问题1:JavaScript 中 `undefined` 和 `not defined` 的区别

JavaScript 未声明变量直接使用会抛出异常：`var name is not defined`，如果没有处理异常，代码就停止运行了。
但是，使用`typeof undeclared_variable`并不会产生异常，会直接返回 `undefined`。

```
var x; // 声明 x
console.log(x); //output: undefined
```

```
console.log(typeof y); //output: undefined
```

```
console.log(z); // 抛出异常: ReferenceError: z is not defined
```

问题2:下面的代码输出什么?

```
var y = 1;
if (function f(){}) {
    y += typeof f;
}
console.log(y);
```

正确的答案应该是 `1undefined`。

JavaScript中if语句求值其实使用`eval`函数，`eval(function f(){})` 返回 `function f(){}` 也就是 `true`。

下面我们可以把代码改造下，变成其等效代码。

```
var k = 1;
if (1) {
    eval(function foo(){});
    k += typeof foo;
}
console.log(k);
```

上面的代码输出其实就是 `1undefined`。为什么那？我们查看下 `eval()` 说明文档即可获得答案

> 该方法只接受原始字符串作为参数，如果 string 参数不是原始字符串，那么该方法将不作任何改变地返回。

恰恰 `function f(){}` 语句的返回值是 `undefined`，所以一切都说通了。

注意上面代码和以下代码不同。

```
var k = 1;
if (1) {
    function foo(){};
    k += typeof foo;
}
console.log(k); // output 1function
```

问题3:在JavaScript中创建一个真正的private方法有什么缺点?

每一个对象都会创建一个private方法的方法，这样很耗费内存

观察下面代码

```
var Employee = function (name, company, salary) {
    this.name = name || "";
    this.company = company || "";
    this.salary = salary || 5000;

    // Private method
    var increaseSalary = function () {
        this.salary = this.salary + 1000;
    };

    // Public method
    this.dispalyIncreasedSalary = function() {
        increaseSlary();
        console.log(this.salary);
    };
};

// Create Employee class object
var emp1 = new Employee("John","Pluto",3000);
// Create Employee class object
var emp2 = new Employee("Merry","Pluto",2000);
// Create Employee class object
var emp3 = new Employee("Ren","Pluto",2500);
```

在这里 emp1,emp2,emp3都有一个increaseSalary私有方法的副本。

所以我们除非必要，非常不推荐使用私有方法。

问题4:JavaScript中什么是闭包？写出一个例子

老生常谈的问题了，闭包是在一个函数里声明了另外一个函数，并且这个函数访问了父函数作用域里的变量。

下面给出一个闭包例子，它访问了三个域的变量

- 它自己作用域的变量
- 父函数作用域的变量
- 全局作用域的变量

```
var globalVar = "abc";

// Parent self invoking function
(function outerFunction (outerArg) { // begin of scope outerFunction
    // Variable declared in outerFunction function scope
    var outerFuncVar = 'x';
    // Closure self-invoking function
    (function innerFunction (innerArg) { // begin of scope innerFunction
        // variable declared in innerFunction function scope
        var innerFuncVar = "y";
        console.log(
            "outerArg = " + outerArg + "\n" +
            "outerFuncVar = " + outerFuncVar + "\n" +
            "innerArg = " + innerArg + "\n" +
            "innerFuncVar = " + innerFuncVar + "\n" +
            "globalVar = " + globalVar);

    }// end of scope innerFunction)(5); // Pass 5 as parameter
}// end of scope outerFunction )(7); // Pass 7 as parameter
innerFunction is closure that is defined inside outerFunc
```

输出很简单：

```
outerArg = 7
outerFuncVar = x
innerArg = 5
innerFuncVar = y
globalVar = abc
```

问题5:写一个mul函数，使用方法如下。

```
console.log(mul(2)(3)(4)); // output : 24
console.log(mul(4)(3)(4)); // output : 48
```

答案直接给出：

```
function mul (x) {
    return function (y) { // anonymous function
        return function (z) { // anonymous function
            return x * y * z;
        };
    };
}
```

简单说明下： mul 返回一个匿名函数，运行这个匿名函数又返回一个匿名函数，最里面的匿名函数可以访问 x,y,z 进而算出乘积返回即可。

对于JavaScript中的函数一般可以考察如下知识点：

- 函数是一等公民
- 函数可以有属性，并且能连接到它的构造方法
- 函数可以像一个变量一样存在内存中
- 函数可以当做参数传给其他函数
- 函数可以返回其他函数

问题6:JavaScript怎么清空数组?

