经典题

初级题

JS

1. 下面输出的结果是什么？

```
var a = [1,2], b = [3], c = 4;
(function(a1,b1,c1){
    a1 = [];
    b1[0] = 99;
    c1 = 999;
})(a,b,c);
console.log(a,b,c)    // [1,2],[99],4
```

2. 下面输出的结果是什么？

```
(function(){
    var a = b = 99;
})();
console.log(a,b); // undefined · 99
```

3. 下面输出的结果是什么？

```
function fn(){
    var a = 0;
    return function(){
        a++;
        console.log(a);
    }
}
var obj1 = fn();
obj1(); // 1
obj1(); // 2
var obj2 = fn();
obj2(); // 1
obj2(); // 2
```

4. 下面输出的结果是什么？

```
var a = 11;
if(!(a in window)){ // 相当于 11 in window 所谓为 false
```

```
    var b = 22;
}
console.log(b); // 22

PS: a 通过 const、let 声明， b 通过 const、let 声明。
```

5. 下面输出的结果是什么？

```
var name = "Tom";
var obj = {name:"Tony"};
function _console(){
    var name = "Jack";
    return function(){
        console.log(this.name);
    }
}

_console()();  // Tom
_console.call(obj)(); // Tom
_console().call(obj); // Tony
```

6. 下面输出的结果是什么？

```
function _console(){
    for(var i = 0; i < 3; i++){
        setTimeout(function(){
            console.log(i); // 3 3 3
        }, 1000 * i)
    }
}
_console();
```

7. 下面输出的结果是什么？

```
(function(){
    a = 5;
    console.log(window.a); // undefined
    var a = 10;
    console.log(a); // 10
})()
```

8. 下面输出的结果是什么？

视频讲解

```
(function(){
   console.log(1, foo); // 1, function foo(){};
   console.log(2, bar); // 2, undefined
   var foo = "hello";
   console.log(3, foo); // 3, hello
   var bar = function(){};
   function foo(){};
   console.log(4, bar); // 4, function (){}
})();

PS: 函数声明提升会高于变量声明提升；
```

9. 下面输出的结果是什么？

```
function Foo (){
    getName = function(){
     console.log(1);
    }
    return this;
}
Foo.getName = function(){
    console.log(2);
}
Foo.prototype.getName = function(){
    console.log(3);
}
var getName = function () {
    console.log(4);
}
function getName(){
    console.log(5);
}
Foo.getName(); // 2      构造函数方法优先原型上的方法
getName(); // 4     当变量声明时没有赋值或初始化，函数声明的优先级高于变量，否则变量优先级高于函数。
Foo().getName(); // 1  先调用Foo()在通过链式调用Foo里面的getName方法，所以就是 1 并返回并返回当前作用域
getName();  // 1   getName的重新赋值是指向的闭包里的getName
new Foo.getName(); // 2   相当 new (Foo.getName())，先执行 Foo.getName()；
new Foo().getName(); // 3  调用原型上的方法
new new Foo().getName(); // 3  相当 new (new Foo().getName())，先执行new Foo().getName 调用先输出alert(3)；
```

10. 下面输出的结果是什么？

```
 var text = function a(){
    console.log(typeof a);  // function
```

```
}
text();
console.log(typeof a); // undefined
```

PS：函数声明在赋值给变量， 用函数声明调用只能在函数内部使用；

11. 隐**式**转换**？**

1. "+" 操作符， 两边有一个为 String 类型隐式转换为字符串；其他情况下转换为数字；
```
console.log(1 + '23'); // '123'
console.log(1 + false); // 1
console.log('1' + false); // '1false'
console.log(true + false); // 1
console.log(NaN + false); // NaN
```
2. "-"、"*"、"/"操作符，会转换为数字；
```
console.log(25 - '23'); // 2
console.log(1 * false); // 0
console.log(1 / 'aa'); // NaN
```
3. '==' 操作符，转换为数字进行比较；
```
console.log(3 == true); // 3 == 1 => false
console.log('0' == false); // 0 == 0 => true
console.log('0' == 0); // 0 == 0 => true
```
4. '<'和'>'操作符，对 String 类型的字母、汉字进行比较，按照 ASCII码 的顺序
```
console.log('a'>'b'); // 97 > 98 => false
console.log('B' < 'A'); // 65 < 64 => false
```
其他情况下，换为为数字进行比较；
```
console.log('12' < 13); // 12 < 13 => true
console.log(false > -1); // 0 > -1 => true
```
5. 对象
```
var a = {};
console.log(a > 2);
```
转换过程：
```
console.log(a.valueOf());
console.log(a.toString()); // [Object Object]
console.log(Number(a.toString())) // NaN
```

12. JS 运行结**果**

1. true + false ==> 1 + 0 ===> 1

2. [,,,].length ==> [empty,empty,empty,] ===> 3 //最后一个逗号是尾随逗号

3. [1,2,3] + [4,5,6] ==> [1,2,3].toString() + [4,5,6].toString() ==> '1,2,3' + '4,5,6' ===> '1,2,34,5,6'

