Node

JS OOP

Object Literal

```
let circle = {
radius: 1,
border: 2,
}
```

// Destructuring

const { radius, border } = circle;

// We have two variables now

Object Literal

```
let circle = {
radius: 1,
border: 2,
location: {
  x: 45,
  y: 35
 }
}
```

Object Literal

```javascript
let circle = {
radius: 1,
draw: function () {
console.log('draw');
}
}
circle.draw();
```

Factory Function

FUNCTION

```
function createCircle(radius, border) {
  return {
    radius: radius,
    border: border,
    getArea: function() {
      return Math.PI * this.radius * this.radius;
    }
  };
}
```

USAGE

```
// Usage
let myCircle = createCircle(1, 2);
console.log(myCircle.radius); // 1
console.log(myCircle.border); // 2
console.log(myCircle.getArea()); // 3.141592653589793
```

Constructor Function

```javascript
function Circle(radius) {
  this.radius = radius;
  this.draw = function () {
    console.log("Draw: r=" + radius);
  }
}
const c = new Circle(5); //new Object
c.draw();
```

Don't Miss It

this

Referes to the object calling current function

Constructor property

```
let x = {}
// let x= new Object()
```

//factory functions use default constructor

//check from browser by

object.constructor

Value vs Reference Types

Value Types

Number

String

Boolean

Symbol

undefined

null

Reference Types

Object

Function

Array

Value vs Reference Types

```
let x = 10;
let y = x;
x = 20;
//y will have 10
```

```
let x = {value:10}
let y = x;
x.value = 20;
//y.value will have 20
```

Primitives are copied by their **value**

Objects are copied by their **reference**

What will be the output

```
let x = 10;
function increase(x) {
x++;
}
increase(x);
console.log(x);
//10
```

```
let y = { value: 10 };
function increaseObj(y) {
y.value++;
}
increaseObj(y);
console.log(y.value);
//11
```

Loop Through keys

```
function Circle(radius) {
this.radius = radius;
this.draw = function () {
console.log("Draw: r=" + radius);
}
}
const c = new Circle(5);
for (let key in c) {
console.log(key, c[key]);
}
```

