

# For loops/Array

Austin

while

# while

```
1  //while (condition)
2  // statement
3
4  let n = 0;
5  while (n < 3) {
6      n++;
7      console.log(n);
8  }
```

```
zhangjunshideMacBook-Air
1
2
3
```

break/continue

# break / continue

```
1  //while (condition)
2  //  statement
3
4  let x = 0;
5  let z = 0;
6  labelCancelLoops: while (true) {
7      x += 1;
8      z = 0;
9      while (true) {
10         console.log("x:", x, "z:", z);
11         z += 1;
12         if (z === 3 && x === 3) {
13             break labelCancelLoops;
14         } else if (z === 3) {
15             break;
16         }
17     }
18 }
```

zhangjunshideMacBook-Air

x: 1 z: 0

x: 1 z: 1

x: 1 z: 2

x: 2 z: 0

x: 2 z: 1

x: 2 z: 2

x: 3 z: 0

x: 3 z: 1

x: 3 z: 2

# For loops

## For loops (basic)

```
1  //for (expression 1; expression 2; expression 3) {
2  |    // code block to be executed
3  |}
4  let text = "";
5  cars = ["BMW", "Volvo", "Saab", "Ford"];
6  for (let i = 0; i < cars.length; i++) {
7  |    text += cars[i] + "<br>";
8  |}
9  console.log(text);
```

zhangjunshideMacBook-Air:js\_folder abao\$ node example.js  
BMW<br>Volvo<br>Saab<br>Ford<br>

## for...in statement

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

```
const arr = [3, 5, 7];  
arr.foo = "hello";
```

```
for (const i in arr) {  
    console.log(i);  
}
```

```
// "0" "1" "2" "foo"
```

```
zhangjunshideMacBook-Air:  
0  
1  
2  
foo
```



## for...of statement

```
1  //for (variable of iterable) zhangjunshideMacBook-Air
2  //    statement              3
3                                5
4  const arr = [3, 5, 7];       7
5  arr.foo = "hello";
6
7  for (const i of arr) {
8    | console.log(i);
9  }
10 // Logs: 3 5 7
```

## for...of statement - Contd.

```
1  //for (variable of iterable)
2  //    statement
3
4  const arr = [3, 5, 7];
5  arr.foo = "hello";
6
7  for (const [key, val] of arr.entries()) {
8  |   console.log(key, val);
9  } ✨
```

```
zhangjunshide
0 3
1 5
2 7
```

# Array

# Array

1. Arrays are a special kind of objects, with non-negative numbered indexes.

## **WARNING !!**

If you use named indexes, JavaScript will redefine the array to an object.

After that, some array methods and properties will produce **incorrect results**.

# Common function of Array

```
1  const cars = [];  
2  cars[0]= "Saab";  
3  cars[1]= "Volvo";  
4  cars[2]= "BMW";  
5  //cars."3" = "Mercedes"; // SyntaxError: Unexpected string  
6  
7  console.log(cars);  
8  console.log("Length of cars array: " + cars.length);  
9  
10 cars.sort();  
11 console.log(cars);
```

```
zhangjunshideMacBook-Air:js_folder abao$ node example.js  
[ 'Saab', 'Volvo', 'BMW' ]  
Length of cars array: 3  
[ 'BMW', 'Saab', 'Volvo' ]
```

# Common function of Array - Contd.

[https://www.w3schools.com/js/js\\_array\\_methods.asp](https://www.w3schools.com/js/js_array_methods.asp)

## Basic Array Methods

[Array length](#)

[Array toString\(\)](#)

[Array at\(\)](#)

[Array join\(\)](#)

[Array pop\(\)](#)

[Array push\(\)](#)

[Array shift\(\)](#)

[Array unshift\(\)](#)

[Array delete\(\)](#)

[Array concat\(\)](#)

[Array copyWithin\(\)](#)

[Array flat\(\)](#)

[Array splice\(\)](#)

[Array toSpliced\(\)](#)

[Array slice\(\)](#)

## See Also:

[Search Methods](#)

[Sort Methods](#)

[Iteration Methods](#)

# Interview problem

# Interview problem

```
1  var num = 0;
2  for (num = 0; num < 5; num++) {
3      setTimeout(function() {
4          console.log(num);
5      }, 1000);
6  }
```

● zhangjunshideMacBook-Air

5  
5  
5  
5  
5

```
1  for (let num = 0; num < 5; num++) {
2      setTimeout(function() {
3          console.log(num);
4      }, 1000);
5  }
```

0  
1  
2  
3  
4



# Interview problem

[https://www.youtube.com/watch?v=N0Au8yc5lOw&ab\\_channel=PJCHENder%28PJCHENder%29](https://www.youtube.com/watch?v=N0Au8yc5lOw&ab_channel=PJCHENder%28PJCHENder%29)

```
1  for (let num = 0; num < 5; num++) {  
2    |   setTimeout(function() {  
3    |     console.log(num);  
4    |   }, 0);  
5  }
```

● zhangjunshideMacBook-Air  
5  
5  
5  
5  
5

# Reference

1. [https://www.w3schools.com/js/js\\_loop\\_for.asp](https://www.w3schools.com/js/js_loop_for.asp)
2. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops\\_and\\_iteration](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration)
3. [https://www.w3schools.com/js/js\\_arrays.asp](https://www.w3schools.com/js/js_arrays.asp)
4. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

# Control flow

Joanne

# if...else

```
1  let score = 85;
2
3  if (score >= 90) {
4      console.log("Grade: A");
5  } else if (score >= 80) {
6      console.log("Grade: B");
7  } else if (score >= 70) {
8      console.log("Grade: C");
9  } else {
10     console.log("Grade: F");
11 }
```

Grade: B

# if...else

```
1  let user = { age: 22, hasMembership: true };
2
3  if (user.age >= 18) {
4      if (user.hasMembership) {
5          console.log("Welcome, member!");
6      } else {
7          console.log("Welcome, guest!");
8      }
9  } else {
10     console.log("Sorry, you're too young to join.");
11 }
```

Welcome, member!

