# 13 Aug 2023 - Lecture JavaScript

## JAVASCRIPT

- 组成：

ECMAScript + DOM(页面文档模型) + BOM(浏览器文档模型)

- 行内式JS

`<input type="button" value="xxx" onclick="alert('Hello World')" />`

- 内嵌JS

```

<script>

alert('Hello World');

</script>

```

- 外部JS

`<script src="my.js"></script>`

- Variable

```

1. // 只声明，不赋值

var age;

console.log(age);

// -> undefined

```

```

2. // 不声明，不赋值，直接使用

console.log(age);

// -> error

```

```

3. // 不声明，只赋值

age = 10;console.log(age);

// -> 10

```

```

4. //局部作用域

function fn() {

var num1 = 20;

console.log("num1", num1); //20

}

fn();

console.log("num1", num1);//Uncaught ReferenceError: num1 is not defined

```

```

// 5. 在{}中间的语句，if,switch条件语句，for,while循环中，var没有块级作用域

if(true) {

var a = 1;

}

for(var i = 0; i < 10; i++) {

console.log("a", a);

console.log("i", i);

}

```

```

//6. 在没有块级作用域之前，内层变量可能会被外层覆盖

var tmp = new Date();

function fn() {

console.log("tmp", tmp);//undefined,因为变量声明被提升，但是未赋值

if(false){

var tmp = "hello world";

}

}

fn();

```

```

//7.块级作用域 let const

if(true) {

let x = 10;

const y = 20;

console.log("x", x); // 10

console.log("y", y); // 20

}

console.log("x", x); // undefined

console.log("y", y); // undefined

```

```

//8. 暂时性死区

let foo = "abc";

console.log(foo);// abc

```

```

//9.const定义的值不能改变

const age = 10;

age = 20 //Uncaught TypeError: Assignment to constant variable.

```

- Name convention

- 字母，数字，下划线，美元符号组成

- 严格区分大小写

- 不能以数字开头

- 不能是关键字保留字eg, break, case, key...

- 驼峰命名法

- 数据类型：

- 类型转换

```

var num = 123;

var str = "abcdef";

var bool = true;

var arr = [1, 2, 3, 4];

var obj = { name: "wenzi", age: 25 };

var func = function () { console.log("this is function");};

var und = undefined;

var nul = null;

var date = new Date();

var reg = /^[a-zA-Z]{5,20}$/;

var error = new Error();

//number,string,boolean,function, undefined 用typeof

console.log("num", typeof num); // number

console.log("str", typeof str); // string

console.log("bool", typeof bool); // boolean

console.log("arr", typeof arr); // object

console.log("obj", typeof obj); // object

console.log("func", typeof func); // function

console.log("und", typeof und); // undefined

console.log("nul", typeof nul); // object

console.log("date", typeof date); // object

console.log("reg", typeof reg); // object

console.log("error", typeof error); // object

```

```

// instanceof不能判断值类型，但是引用类型可以，值得注意的是arr和obj在instanceof Object的时候的值都是true，

//这就导致判断是对象时不准确

console.log(num instanceof Number); // false

console.log(str instanceof String); // false

console.log(bool instanceof Boolean); // false

console.log(arr instanceof Array); // false

console.log(obj instanceof Object); // true

console.log(func instanceof Function); // true

console.log(date instanceof Date); // true

console.log(reg instanceof RegExp); // true

console.log(error instanceof Error); // true

console.log(nul instanceof Object); // false

// (2) Array.isArray(参数); H5新增的方法 ie9以上版本支持

console.log("arr", Array.isArray(arr)); // true

console.log("obj", Array.isArray(obj)); // false

```

- - String

```

// 1. 把数字型转换为字符串型 变量.toString()

var num = 10;

var str = num.toString();

console.log(str, typeof(str));

// 2. 利用 String(变量)

console.log(typeof String(num));

// 3. 利用 + 拼接字符串的方法实现转换效果 隐式转换

var str2 = num + "";

console.log(str2, typeof(str2));

// 1. 检测获取字符串的长度 length

var str = "my name is andy";

console.log(str.length); // 15

// 2. 字符串的拼接 + 只要有字符串和其他类型相拼接 最终的结果是字符串类型

console.log("hello" + "world"); // hello world

console.log("Ben" + 18); // Ben18

console.log("Ben" + true); // Bentrue

console.log(12 + 12); // 24

console.log("12" + 12); // 1212