Private Properties And Methods

```javascript
function createCircle(radius, border) {
  // Private variables
  let privateRadius = radius;
    // Public methods
  return {
    getRadius: function() {
      return privateRadius;
    },
  };
}
```

```javascript
// Usage
let myCircle = createCircle(1, 2);

console.log(myCircle.getRadius()); // 1
console.log(myCircle.getBorder()); // 2
console.log(myCircle.getArea());   // 3.141592653589793
```
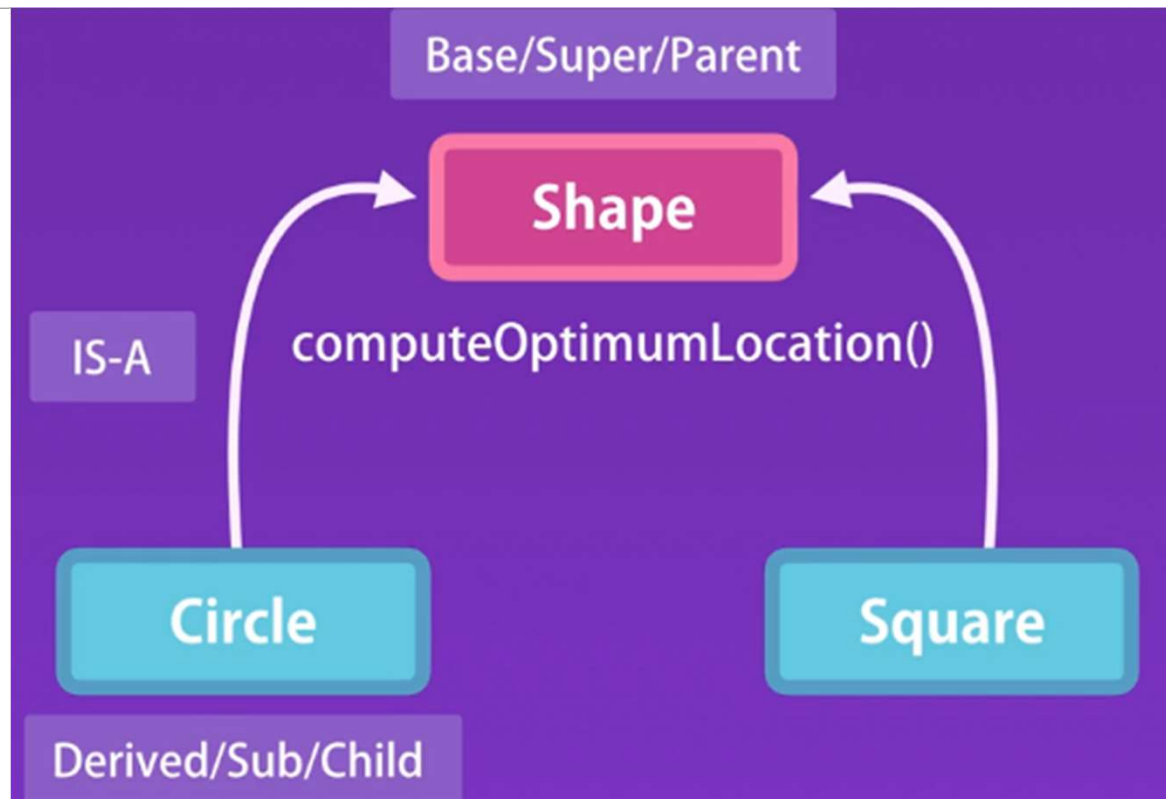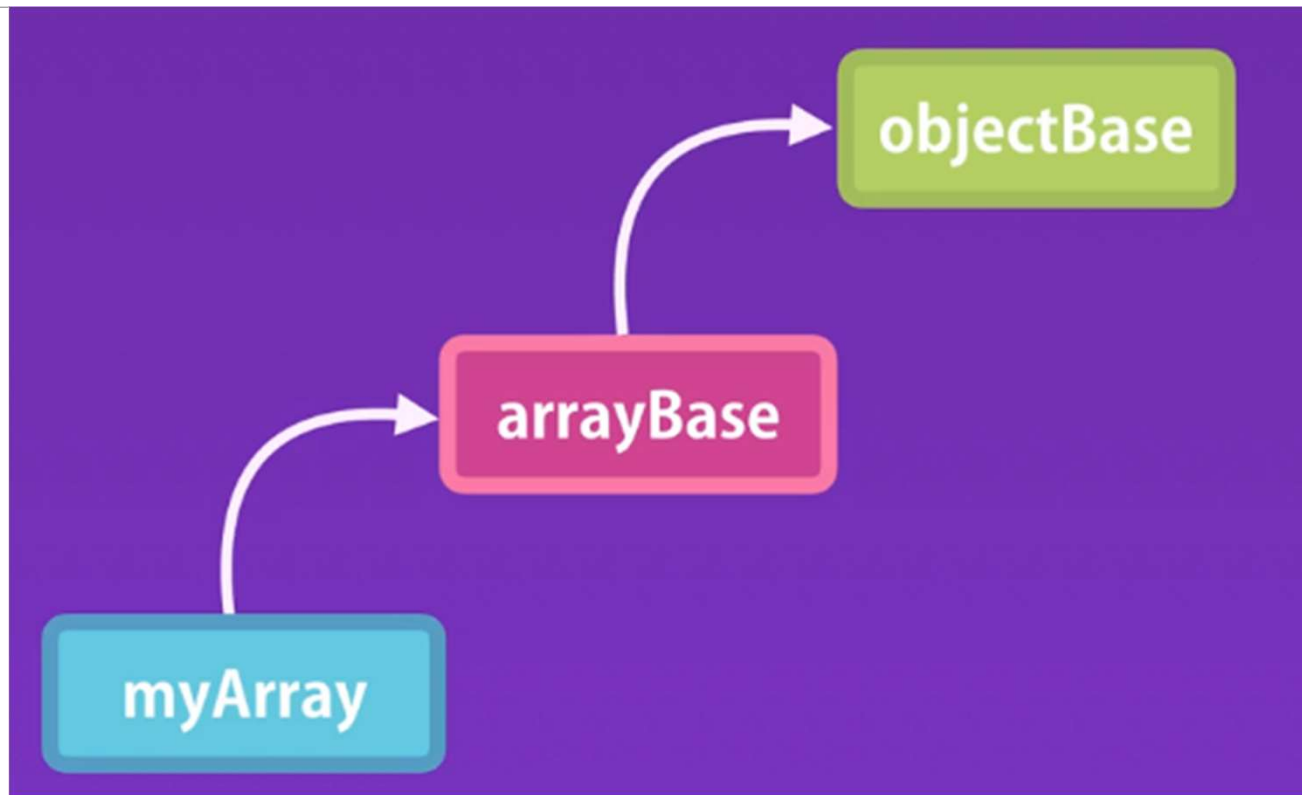
Inheritence