# Conditional Evaluation

1. true: non-zero, non-empty, non-null, non-undefined
2. false: 0, "", null, undefined, NaN

```
let x;  
if (x) {  
    console.log("This won't run because undefined is falsy.");  
} else {  
    console.log("This will run because undefined is falsy.");  
}
```

This will run because undefined is falsy.

# Conditional Evaluation

1. true: non-zero, non-empty, non-null, non-undefined
2. false: 0, "", null, undefined, NaN, false

```
1  if ([]) {  
2  |    console.log("This will run because arrays are truthy.");  
3  |  }  
4  
5  if ({}) {  
6  |    console.log("This will run because objects are truthy.");  
7  |  }
```

# Conditional Evaluation

1. true: non-zero, non-empty, non-null, non-undefined
2. false: 0, "", null, undefined, NaN, false

```
1  if ([]) {  
2  |    console.log("This will run because arrays are truthy.");  
3  |  }  
4  
5  if ({}) {  
6  |    console.log("This will run because objects are truthy.");  
7  |  }
```

```
This will run because arrays are truthy.  
This will run because objects are truthy.
```



# Conditional operators

1. Ternary Operator (?:)
2. Logical operators( || &&)
3. Nullish Coalescing Operator (??)

# Ternary operators

condition ? expressionIfTrue : expressionIfFalse;

```
1  let age = 18;  
2  let status = age >= 18 ? "Adult" : "Minor";
```

# Ternary operators

```
1  const score = 85;  
2  const grade = score >= 90 ? "A" :  
3  |   |   |   | score >= 80 ? "B" :  
4  |   |   |   | score >= 70 ? "C" : "D";  
5  console.log(grade);
```

# Logical operators

```
1  let name = "John";  
2  let displayName = name || "Guest";  
3  console.log(displayName);  
4  
5  let user = { name: "Alice", isMember: true };  
6  let memberName = user.isMember && user.name;  
7  console.log(memberName); |
```

John  
Alice

# Nullish Coalescing Operator (??)

```
value1 ?? value2;
```

# Nullish Coalescing Operator (??)

value1 ?? value2;

```
1  let username = null;  
2  let defaultUsername = username ?? "Anonymous";  
3  console.log(defaultUsername);
```

Anonymous

# throw statement

The primary purpose of the throw statement is to actively raise an error.

**try:** Used to wrap code that may throw an error.

**catch:** Used to handle errors thrown in the try block.

**finally:** The code in this block will execute regardless of whether an error occurred, typically used for cleanup operations.

# example

```
1  function divide(a, b) {  
2      if (b === 0) {  
3          throw new Error("Division by zero is not allowed.");  
4      }  
5      return a / b;  
6  }  
7  
8  try {  
9      console.log(divide(10, 2));  
10     console.log(divide(10, 0));  
11 } catch (error) {  
12     console.error("Caught an error:", error.message);  
13 } finally {  
14     console.log("Execution completed.");  
15 }
```

```
5  
Caught an error: Division by zero is not allowed.  
Execution completed.
```



# error objects

- `SyntaxError`
- `TypeError`
- `ReferenceError`
- `RangeError`
- `EvalError`

```
1  const error = new Error("This is a generic error message.");  
2  console.log(error.message);
```

This is a generic error message.

# SyntaxError

```
1  try {  
2    |   eval("var a = ;");  
3  } catch (e) {  
4    |   console.error(e instanceof SyntaxError);  
5  }
```

true

# TypeError

```
1  try {  
2    let obj = { name: "Alice" };  
3    console.log(obj.age.value);  
4  } catch (e) {  
5    console.error(e instanceof TypeError);  
6    console.error("Caught an error:", e.message);  
7  }
```

true

Caught an error: Cannot read properties of undefined (reading 'value')

```
1  let obj = null;  
2  console.log(obj.name);  
3  
4  let notAFunction = 42;  
5  notAFunction();
```

# ReferenceError

```
1  try {  
2    console.log(nonExistentVariable);  
3  } catch (e) {  
4    console.error(e instanceof ReferenceError);  
5  }
```

true

```
1  function test() {  
2    console.log(innerVar);  
3  }  
4  test();
```

# CustomError

```
1  class CustomError extends Error {  
2    constructor(message) {  
3      super(message);  
4      this.name = "CustomError";  
5    }  
6  }  
7  
8  try {  
9    throw new CustomError("This is a custom error message.");  
10 } catch (error) {  
11   console.error(error.name);  
12   console.error(error.message);  
13 }
```

```
CustomError  
This is a custom error message.
```

# review

```
1  function getDiscount(user) {
2      if (user.isPremium) {
3          return "20% discount";
4      } else if (user.hasMembership) {
5          return "10% discount";
6      } else {
7          return "No discount";
8      }
9  }
10
11  // ternary operator
12  function getDiscountUsingTernary(user) {
13      return user.isPremium
14          ? "20% discount"
15          : user.hasMembership
16          ? "10% discount"
17          : "No discount";
18  }
19
20  let user1 = { isPremium: true, hasMembership: false };
21  let user2 = { isPremium: false, hasMembership: true };
22  let user3 = { isPremium: false, hasMembership: false };
23  ??
```