```

- - Number

```

// 1. parseInt(变量) 可以把 字符型的转换为数字型 得到是整数

console.log(parseInt("3.14")); // 3

console.log(parseInt("3.94")); // 3

console.log(parseInt("120px")); // 120

console.log(parseInt("rem120px")); // NaN - Not a Number

// 2. parseFloat(变量) 可以把 字符型的转换为数字型 得到是小数 浮点数

console.log(parseFloat("3.14")); // 3.14

console.log(parseFloat("120px")); // 120

console.log(parseFloat("rem120px")); // NaN

// 3. 利用 Number(变量)

var str = "123";

var num = Number(str);

console.log(typeof(num)); // number

// 4. 利用了算数运算 - \* / 隐式转换

let value = '12' - 0;

console.log(typeof(value), value);// number 12

let value1 = '12' \* 2;

console.log(typeof(value1), value1);// number 24

```

- - Boolean

```

console.log(Boolean("")); // false

console.log(Boolean(0)); // false

console.log(Boolean(NaN)); // false

console.log(Boolean(null)); // false

console.log(Boolean(undefined)); // false

console.log(Boolean("123")); // true

console.log(Boolean("hello")); // true

```

- - Operators

```

// 1. % 取余 （取模）

console.log(4 % 2); // 0

console.log(5 % 3); // 2

console.log(3 % 5); // 3

// 2. 浮点数 算数运算里面会有问题

console.log(0.1 + 0.2); // 0.30000000000000004

console.log(0.07 \* 100); // 7.000000000000001

// 3. 不能直接拿着浮点数来进行相比较 是否相等

//因为在计算机运行过程中，需要将数据转化成二进制，然后再进行计算,所以计算产生误差

var num = 0.1 + 0.2;

console.log(num == 0.3); // false

console.log(num); // 0.30000000000000004

```

> 解决浮点数计算精度问题

1. 转化成证书，再相加之后转回小数。

```

let num2 = (0.1 \* 10 + 0.2 \* 10) / 10;

console.log(num2);

```

2. toFixed()

`console.log(num.toFixed(1));`

- - 前置递增 & 后置递增（单独使用效果是一样的）

`p++`先返回值后+1

`++p`先+1后返回值

`console.log("p", ++p + 10); // 21`

`console.log("p", p++ + 10); // 20`

- - 比较

```

console.log(3 >= 5);

console.log(2 <= 4);

//1. 我们程序里面的等于符号 是 == 默认转换数据类型 会把字符串型的数据转换为数字型 只要求值相等就可以

console.log(3 == 5);

console.log(18 == 18);

console.log(18 == "18");

console.log(18 != 18);

// 2. 我们程序里面有全等 一模一样 要求 两侧的值 还有 数据类型完全一致才可以 true

console.log(18 === 18);

console.log(18 === "18");

// 3. 逻辑与 && and 两侧都为true 结果才是 true 只要有一侧为false 结果就为false

console.log(3 > 5 && 3 > 2);

console.log(3 < 5 && 3 > 2);

// 4. 逻辑或 || or 两侧都为false 结果才是false 只要有一侧为true 结果就是true

console.log(3 > 5 || 3 > 2);

console.log(3 > 5 || 3 < 2);

// 5. 逻辑非 not ！

console.log(!true);

```

- - 运算符优先级

```

1. `()`

2. ++ -- !

3. \* / % 后 + -

4. > >= < <=

5. == != === !==

6. 先&& 后||

7. =

8. ,

```

- Array

```

var arr1 = ["red", "green", "blue"]; //index是从0开始

```

添加删除数组元素方法

1. push() 在我们数组的末尾 添加一个或者多个数组元素 push 推

```

// (1) push 是可以给数组追加新的元素

// (2) push() 参数直接写 数组元素就可以了

// (3) push完毕之后，return的结果是 新数组的长度

// (4) 原数组也会发生变化

let arr = [1, 2, 3];

console.log(arr.push(4, "hello")); // return 5

```

2. unshift 在我们数组的开头 添加一个或者多个数组元素

```

// (1) unshift是可以给数组前面追加新的元素

// (2) unshift() 参数直接写 数组元素就可以了

// (3) unshift完毕之后，返回的结果是 新数组的长度

// (4) 原数组也会发生变化

let arr3 = ["e", "f", "g"];

console.log("arr3", arr3.unshift("red")); // return 4

console.log("arr3", arr3);

```

3. pop() 它可以删除数组的最后一个元素

```

// (1) pop是可以删除数组的最后一个元素 记住一次只能删除一个元素

// (2) pop() 没有参数

// (3) pop完毕之后，返回的结果是 删除的那个元素

// (4) 原数组也会发生变化

let arr2 = [1, 2, "c"];

console.log("arr2", arr2.pop()); // return c

console.log("arr2", arr2); // [1, 2]