Prototypical Inheritence

```
 //A prototype is an object from which other objects inherit properties
// Every object (except the root object) has a prototype (parent).
// To get the prototype of an object:
Object.getPrototypeOf(obj);

// In Chrome, you can inspect "__proto__" property. But you should
// not use that in the code.
// x.__proto__ === y.__proto__
```

Multi level Inheritence

"prototype" property

```javascript
// Constructors have a "prototype"
property. It returns the object
// that will be used as the prototype for
objects created by the constructor.
Object.prototype ===
Object.getPrototypeOf({})
Array.prototype ===
Object.getPrototypeOf([])
```

Same Constructor Same Prototype

```
// All objects created with the same
constructor will have the same prototype.

// A single instance of this prototype will
be stored in the memory.

const x = {};

const y = {};

Object.getPrototypeOf(x) ===
Object.getPrototypeOf(y); // returns true
```

Best Practice

```
// When dealing with large number of
objects, it's better to put their

// methods on their prototype. This way, a
single instance of the methods

// will be in the memory.

Circle.prototype.draw = function() {}
```

Prototypical Inheritence

```javascript
function Shape() {}

function Circle() {}

// Prototypical inheritance
Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;
```

Call Super

```javascript
function Rectangle(color) {
// To call the super constructor
Shape.call(this, color);
}
```

Method Overriding

```
// Method overriding
Shape.prototype.draw = function() {}
Circle.prototype.draw = function() {
// Call the base implementation
Shape.prototype.draw.call(this);

// Do additional stuff here
}
```

Dos & Donts

```
 // Don't create large inheritance
hierarchies.

 // One level of inheritance is fine.


 // Use mixins to combine multiple objects

 // and implement composition in JavaScript.
```

Resources

https://1drv.ms/f/s!AtGKdbMmNBGdhQmUmPL4RQRrfM1Y

ES6 Classes

Syntactical Sugar to Prototypical Inheritence

Class

```
class Circle {
  constructor(radius) {
    this.radius = radius;
  }
  // These methods will be added to the prototype.
  draw() {
  }
}
```

Static Methods

```
// This will be available on the Circle
class (Circle.parse())
static parse(str) {

}
```

Private Symbol

```
// Using symbols to implement private
properties and methods
const _size = Symbol();
const _draw = Symbol();
```