# review

```
1  function getDiscount(user) {
2      if (user.isPremium) {
3          return "20% discount";
4      } else if (user.hasMembership) {
5          return "10% discount";
6      } else {
7          return "No discount";
8      }
9  }
10
11  // ternary operator
12  function getDiscountUsingTernary(user) {
13      return user.isPremium
14          ? "20% discount"
15          : user.hasMembership
16          ? "10% discount"
17          : "No discount";
18  }
19
20  let user1 = { isPremium: true, hasMembership: false };
21  let user2 = { isPremium: false, hasMembership: true };
22  let user3 = { isPremium: false, hasMembership: false };
23
```

20% discount  
10% discount  
No discount

# review

```
1  class ValidationError extends Error {
2      constructor(message) {
3          super(message);
4          this.name = "ValidationError";
5      }
6  }
7
8  function validateAge(age) {
9      if (age < 18) {
10         throw new ValidationError("Age must be at least 18");
11     }
12     return "Age is valid";
13 }
14
15 try {
16     console.log(validateAge(15));
17 } catch (e) {
18     console.error(e instanceof ValidationError);
19     console.error("Caught error:", e.message);
20 } finally {
21     console.log("This code will run even if an error occurs");
22 }
```



# review

```
1  class ValidationError extends Error {
2      constructor(message) {
3          super(message);
4          this.name = "ValidationError";
5      }
6  }
7
8  function validateAge(age) {
9      if (age < 18) {
10         throw new ValidationError("Age must be at least 18");
11     }
12     return "Age is valid";
13 }
14
15 try {
16     console.log(validateAge(15));
17 } catch (e) {
18     console.error(e instanceof ValidationError);
19     console.error("Caught error:", e.message);
20 } finally {
21     console.log("This code will run even if an error occurs");
22 }
```

true

Caught error: Age must be at least 18

This code will run even if an error occurs

# Reference

1. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator)
2. [https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Guide/Control\\_flow\\_and\\_error\\_handling](https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Guide/Control_flow_and_error_handling)
3. <https://medium.com/@ks.deepak07/error-object-in-js-1e433541bb4f>

# Keyed Collections

Kai

# Last week's question...

## “How to distinguish between normal objects and function objects?”

- Declared with “**function ()**” or “**()=>**” (arrow function)

while regular object is declared by only “**= { ... }**”

```
1 // Function expression
2 var test = function() {
3     console.log("I'm a function");
4 };
5
6 // Arrow function
7 var anotherTest = () => {
8     console.log("I'm also a function");
9 };
```

# Last week's question...

**“How to distinguish between normal objects and function objects?”**

- Use `typeof` or `instanceof`

```
1  const myFunc = function() {};  
2  const myObj = {};  
3  const anotherObj = { key: "value" };  
4  
5  console.log(typeof myFunc);           // "function"  
6  console.log(typeof myObj);            // "object"  
7  console.log(typeof anotherObj);       // "object"  
8  
9  console.log(myFunc instanceof Function); // true  
10 console.log(myObj instanceof Function);  // false  
11 console.log(anotherObj instanceof Function); // false
```

## What are normal objects and what are function objects?

```
1  var x = 1;
2  var obj = {
3      x: 20,
4      fn: function(){
5          console.log(this.x)
6          var test = function() {
7              console.log(this.x)
8          }
9          test()
10     }
11 }
12 obj.fn()
```

Is obj a function object or a normal object?

```
1 var obj = {  
2   fn: function() {  
3     console.log("This is a method in obj");  
4   }  
5 };
```

## Is obj a function object or a normal object?

```
1 var obj = {  
2   fn: function() {  
3     console.log("This is a method in obj");  
4   }  
5 };  
6  
7 obj.fn(); // This works because `fn` is a function property of  
           // `obj`  
8 obj();    // Error, because `obj` is not a function object.
```



# For this week

- **Map**
  - Introduction
  - What are the differences between Map and Object?
  - Map v.s. Weak Map
- **Set**
  - Introduction
  - Set v.s. Weak Set

# What is a map?

**Def:** A simple key-value pairs where

1. Keys & Values can be any data type
2. Every key is unique

# Basic Operation – Initialization & Insertion

## 1. Set data upon declaration

```
1 const map = new Map([
2   [1, { name: 'Jimmy', email: 'jimmy@test.com' }],
3   [2, { name: 'Jed', email: 'jed@test.com' }],
4   [3, { name: 'Frank', email: 'frank@test.com' }]
5 ])
6 console.log(map)
7 console.log(map.size)
```

type name! not the name you declare with



```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}  
3
```

# Basic Operation – Initialization & Insertion

2. Declare with an array of key-value pairs

```
1 - const data = [  
2   [1, { name: 'Jimmy', email: 'jimmy@test.com' }],  
3   [2, { name: 'Jed', email: 'jed@test.com' }],  
4   [3, { name: 'Frank', email: 'frank@test.com' }]  
5 ]  
6 const map = new Map(data)
```