4. 0.2 + 0.1 === 0.3 ===> false; // 0.1 + 0.2 的结果并不完全是 0.3

5. 10,2 ==> /*从左到右,返回最后一个数*/ ===> 2

```
    6. !!"" ==> /* ''、null、0、undefined 都是假值 */ ===> false

    7. +!![] ==> /* 数组、空数组 都是真值 */ ==> +true ===> 1

    8. true == 'true' ==> /* 根据 '==' 转换为数字 */ ==> Number(true) ==
Number('true') ==> 1 == NaN ===> false

    9. 010 - 03 ==> 8 - 3 ===> 5 /* 以 0b 开头为二进制、以 0 开头为八进制、以 0x 开头
为十六进制 */

    10. ''- -'' ==> /* 空字符为false */  ==> 0 - 0 ===> 0

    11. null + 0 ==> /* null为false */ ==> 0 + 0 ===> 0

    12. 0/0 ===> NaN

    13. true++ ===> '报错语法'

    14. ''-1 ==> Number('') - 1 ==> 0 - 1 ===> 1

    15. (null - 1) - "1" ==> (0-1)-1 ===> -2

    16. 5 + !5 + !!5 ==> 5 + 0 + 1 ===> 6

    17. [] + [1] + 2 ==> '' + '1' + 2 ===> '12'

    18. 1+2+"3" ==> /* 从左到右依次执行 */ ==> 3 + '3' ===> '33'
```

13. ☆冒泡排序☆

```
    var arr = [1,59,461,64,86,6,46,99,61,3,88,2];
    for(var i = 0; i<= arr.length;i++){
        for(var j = 0; j <= i; j++){
            if(arr[i] < arr[j]){
                var tamp = arr[i];
                arr[i] = arr[j];
                arr[j] = tamp;
            }
        }
    }
    console.log(arr);
```

14. ☆深拷贝☆

```
    function deepClone(obj) {
            let objClone = Array.isArray(obj) ? [] : {};
            if (obj && typeof obj === "object") {
```

```
                for (key in obj) {
                    if(obj.hasOwnProperty(key)){
                        if(obj[key]&& typeof obj[key] === 'object'){
                            objClone[key] = deepClone(obj[key]);
                        }else{
                            objClone[key] = obj[key];
                        }
                    }
                }
            }
            return objClone;
        }
        var arr = [1, 2, 3];
        var arr2 = deepClone(arr);
        arr2[0] = 99;
        console.log(arr);
        console.log(arr2);
```

JS 执行结果

1. 定时器中的变量 a 没有使用 var 声明，访问的是 window 中的 a；
   用 let、const 就会报错；

```
var a = 0;
setTimeout(() => {
    console.log(a)
    a = 10000;
}, 1000);
a = 66;
// 66
```

2. 封装一个函数，传入一段字符串，总计出现最多的字符

```
var str = '123121';
function fn(str) {
    var obj = {};
    for (var i = 0; i < str.length; i++) {
        if(!obj[str[i]]){
            obj[str[i]] = 1
        }else{
            obj[str[i]]++;
        }
    }
    return obj;
}
console.log(fn(str));
```

3. 实现一个字符的反转

```
var str = '123456789';
console.log(str.split("").reverse().join(''));
```

4. !! a1 = []; 不会修改原始的数组，只会开辟新的空间 !!

```
var a = [1,2],b=[3],c= 9;
function fn(a1,b1,c1){
```

```
        a1 = [];
        b[0] = 99;
        c = 88;
    }
    fn(a,b,c);
    console.log(a,b,c); // [1,2],[99],88
```

5. !!变量提升!!

```
    if(!('a' in window)){
        var a = 10;
    }
    console.log(a); // undefined
```

6. !!b没有使用关键字声明，所以是全局变量!!

```
    var a = b = 3;
    (function(){
        var a = b = 66;
    })()
    console.log(a,b);  // 3,66
```

7. !!逐层上找!!

```
    var str = 'ss'
    function fn (){
        var str = 'aa';
        function f(){
            return str;
        }
        return f();
    }
    console.log(fn());
```

8. !! var 存在变量提升的问题，解决：1.var 换成let 2.使用立即执行函数包裹定时器 !!

```
   for (var i = 0; i < 5; i++) {
        setTimeout(() => {
            console.log(i)  // [5] 6
        }, 500);
    }
```

9. !! 执行到第三行的时候，'.' 优先于 '=' ， a 重新开辟了一块空间 **********!!

```
    var a = {n:1};
    var b = a;
    a.x = b = {n:2};
    console.log(a) // {n:2}
    console.log(b) // {n:1,x:{n:2}}
```

10.

```
    var fun = (function (a) {
        this.a = a;
        return function(a){
            a += this.a;
            return a;
        }
    })(function (a,b) {
        return a;
```

```
    }(1, 2))
    fun(88); // 这里调用的是内部return出来的函数。

11. // 函数的length是表示有几个参数
    function foo(a,d,c){}
    delete foo.length;  // 将参数全部删除
    console.log(foo.length)

12.
    var k = 0;
    for (var i = 0, j = 0; i < 10, j < 6; i++, j++) {
        k = k + i + j;
    }
    console.log(k) // 30

13. function Fn(num) {
        this.num = num;
    }
    Fn.prototype.fun = function() {
        console.log(this.num)
    }
    var a = new Fn(100);
    a.num = 200;
    a.__proto__.num = 300; // 直接在原型上新添加num属性，
    a.__proto__.fun(); // 直接调用原型上原型上的方法， 300
    a.fun(); // a 对象中存在num属性， 200

14.
    function fn(o) {
        o.str = '123456';
        o = new Object();  // 和 obj 脱离关系，重新开辟空间
        o.str = '987654';
    }
    var obj = new Object();
    fn(obj);
    console.log(obj.str);

15.
    class A {}
    class B extends A {}
    let a = new A();
    let b = new B();
    console.log(a.__proto__ === A.prototype); // true
    console.log(b.__proto__ === B.prototype); // true
    console.log(B.__proto__ === A); // true
    console.log(B.prototype.__proto__ === A.prototype); // true
    console.log(b.__proto__.__proto__ === A.prototype); // true
```