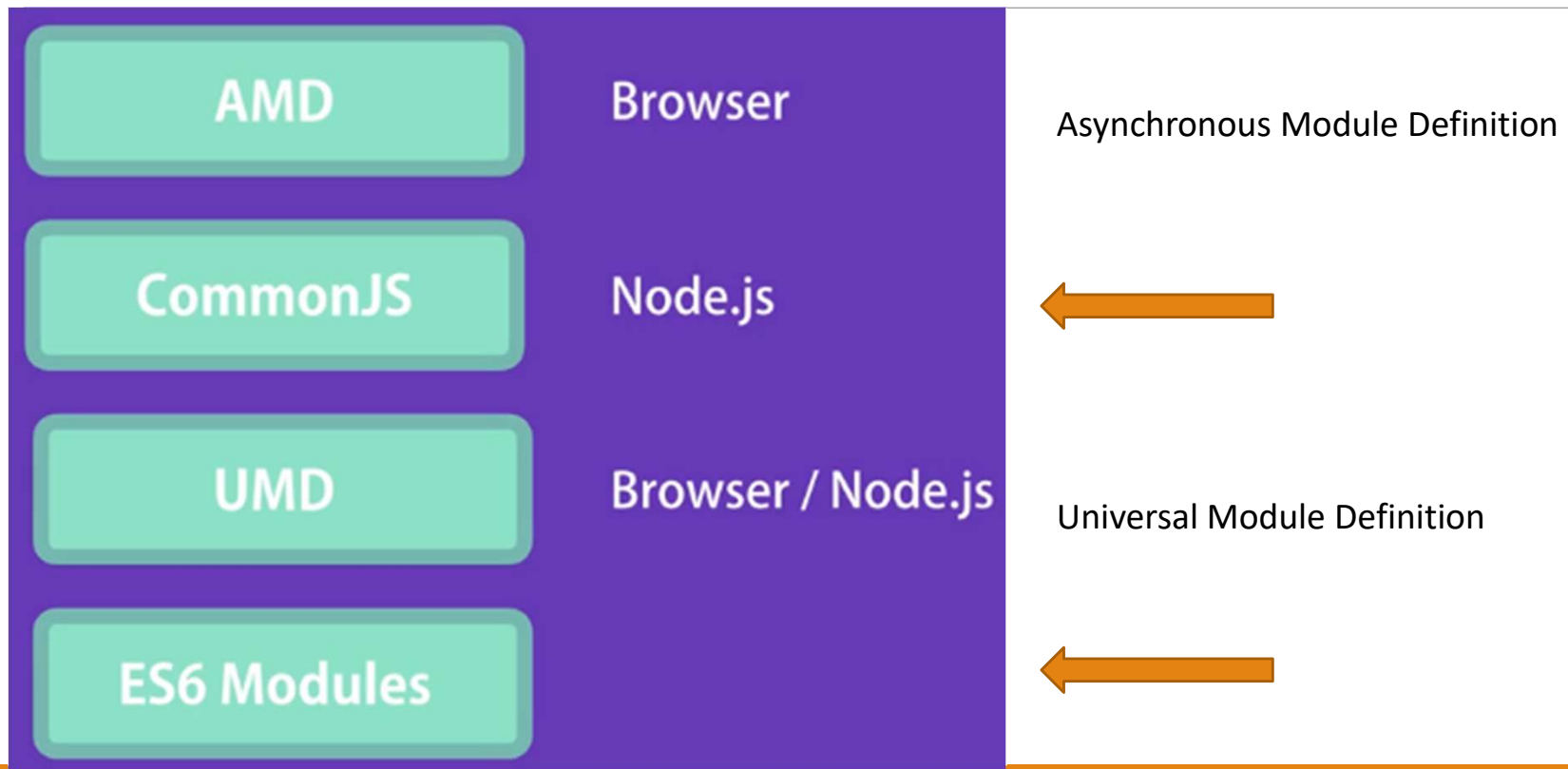# Inheritence

```
// Inheritance
class Triangle extends Shape {
  constructor(color) {
    // To call the base constructor
    super(color);
  }

  draw() {
    // Call the base method
    super.draw();

    // Do some other stuff here
  }
}
```

Module Formats

AMD	Browser
	Asynchronous Module Definition
CommonJS	Node.js ←
UMD	Browser / Node.js
	Universal Module Definition
ES6 Modules	←

Common JS

```
// CommonJS (Used in Node)
// Exporting
module.exports.Cirlce = Circle;
// Importing
const Circle = require('./circle');
```

ECMAScript 2015 (ES6) include:

- **Arrow Functions:** A more concise syntax for writing functions

- **let and const:** Block-scoped variable declarations.

- **Template Literals:** A new way to create strings using backticks (`).

- **Destructuring Assignment:** Easily extract values from arrays and objects.

- **Default Parameters:** Specify default values for function parameters.

- **Rest and Spread Operators:** Collect remaining parameters or spread elements.

- **Classes:** A more straightforward way to create and work with constructor functions.

- **Promises:** A standard for handling asynchronous operations.

- **Modules:** A standardized system for organizing and importing/exporting code between files.

- **Symbol and Iterators:** New data types and iteration protocols.

- **Map and Set Collections:** New data structures for key-value pairs and unique values.