# Basic Operation – Initialization & Insertion

## 3. Insert/Update data

```
1  const map = new Map()
2  map.set(1, { name: 'Jimmy', email: 'jimmy@test.com' })
3  map.set(2, { name: 'Jed', email: 'jed@test.com' })
4  map.set(3, { name: 'Frank', email: 'frank@test.com' })
5  console.log(map)
6  map.set(1, {name: 'Justin', email: 'itsjustin@test.com'})
7  map.set(4, {name: 'Kai', email: 'kaihsuant@test.com'})
8  console.log(map)
```

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}  
Map(4) {  
  1 => { name: 'Justin', email: 'itsjustin@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' },  
  4 => { name: 'Kai', email: 'kaihsuant@test.com' }  
}
```

# Basic Operation – get value from key

```
1 const map = new Map()
2 map.set("name", "Alice");
3 map.set(1, "one");
4 map.set(true, "boolean value");
5 console.log(map)
6
7 console.log(map.has(1))
8 console.log(map.has(4))
9
10 console.log(map.get(true))
```

```
node /tmp/APLwQiXCoU.js
```

```
Map(3) { 'name' => 'Alice', 1 => 'one', true => 'boolean value' }
```

```
true
```

```
false
```

```
boolean value
```



# Basic Operation – deletion & clear

```
1  const map = new Map()
2  map.set("name", "Alice").set(1, "one").set(true, "boolean value");
3  console.log(map)
4
5  map.delete("name")
6  console.log(map)
7  map.clear()
8  console.log(map)
```

```
node /tmp/GHzRCkuUlK.js
Map(3) { 'name' => 'Alice', 1 => 'one', true => 'boolean value' }
Map(2) { 1 => 'one', true => 'boolean value' }
Map(0) {}
|
```

# Basic Operation – Iteration

## 1. for...of

```
for (const [key, value] of map) {  
  console.log(`${key} = ${value}`);  
}
```

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}
```

# Basic Operation – Iteration

## 1. for...of

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}
```

```
for (const [key, value] of map) {  
  console.log(`${key} = ${value}`);  
}
```

Output:

```
1 = [object Object]  
2 = [object Object]  
3 = [object Object]
```

Why?

# Basic Operation – Iteration

## 1. for...of

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}
```

```
for (const [key, value] of map) {  
  console.log(`${key} = ${JSON.stringify(value)}`);  
}
```

Output:

```
1 = {"name":"Jimmy","email":"jimmy@test.com"}  
2 = {"name":"Jed","email":"jed@test.com"}  
3 = {"name":"Frank","email":"frank@test.com"}
```

# Basic Operation – Iteration

## 2. for each

```
map.forEach((key, value) => {  
  console.log(`${key} = ${JSON.stringify(value)}`);  
});
```

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}
```

# Basic Operation – Iteration

## 2. for each

```
map.forEach((key, value) => {  
  console.log(`${key} = ${JSON.stringify(value)}`);  
});
```

Output:

```
[object Object] = 1  
[object Object] = 2  
[object Object] = 3
```

Why?

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}
```

# Basic Operation – Iteration

## 2. for each

```
map.forEach((value, key) => {  
    console.log(`${key} = ${JSON.stringify(value)}`);  
});
```

Output:

```
1 = {"name":"Jimmy","email":"jimmy@test.com"}  
2 = {"name":"Jed","email":"jed@test.com"}  
3 = {"name":"Frank","email":"frank@test.com"}
```

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}
```

# Basic Operation – Iteration

## 3. Iteration of key only / value only

```
Map(3) {  
  1 => { name: 'Jimmy', email: 'jimmy@test.com' },  
  2 => { name: 'Jed', email: 'jed@test.com' },  
  3 => { name: 'Frank', email: 'frank@test.com' }  
}
```

Output:

```
9 ▾ for (const key of map.keys()) {  
10   console.log(key);  
11 }  
12  
13 ▾ for (const value of map.values()) {  
14   console.log(JSON.stringify(value));  
15 }
```

```
1  
2  
3  
{"name":"Jimmy","email":"jimmy@test.com"}  
{"name":"Jed","email":"jed@test.com"}  
{"name":"Frank","email":"frank@test.com"}
```



# Map v.s. Object

Feature	Map	Object
Key Types	Can be any data type (e.g., string, object)	Typically strings or symbols
Iteration	Iterates in insertion order using for...of, forEach, .keys(), etc.	Must use for...in or Object.keys()
Size Property	size property directly gives entry count	No size property; use Object.keys().length
<b>Performance</b>	<b>Optimized for frequent additions and deletions</b>	<b>Performance can degrade with frequent changes</b>

# Map v.s. Weak Map

## 1. Key Types

WeakMap only accepts objects as keys.

```
1  const weak = new WeakMap();
2  weak.set({}, "some value"); // Works
3  weak.set(123, "ID number"); // Error: Invalid value used as weak
                                map key
```

# Map v.s. Weak Map

## 2. Weak Reference

- A reference to an object that allows automatic garbage collection when there are no other references to the object
- Garbage Collection: JavaScript can **automatically** free up memory by removing objects that only have weak references.

Data Structure	Reference Type	Description
Variables, Arrays, etc	Strong	Keeps the object in memory as long as referenced
Map, Set	Strong	Retains all entries, preventing GC
WeakMap, WeakSet	Weak	GC if unreferenced

## Example 1

```
1  const weakMap = new WeakMap();
2  let user = { name: "Alice" };
3  weakMap.set(user, "some associated data");
4  console.log(weakMap.get(user));
5
6  user = null;
7  console.log(weakMap.get(user));
```

