.log(`Hello ${name}!`);
}

hello("Ari"); // Hello Ari!
hello(); // Hello Chris!
```

Return values

- Function can return values

```
function add(a, b) {
  return a + b;
}

const result = add(1, 2); // 3
```

Anonymous functions

```
function() {
  alert('hello');
}
```

- Has no name.
- Used when a function expects to receive another function as a parameter.

Anonymous function example (1)

```
function operate(a, b, ops) {
  return ops(a, b);
}

// Call the function with anonymous function as an argument.
const result = operate(1, 2, function (a, b) {
  return a + b;
});

console.log(result); // 3
```

Arrow functions

- If you pass an anonymous function like this, there's an alternative form you can use, called an *arrow function*.
- Instead of `function(event)`, you write `(event) =>`

Arrow functions example (1)

```
function operate(a, b, ops) {
  return ops(a, b);
}

const result = operate(1, 2, (a, b) => {
  return a + b;
});

console.log(result); //3
```

Concise form of arrow functions

If the function only has one line in the curly brackets, you omit the curly brackets and `return`:

```
function operate(a, b, ops) {
  return ops(a, b);
}

const result = operate(1, 2, (a, b) => a + b);

console.log(result);
```

Be careful

```
// Correct
const result = operate(1, 2, (a, b) => a + b);

// Incorrect
const resultWrong = operate(1, 2, (a, b) => {
  a + b;
});

console.log(result); // 3
console.log(resultWrong); //Undefined
```

Scope

- When you create a function, the variables and other things defined inside the function are inside their *own separate scope*.
- This means that they are locked away in their own separate compartments, unreachable from code outside the functions.
- The top level outside all your functions is called the **global scope**.
  - Values defined in the global scope are accessible from everywhere in the code.

Scope example

- Local variable is not accessible to the outside.

```
function myFunction() {
  let carName = "Volvo";
}

console.log(carName); // Error
```

Scope example

- Global variable in accessible everywhere.

```
let carName = "Volvo";

function myFunction() {
  console.log(carName);
}

myFunction(); // 'Volvo'
```

Scope example

- You can declarethe same variable name in a separate scope.

```
let carName = "Volvo";

function myFunction() {
  let carName = "Honda";
  console.log(carName); // Honda
}

console.log(carName); // Volvo
```

Conditions

- `if` statement

```
let choice = "A";

if (choice === "A") {
  console.log("You chose A.");
}
```

- One-line

```
if (choice === "A") console.log("You chose A.");
```

`if` - `else`

```
let choice = "A"; // 'A', 'B', 'C'

if (choice === "A") {
  console.log("You chose A.");
} else {
  console.log("You did not choose A");
}
```

`if` - `else if` - `else`

```
let choice = "A"; // 'A', 'B', 'C'

if (choice === "A") {
  console.log("You chose A.");
} else if (choice === "B") {
  console.log("You chose B.");
} else if (choice === "C") {
  console.log("You chose C.");
} else {
  console.log("You did not choose A, B, or C.");
}
```

Truthy values

- A truthy value is a value that is considered `true` when encountered in a Boolean context.

```
if (true)
if ({})
if ([])
if (42)
if ("0")
if ("false")
if (new Date())
if (-42)
if (12n)
if (3.14)
if (-3.14)
if (Infinity)
if (-Infinity)
```

Falsy value

- A falsy value is considered `false` in a Boolean context.

```
if (false)
if (null)
if (undefined)
if (0)
if (-0)
if (0n)
if (NaN)
if ("")
```

Check for `null` or `undefined`

```
function absolute(number) {
  if (!number) return -1; // Error code
  return Math.abs(number);
}

console.log(absolute(-2)); // 2
console.log(absolute(null)); // -1
console.log(absolute(undefined)); // -1
```

`switch` - `case`

- Can be used instead of multiple `else if`

```
let choice = "A";
let score;
switch (choice) {
  case "A":
    score = 10;
    break;
  case "B":
    score = 5;
    break;
  case "C":
    score = 1;
    break;
  default:
    score = 0;
    break;
}
```

`break`

- *What happens if I forgot a break?*
- If you forget a break then the script will run from the case where the criterion is met and will run the cases after that regardless if a criterion was met.

You can use `object`.

```
let choice = "A";
let mapping = {
  A: 10,
  B: 5,
  C: 1,
};

result = mapping[choice];
console.log(result);
```

- However, this does not handle the *default* case.

Conditional (ternary) operator

- Executing expressions

```
let loading = true;
loading ? console.log("Loading...") : console.log("Done!");
```

- Return values

```
var age = 26;
var beverage = age >= 21 ? "Beer" : "Juice";
console.log(beverage); // "Beer"
```

Back to the *choice* example

- This handles all cases - cool!.

```
let choice = "A";
let mapping = {
  A: 10,
  B: 5,
  C: 1,
};

result = mapping[choice] ? mapping[choice] : 0;
console.log(result);
```

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);
```