Arrow Functions

BEFORE

```
hello = function() {
   return "Hello World!";
}
```

AFTER

```
hello = () => {
   return "Hello World!";
}
// Or
hello = () => "Hello World!";
// Or
hello = (val) => "Hello " + val;
//Or
hello = val => "Hello " + val;
```

Arrow functions allow us to write shorter function syntax:

Let Vs var

VAR

```
var x = 10;
// Here x is 10

{
var x = 2;
// Here x is 2
}

// Here x is 2
```

LET

```
let x = 10;
// Here x is 10

{
let x = 2;
// Here x is 2
}

// Here x is 10
```

Template Literals (String with back ticks)

```
let firstName = "John";
let lastName = "Doe";

let text = `Welcome ${firstName}, ${lastName}!`;


// multi line strings

let text1 =
`The quick brown fox
jumps over the lazy dog`;


let price = 10;
let VAT = 0.25;
let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
```

Destructring

BEFORE

const vehicles = ['mustang', 'f-150', 'expedition'];

// old way

const car = vehicles[0];

const truck = vehicles[1];

const suv = vehicles[2];

AFTER

const vehicles = ['mustang', 'f-150', 'expedition'];

const [car, truck, suv] = vehicles;

Destructure Objects

```
// Sample object
const person = {
  name: 'Alice',
  age: 25,
  city: 'Wonderland'
};
// Destructuring the object
const { name, age, city } = person;
// Using the extracted values
console.log(name); // Output: Alice
```

Default Parameters

BEFORE

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 2;
  }
}
```

AFTER

```
function myFunction (x, y = 2) {
  // function code
}
```

The Spread Operator (...)

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

```
const numbers = [1, 2, 3, 4, 5, 6];

const [one, two, ...rest] = numbers;
```

Spread Objects

```
// Original object
const person = {
  name: 'Alice',
  age: 25,
  city: 'Wonderland'
};
// Creating a shallow/deep copy
using the spread operator
```

```
const personCopy = { ...person };

// Modifying the copy
personCopy.age = 26;
// Displaying the original and
modified objects
console.log('Original object:',
person);
```

Shallow Copy Vs Deep Copy

```
// Original object with nested object

const originalObject = {

  name: 'John',

  details: {

    age: 30,

    city: 'Example City'

  }

};
```

```
// Shallow copy using the spread operator

// spread will not deep copy nested details
//object

const shallowCopy = { ...originalObject };

// Deep copy using JSON.parse and
//JSON.stringify

const deepCopy =
JSON.parse(JSON.stringify(originalObject));
```

JavaScript Iterables

```
for (const x of "W3Schools") {
  // code block to be executed
}
```

const myObject = {

  key1: 'value1',

  key2: 'value2',

};

for (const key in myObject) {

console.log(key); // Outputs: key1, key2

}

```
for (const x of [1,2,3,4,5]) {
  // code block to be executed
}
```

Non Conventional For Loops

ES6

```
// ES6 Modules (Used in Browser)
// Exporting
export class Square {}
// Importing
import {Square} from './square';
```

Babel

```
// We use Babel to transpile our modern
JavaScript code
// into code that browsers can understand
(typically ES5).
```

Web Pack

```
// We use Webpack to combine our JavaScript
files into a
// bundle.
```

Arrow Functions

Before

```
hello = function() {
  return "Hello World!";
}
```

After Arrow

```
hello = () => {
  return "Hello World!";
}
```

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
hello = val => "Hello " + val;
```

JavaScript Array find()

```
const ages = [3, 10, 18, 20];

function checkAge(age) {
  return age > 18;
}


  ages.find(checkAge);
```

JavaScript Array splice()

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];

fruits.splice(2, 0, "Lemon", "Kiwi");
```

Parameter	Description
index	Required. The position to add/remove items. Negative value defines the position from the end of the array.
howmany	Optional. Number of items to be removed.
item1, ..., itemX	Optional. New elements(s) to be added.

JavaScript Array map()

```
const numbers = [65, 44, 12, 4];
const newArr = numbers.map(myFunction);

function myFunction(num) {
  return num * 10;
}
```

Multiply every element in the array with 10:
650,440,120,40

JavaScript Array filter()