```
node /tmp/e8tCjs7qs0.js
some associated data
undefined
```

## Example 2

```
1  const weakMap = new WeakMap();
2
3  (function() {
4      let user = { name: "Alice" };
5      weakMap.set(user, "some associated data");
6
7  })();
8  console.log(weakMap);
```

## Example 2

```
1  const weakMap = new WeakMap();
2
3  (function() {
4      let user = { name: "Alice" };
5      weakMap.set(user, "some associated data");
6      // At the end of this block, `user` goes out of scope
7  })();
8  console.log(weakMap);
```

### Output

```
node /tmp/6xEIb1Hxn1.js
WeakMap { <items unknown> }
```

## Example 2 – Comparison

```
1  const map = new Map();  
2  
3  (function() {  
4      let user = { name: "Alice" };  
5      map.set(user, "some associated data");  
6  })();  
7  console.log(map);
```

### Output

```
node /tmp/zLOVn1BAqF.js
```

```
Map(1) { { name: 'Alice' } => 'some associated data' }
```

# Map v.s. Weak Map

## 3. Iteration

- Weak map is not iterable, there's no method to get all keys/values

## 4. Use Cases

- When data is only needed temporarily
- Where it's a big program and there's high potential of memory leak



# Set

- a collection of unique values
- maintains the insertion order of elements
- stores any type of value, such as numbers, strings, objects, or even other Sets.

# Basic Operations

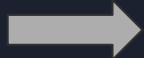
```
1 let set = new Set();
2 set.add(1);
3 set.add("two");
4 set.add(1);
5 console.log(set);
6
7 set.delete(1);
8 console.log(set);
9
10 console.log(set.has("two"));
11 console.log(set.has(3));
12
13 set.clear();
14 console.log(set.size);
```

## Output

```
node /tmp/QtsqJtp7Jk.js
Set(2) { 1, 'two' }
Set(1) { 'two' }
true
false
0
```

# Unique Object?

```
1 const kaiObj = {  
2   name: 'Kai',  
3   age: '20'  
4 };  
5 const set = new Set([kaiObj]);  
6 set.add(kaiObj)  
7 console.log(set)  
8  
9 set.add({  
10   name: 'Kai',  
11   age: '20'  
12 });  
13 console.log(set)
```



```
Set(1) { { name: 'Kai', age: '20' } }
```

```
1 const kaiObj = {  
2   name: 'Kai',  
3   age: '20'  
4 };  
5 const set = new Set([kaiObj]);  
6 set.add(kaiObj)  
7 console.log(set)  
8  
9 set.add({  
10   name: 'Kai',  
11   age: '20'  
12 });  
13 console.log(set)
```

```
node /tmp/cTutEpAeiH.js
```

```
Set(1) { { name: 'Kai', age: '20' } }
```

```
Set(2) { { name: 'Kai', age: '20' }, { name: 'Kai', age: '20' } }
```

# Iteration

```
1  const set = new Set([1, 2, 3]);
2  for (let value of set) {
3      console.log(value);
4  }
5
6  set.forEach(value => console.log(value));
```

## Output

node /tmp/9GKqi

1

2

3

1

2

3

# Use Case

When wanting to remove duplicates

```
1  const numbers = [1, 2, 2, 3, 4, 4];  
2  const uniqueNumbers = [...new Set(numbers)];  
3  console.log(uniqueNumbers);
```

```
node /tmp/fj7XIBY3Et.js  
[ 1, 2, 3, 4 ]
```

- The ... (spread operator) is used to spread each entry from the Set into an array.

# WeakSet

- Only objects as entries
- Weak reference
- No iteration, no `.size`, no `.clear`

## WeakSet – Example

```
1  const weakSet = new WeakSet();      node /tmp
2  let user = { name: "Alice" };      true
3  weakSet.add(user);                  false
4
5  console.log(weakSet.has(user));
6
7  user = null;
8  console.log(weakSet.has(user));
```



# Reference

[1]Map:<https://jimmyswebnote.com/%E3%80%90-javascript-%E3%80%91map-%E7%A8%A8%E8%A8%98/>

[2] Weakmap and Weakset application:<https://zh.javascript.info/weakmap-weakset>

[3]延伸-- Garbage Collection:<https://zh.javascript.info/garbage-collection>

[4]Set:<https://pink-learn-frontend.medium.com/javascript%E8%90%8C%E6%96%B0%E7%AD%86%E8%A8%98-%EF%BD%8Dap%E7%9A%84%E5%85%84%E5%BC%9Fset-%E8%AA%8D%E8%AD%98set%E9%80%99%E5%80%8B%E7%89%A9%E4%BB%B6-229340b574ea>

# Primitive vs Object

Max

# Primitive vs Object

Primitive: simplest forms of data

- string
- number
- bigint
- boolean
- undefined
- symbol
- null

Object: a value composed of primitives

```
> typeof [1, 2, 3]
< 'object'
> typeof {}
< 'object'
> typeof new Date()
< 'object'
```

# Assigning primitive to variable

- copying **value** to the new variable

```
> let a = 123
   let b = a

console.log(a, b)
a = 456
console.log(a, b)
123 123
456 123
```

# Assigning object to variable

- copying **reference** to the new variable

```
> let a = [123]  
    let b = a
```

```
console.log(a, b)  
a.push(456)  
console.log(a, b)
```

---

```
▶ [123] ▶ [123]
```

---

```
▶ (2) [123, 456] ▶ (2) [123, 456]
```

# Object prototypes

```
> const a = {}
```

```
console.log(a.toString())
```

---

```
[object Object]
```

## Where is `toString()` defined?

- Every object in JavaScript has a built-in property called its prototype.
- When we try to access a property that can't be found in the object itself, the prototype is searched for the property.



When we call `a.toString()`:

1. looks for `toString` in `a`
2. can't find it there, so looks in the prototype object of `a` for `toString`
3. finds it there, and calls it.

# Access prototype

- Through `__proto__`