```

4. shift() 它可以删除数组的第一个元素

```

// (1) shift是可以删除数组的第一个元素 记住一次只能删除一个元素

// (2) shift() 没有参数

// (3) shift完毕之后，返回的结果是 删除的那个元素

// (4) 原数组也会发生变化

let arr4 = ["aa", "bb", "cc"];

console.log("arr4", arr4.shift()); // aa

console.log("arr4", arr4);

```

5. indexOf(数组元素)

```

// 返回数组元素索引号方法 作用就是返回该数组元素的索引号 从前面开始查找

// 它只返回第一个满足条件的索引号

// 它如果在该数组里面找不到元素，则返回的是 -1

var arr = ["red", "green", "blue", "pink", "blue"];

console.log(arr.indexOf('green')); // 1

console.log(arr.indexOf('white')); // -1

//lastIndexOf()从数组后面开始查找

console.log(arr.indexOf('bule')); // 4

```

- Object

```

// 遍历对象

var obj = {

name: "Ben",

age: 18,

sex: "male",

fn: function () {},

};

console.log(obj.name); // Ben

console.log(obj['age']); // 18

// for in 遍历我们的对象

for (let key in obj) {

console.log("key", key);

console.log("value", obj[key]);

}

// Object.keys

const res = Object.keys(obj);

console.log("res", res);

const value = Object.keys(obj).includes('name') && obj.name;

console.log("value", value);

Object.keys(obj).forEach((key) => {

console.log(key, obj[key]);

})

// Object.entries() method returns an array of arrays, each containing a key-value pair of the object'

// [[], []]

const res2 = Object.entries(obj);

console.log("res2", res2);

Object.entries(obj).forEach(([key, value]) => {

console.log(key, value);

});

Object.values(obj);

```

- map && filter && reduce

- map对每一项都进行修改，不会改变原数组而是返回一个新数组，新数组的长度是不变的。

```

const newNumbersMap = numbers.map((item) => {

return item\*2;

});

console.log(newNumbersMap);

```

- filter不会修改原数组的长度而是返回一个新数组，新书组会返回符合要求的items。

```

const newNumbersFilter = numbers.filter((item) => {

return item < 10;

});

console.log(newNumbersFilter);

```

- reduce 会修改原数组长度。

```

const newNumbersReduce = numbers.reduce((item) => {

return item < 10;

});

console.log(newNumbersReduce);

```

- function

```

// function expression

const getSum2 = function() {};

// 函数形参实参个数匹配

// function declaration

function getSum(num1, num2){

console.log(num1 + num2);

}

// 1. 如果实参的个数和形参的个数一致 则正常输出结果

getSum(1, 2);

// 2. 如果实参的个数多于形参的个数 会取到形参的个数

getSum(1, 2, 3);

// 3. 如果实参的个数小于形参的个数 多于的形参定义为undefined

// 形参可以看做是不用声明的变量 num2 是一个变量但是没有接受值 结果就是undefined

getSum(1);

// 建议 尽量让实参的个数和形参相匹配

```

- shallow copy(浅拷贝) - 拷贝内存地址：

```

//shallow copy:

//lodash \_.clone方法

let arr = ["apple", "banana", "orange"];

const arr1 = [...arr]; //spread operator

const arr2 = Array.from(arr);

const arr3 = arr.slice();

console.log(arr1, arr2, arr3);

//spread operator

let objOne = { name: "john", age: 30 };

const objOneShallowCopy = {...objOne};

console.log(objOneShallowCopy);

//Object.assign()

let obj = {

id: 1,

name: "andy",

msg: {

age: 18,

},

};

let copiedObj = Object.assign({}, obj);

console.log(copiedObj);

obj.msg.age = 20;

console.log("obj", obj);

console.log("copied", copiedObj);

copiedObj.msg.age = 257;//也会将原数组进行更改,会与原数组互相影响。

console.log("copiedModified", copiedObj);

```

- deep copy(深拷贝) - 完全拷贝一个Obj:

```

// 深拷贝 JSON.parse(JSON.stringify(object))

//lodash 的 cloneDeep \_.cloneDeep

//structuredClone()

const obj = {

person: {

name: "lin",

},

};

//structuredClone()

const newObj = structuredClone(obj);

obj.person.name = "Chris";//不会影响到深拷贝出来的newObj

console.log("Obj", obj);

console.log("newObj", newObj);

```