```
const ages = [32, 33, 16, 40];
const result = ages.filter(checkAdult);

function checkAdult(age) {
  return age >= 18;
}
```

//Return an array of all values in ages[] that are 18 or over:

Sync ASync

Problem

```
console.log("Before...");
setTimeout(function () {
  console.log("Reading a user from DB");
}, 2000);
console.log("After...");
// Whats the output
```

Output of Problem

Before...

After...

Reading a user from DB

ASynchronous

Its not
- Concurrent
- Multi Threaded

It is
- Just a function scheduled to be called in future

Output

```
console.log("Before");

const user = getUser();

console.log(user);

console.log("After");

//Output

//Before

//dummy

//After

//DB Query entertained
```

```
function getUser(){

 setTimeout(function(){

 console.log('DB Query entertained');

 return {id:9,name:'usman'}

 },1000);

return "dummy";

}
```

Patterns for Dealing with Asynchronous Code

Callback

Promises

Async/await

CallBack

```javascript
function getUser(id, callback) {
  setTimeout(function () {
    console.log("Reading User");
    callback({ id: id, name: "Usman" });
  }, 2000)
}
```

CallBack

```
console.log("Before");
getUser(1, function (userObj) {
  console.log("Received User");
  console.log(userObj);
});
console.log("After");
```

Sync Vs Async

ASYNC

```
fs.readdir(__dirname, (err, files) => {
  if (err) console.log(err);
  else {
    console.log("\nCurrent directory filenames:");
    files.forEach(file => {console.log(file);})
  }
})
```

SYNC

```
filenames = fs.readdirSync(__dirname);
console.log("\nCurrent directory filenames:");
filenames.forEach(file => {
  console.log(file);
});
```

Imagine this ☹ (CallBack Hell)

```javascript
getUser(1, (user) => {
  getRepositories(user.gitHubUsername, (repos)        =>
{
    getCommits(repos[0], (commits) => {
    console.log(commits);
    })
  })
});
//You can use named Functions but still NOOOOO
```

Promises -a function that returns a promise

```
function doSomething() {
    return new Promise((resolve, reject) => {
        // Simulating an asynchronous operation (e.g., fetching data from a server)
        setTimeout(() => {
            const data = "Some data fetched from the server";
            resolve(data); // Resolve the promise with the data
        }, 2000); // Simulate a delay of 2 seconds
    });
}
```

Using Promise

```
doSomething()
    .then((result) => {
        console.log("Promise resolved with data:", result);
    })
    .catch((error) => {
        console.error("An error occurred:", error);
    })
    .finally(() => {
        console.log("Promise chain completed"); // This block will execute regardless of success or failure
    });
```

Promise

```javascript
const p = new Promise(function(resolve,reject){
if (true) resolve({name:"hareem"});
else reject(new Error("Hareem is naughty"));
});
p.then((result)=>{
console.log(result.name);
});
p.catch((error)=>{console.log("Error Caught"+error.message)});
```

Promise

```
const p = new Promise((resolve, reject) => {
 // Kick off some async work
 // ...
 setTimeout(() => {
 resolve(1); // pending => resolved, fulfilled
 reject(new Error('message')); //pending => rejected
 }, 2000);
});
```

Using Promise

```
 p.then(result => console.log('Result',
result))
 .catch(err => console.log('Error',
err.message));
```

Beauty in Code

```javascript
getUser(1)

.then(user => getRepositories(user.gitHubUsername))

.then(repos => getCommits(repos[0]))

.then(commits => console.log('Commits', commits))

.catch(err => console.log('Error', err.message));
```

Async and Await approach

```javascript
async function displayCommits() {
 try {
   const user = await getUser(1); //code execution will halt here
   const repos = await getRepositories(user.gitHubUsername);
   const commits = await getCommits(repos[0]);
   console.log(commits);
 }
 catch (err) {
   console.log('Error', err.message);
 }
}
```

Resolved Promises (For Testing)

```
Promise.reolve(1);
Promise.reject(new Error(''));
```

Running Promses in parallel

```
Promise.all([p1, p2]);// When all promises
are resolved

Promise.race([p1, p2]);// When any one
finished first
```

Sample Code

https://1drv.ms/f/s!AtGKdbMmNBGd0V1N14IfBGU1Npoi