```
> console.log(a.__proto__)
```

```
VM1116:1  
{__defineGetter__: f, __defineSetter__: f, has  
▼ OwnProperty: f, __lookupGetter__: f, __lookupS  
etter__: f, ...} i  
  ▶ constructor: f Object()  
  ▶ hasOwnProperty: f hasOwnProperty()  
  ▶ isPrototypeOf: f isPrototypeOf()  
  ▶ propertyIsEnumerable: f propertyIsEnumerable  
  ▶ toLocaleString: f toLocaleString()  
  ▶ toString: f toString()  
  ▶ valueOf: f valueOf()  
  ▶ __defineGetter__: f __defineGetter__()  
  ▶ __defineSetter__: f __defineSetter__()  
  ▶ __lookupGetter__: f __lookupGetter__()  
  ▶ __lookupSetter__: f __lookupSetter__()  
  __proto__: null  
  ▶ get __proto__: f __proto__()  
  ▶ set __proto__: f __proto__()
```

# Prototype chain



# Most basic prototype

- `Object.prototype`

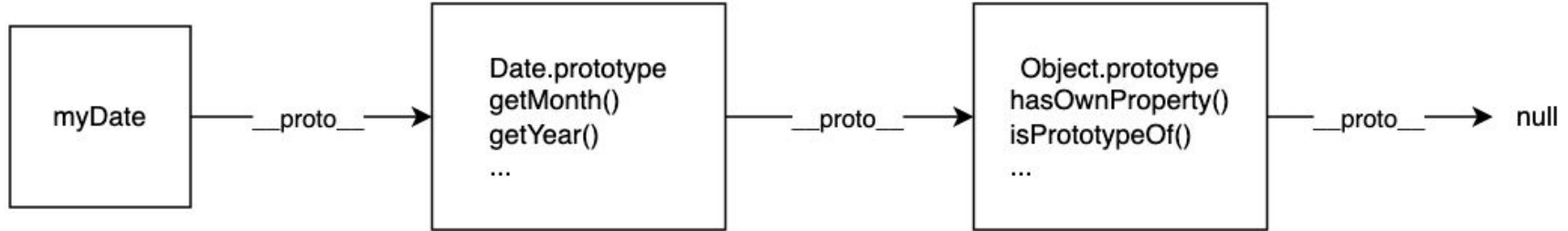
```
> console.log(Object.prototype)
```

[VM872:1](#)

```
{__defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, __lookupSetter__: f, ...} ⓘ  
  ▶ constructor: f Object()  
  ▶ hasOwnProperty: f hasOwnProperty()  
  ▶ isPrototypeOf: f isPrototypeOf()  
  ▶ propertyIsEnumerable: f propertyIsEnumerable()  
  ▶ toLocaleString: f toLocaleString()  
  ▶ toString: f toString()  
  ▶ valueOf: f valueOf()  
  ▶ __defineGetter__: f __defineGetter__()  
  ▶ __defineSetter__: f __defineSetter__()  
  ▶ __lookupGetter__: f __lookupGetter__()  
  ▶ __lookupSetter__: f __lookupSetter__()  
  __proto__: null  
  ▶ get __proto__: f __proto__()  
  ▶ set __proto__: f __proto__()
```

# Other prototype chain example

```
> const myDate = new Date()
```



# Shadowing properties

```
> const myDate = new Date();  
  
console.log(myDate.getTime());  
  
myDate.getTime = function () {  
  console.log("shadowing the property");  
};
```

```
myDate.getTime();
```

---

```
1730020533173
```

```
VM1946:3
```

---

```
shadowing the property
```

```
VM1946:6
```

# Summary

- Every object in JavaScript has a built-in property called its prototype.
- Prototype is stored in `__proto__`
- The most basic prototype is `Object.prototype`
- When accessing a property of an object, the prototype chain is searched until the end ( `null` ) is reached

# Javascript Closure

SGarry



# JavaScript Clousre

- JavaScript Clousre is the combination of a function and the Lexical Environment within which that function was declared [1].
- In short, Closure is a function which can store and access the available variables in current environment.

# Lexical Environment

- Lexical Environment consists of
  - Environment Record
    - a specification type used to define the association of Identifiers to specific variables and functions [2]
  - Outer reference of Environment Record
    - Environment Record from parent or global environment
- A function will try to find variables or function in own Environment Record, if it can't find, it will try to find them from outer.

# Rule of Accessing Variables

- In JavaScript, inner scope can access the variables from outer scope, but outer scope can't access inner scope.

```
1  ✓ function init(){
2      let name = "John"
3  ✓  function displayName(){
4      |    console.log(name)
5      |
6      |    displayName()
7      |
8      }
9  }
10 init()
```

▲ Inner function can access outer variable

```
1  ✓ function init(){
2      console.log(name)
3  ✓  function displayName(){
4      |    let name = "John"
5      |
6      |    displayName()
7      |
8      }
9  }
10 init()
```

▲ outer function can't access inner variable

# var, let in Lexical Environment

- var will be hoisted at the top of function. Therefore, the lexical env. of var will be outer one.
- In the other hands, lexical env. of let is in the block, which is the inner one.

```
1  var num = 0;
2  for (num = 0; num < 5; num++) {
3      |   setTimeout(function() {
4          |   console.log(num);
5          |   }, 1000);
6      |   }
```

```
1  for (let num = 0; num < 5; num++) {
2      |   setTimeout(function() {
3          |   console.log(num);
4          |   }, 1000);
5      |   }
```

# Retention of Environments

- Closure have a feature that it can keep the lexical environment.
- As long as the reference of inner environment exists, the outer one won't disappear.

```
1  function printName() {  
2      let name = "John"  
3      function displayName() {  
4          console.log(name);  
5      }  
6      return displayName;  
7  }  
8  
9  let func = printName();  
10 func();
```

▲ Retention of Environments of printName() [3]

# Independent Environments

- Each function will keep their own environment, which means each of them will keep their reference.

```
1  function makeAdder(x) {  
2      return function (y) {  
3          return x + y;  
4      };  
5  }  
6  
7  const add5 = makeAdder(5);  
8  const add10 = makeAdder(10);  
9  
10 console.log(add5(2)); // 7  
11 console.log(add10(2)); // 12
```

▲ The example of producing 2 lexical environments [1]

# Practical Example of Closure

- Following is an example of Emulating private methods with closures.
- Counter will keep variable as private, Outer environment can't access.

Private variable using closure [1] ►

```
1  const counter = (function () {
2      let privateCounter = 0;
3      function changeBy(val) {
4          privateCounter += val;
5      }
6
7      return {
8          increment() {
9              changeBy(1);
10             },
11          decrement() {
12              changeBy(-1);
13             },
14          value() {
15              return privateCounter;
16             },
17      };
18  })();
19
20  console.log(counter.value()); // 0.
21
22  counter.increment();
23  counter.increment();
24  console.log(counter.value()); // 2.
25
26  counter.decrement();
27  console.log(counter.value()); // 1.
28
```

# Performance of Closure

- Javascript closure lets function manages own scope and closure, which leads to the worse performance if closures are not needed for a particular function.
- For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. [1]

```
1  function MyObject(name, message) {  
2      this.name = name.toString();  
3      this.message = message.toString();  
4      this.getName = function () {  
5          return this.name;  
6      };  
7  
8      this.getMessage = function () {  
9          return this.message;  
10     };  
11 }
```

▲ function within a object [1]

```
1  function MyObject(name, message) {  
2      this.name = name.toString();  
3      this.message = message.toString();  
4  }  
5  MyObject.prototype = {  
6      getName() {  
7          return this.name;  
8      },  
9      getMessage() {  
10         return this.message;  
11     },  
12 };
```

▲ function prototype within a object [1]



# Shallow copy & deep copy

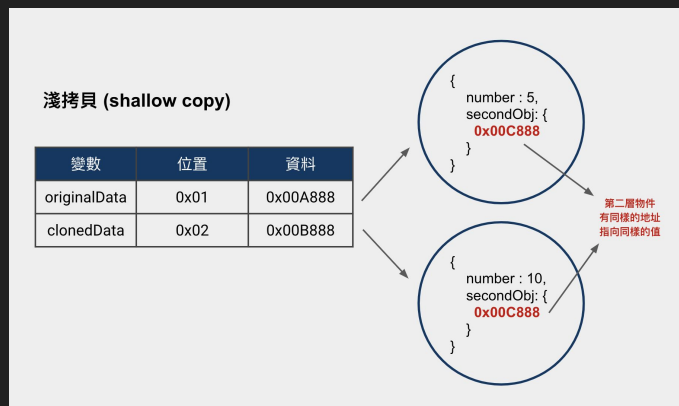
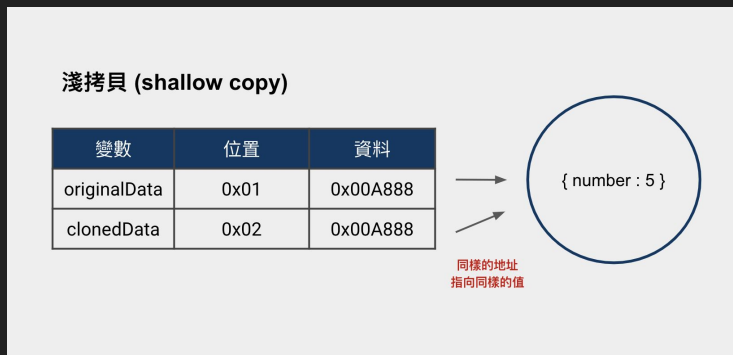
SGarry

# Primitive type and Complex type

- Primitive type is data that is not an object and has no methods or properties[1].
  - for example: string, number, bigint, boolean, undefined, symbol, null.
- In contrast, an object with methods and properties will be complex type (also called Object).
  - for example: object, array, function.

# Shallow copy & Deep copy

- Shallow copy is pass-by-reference.  
The reference of the new data is as the same as the the original data.
- Deep copy (also called real copy) is pass-by-value.  
The reference of the new data is different from the reference of the original data.



▲ Shallow copy and deep copy [2]

# Practical example of shallow copy

- Assume we have an array, and we use assignment operator to copy the data.

```
originalData = [1, 2, 3]
newData = originalData

console.log("Original Data:", originalData) // 1, 2, 3
console.log("Original Data:", newData)      // 1, 2, 3

newData[0] = 0

console.log("Original Data:", originalData) // 0, 2, 3
console.log("Original Data:", newData)      // 0, 2, 3
```

# Understanding Shallow Copy

- When copying data, if any references in the new data still point to the original data, it's called a "shallow copy."

```
const originalData = {  
  firstLayerNum: 10,  
  obj: {  
    secondLayerNum: 100,  
  },  
};  
  
const clonedData = {  
  firstLayerNum: originalData.firstLayerNum,  
  obj: originalData.obj,  
};  
  
clonedData.firstLayerNum = 20;  
clonedData.obj.secondLayerNum = 200;  
  
console.log(originalData.firstLayerNum);           // 10  
console.log(originalData.obj.secondLayerNum);      // 200
```

# Simplistic approach to Deep copy

- `Json.stringify()` and `Json.parse()` are the common way to achieve deep copy.
- However, some data may be change after the process.

```
const originalData = {
  undefined: undefined,      // undefined will be completed lost including the key
  notANumber: NaN,           // NaN will be changed to null
  infinity: Infinity,        // Infinity will be changed to null
  regexp: /.*/,              // regexp will be changed to an empty object {}
  date: new Date('1999-12-31T23:59:59'), // Date will get stringified
};
const faultyClonedData = JSON.parse(JSON.stringify(originalData));

console.log(faultyClonedData.undefined); // undefined
console.log(faultyClonedData.notANumber); // null
console.log(faultyClonedData.infinity); // null
console.log(faultyClonedData.regexp); // {}
console.log(faultyClonedData.date); // "1999-12-31T15:59:59.000Z"
```

# Other approaches to Deep copy

- Create a new object instead of directly copy [3].
  - `const myDogObjAssign = Object.assign({}, myDog, { name: 'puppy', age: 2 });`
- Use spread operator to iterate the data [3].
  - `const myDogSpreadParams = { ...myDog, name: 'puppy', age: 2 };`
- Utilize `structuredClone()` is also a way to achieve the deep copy [3].
  - `const clone = structuredClone(original);`
- Apply third party library to achieve the deep copy.
  - Lodash library provide `cloneDeep()` to easily deep copy the data.
  - Ramda also provide similar functionality.

# Event loop 事件循環



JavaScript has a **single-thread** execution environment and relies on the **event loop** to manage **asynchronous** operations.

**synchronous**

試想一個情境

**Why can JavaScript implement  
asynchronous operations?**

**asynchronous**

**Why can JavaScript implement  
asynchronous operations?**

# Event Loop

**Event loop 不存在JavaScript本身，  
是由JavaScript的執行環境 (browser or node.js)實現**



異步任務又分宏任務跟微任務

macro-task(宏任務): script, setTimeout, setInterval...

micro-task(微任務): Promise, process.nextTick(node.js)...

微任務比宏任務有更優先的執行順序

JS

call stack

webapis

DOM(document)

AJAX

setTimeout

event loop



callback queue(task queue)

OnClick

OnLoad



```
console.log('hi GDSC');  
  
setTimeout(function () {  
  console.log('Event loop');  
}, 1000);  
  
console.log('byebye')
```



```
console.log('hi GDSC');  
  
setTimeout(function () {  
  console.log('Event loop');  
}, 1000);  
  
console.log('byebye')
```

call stack

webapis

console

event loop 

callback queue (task queue)




```
console.log('hi GDSC');  
  
setTimeout(function () {  
  console.log('Event loop');  
}, 1000);  
  
console.log('byebye')
```

console  
hi GDSC

call stack

console.log( 'hi GDSC' )

webapis

event loop 

callback queue (task queue)



```
console.log('hi GDSC');  
  
setTimeout(function () {  
  console.log('Event loop');  
}, 1000);  
  
console.log('byebye')
```

console  
hi GDSC

call stack

setTimeout function

webapis

Timer

setTimeout function

event loop



callback queue (task queue)

setTimeout function





```
console.log('hi GDSC');  
  
setTimeout(function () {  
  console.log('Event loop');  
}, 1000);  
  
console.log('byebye')
```

call stack

console.log('byebye')

webapis

console

hi GDSC

byebye

event loop



callback queue (task queue)

setTimeout function



```
console.log('hi GDSC');  
  
setTimeout(function () {  
  console.log('Event loop');  
}, 1000);  
  
console.log('byebye')
```

call stack

webapis

console

hi GDSC

byebye

Event loop

event loop



callback queue (task queue)

setTimeout function





## Step 1

JavaScript 在 Single Thread 中執行所有任務，  
同步任務即時在 call stack 中處理。

## Step 2

如果遇到異步任務, 例如: `setTimeout`, 執行環境會調用相關的 API ( Web API), 等待此異步任務的結果之後, 再被放置到 `callback queue(task queue)`中

### Step 3

當call stack中的所有同步任務都完成後， JavaScript 會檢查微任務隊列，並優先執行其中的任務。只有當微任務隊列空了，才會從宏任務隊列中提取下一個任務到 call stack 中執行。

## Event loop 步驟

### Step 1

JavaScript 在 Single Thread 中執行所有任務，同步任務即時在 call stack 中處理。

### Step 2

如果遇到異步任務，例如：setTimeout，執行環境會調用相關的 API ( Web API)，等待此異步任務的結果之後，再被放置到 callback queue(task queue) 中

### Step 3

當 call stack 中的所有同步任務都完成後，JavaScript 會檢查微任務隊列，並優先執行其中的任務。只有當微任務隊列空了，才會從宏任務隊列中提取下一個任務到 call stack 中執行。

## Summary

只要call stack空了之後，就會讀取 callback queue，不斷重複這個步驟，直到所有任務完成。