如

```
var arrayList = ['a','b','c','d','e','f'];
```

怎么清空 `arrayList`

方法1

```
arrayList = [];
```

直接改变arrayList所指向的对象，原对象并不改变。

方法2

```
arrayList.length = 0;
```

这种方法通过设置length=0 使原数组清除元素。

方法3

```
arrayList.splice(0, arrayList.length);
```

和方法2相似

问题7:怎么判断一个object是否是数组(array)?

方法1

使用 Object.prototype.toString 来判断是否是数组

```
function isArray(obj){
    return Object.prototype.toString.call( obj ) === '[object Array]';
}
```

这里使用call来使 toString 中 this 指向 obj。进而完成判断

方法二

使用 原型链 来完成判断

```
function isArray(obj){
    return obj.__proto__ === Array.prototype;
}
```

基本思想是利用 实例如果是某个构造函数构造出来的那么 它的 `__proto__`是指向构造函数的 `prototype`属性。

方法3

利用JQuery

```
function isArray(obj){
    return $.isArray(obj)
}
```

JQuery isArray 的实现其实就是方法1

问题8:下面代码输出什么?

```
var output = (function(x){
    delete x;
    return x;
})(0);

console.log(output);
```

输出是 `0`。 `delete` 操作符是将object的属性删去的操作。但是这里的 `x` 是并不是对象的属性， `delete` 操作符并不能作用。

问题9:下面代码输出什么?

```
var x = 1;
var output = (function(){
    delete x;
    return x;
})();

console.log(output);
```

输出是 `1`。`delete` 操作符是将object的属性删去的操作。但是这里的 `x` 是并不是对象的属性， `delete` 操作符并不能作用。

问题10:下面代码输出什么?

```
var x = { foo : 1};
var output = (function(){
    delete x.foo;
    return x.foo;
})();

console.log(output);
```

输出是 `undefined`。x虽然是全局变量，但是它是一个object。`delete`作用在`x.foo`上，成功的将`x.foo`删去。所以返回`undefined`

问题11:下面代码输出什么?

```
var Employee = {
    company: 'xyz'
}
var emp1 = Object.create(Employee);
delete emp1.company
console.log(emp1.company);
```

输出是 `xyz`，这里的 emp1 通过 prototype 继承了 Employee的 company。emp1自己并没有company属性。所以delete操作符的作用是无效的。

问题12:什么是 `undefined x 1`？

在chrome下执行如下代码，我们就可以看到`undefined x 1`的身影。

```
var trees = ["redwood","bay","cedar","oak","maple"];
delete trees[3];
console.log(trees);
```

当我们使用 delete 操作符删除一个数组中的元素，这个元素的位置就会变成一个占位符。打印出来就是`undefined x 1`。
注意如果我们使用`trees[3] === 'undefined × 1'`返回的是 `false`。因为它仅仅是一种打印表示，并不是值变为`undefined x 1`。

问题13:下面代码输出什么?

```
var trees = ["xyz","xxxx","test","ryan","apple"];
delete trees[3];

console.log(trees.length);
```

输出是5。因为delete操作符并不是影响数组的长度。

问题14:下面代码输出什么?

```
var bar = true;
console.log(bar + 0);
console.log(bar + "xyz");
console.log(bar + true);
console.log(bar + false);
```

输出是

```
1
truexyz
2
1
```

下面给出一个加法操作表

- Number + Number -> 加法
- Boolean + Number -> 加法
- Boolean + Boolean -> 加法
- Number + String -> 连接
- String + Boolean -> 连接
- String + String -> 连接

问题15:下面代码输出什么?

```
var z = 1, y = z = typeof y;
console.log(y);
```

输出是 `undefined`。js中赋值操作结合律是右至左的，即从最右边开始计算值赋值给左边的变量。

上面代码等价于

```
var z = 1
z = typeof y;
var y = z;
console.log(y);
```

问题16:下面代码输出什么?

```
var foo = function bar(){ return 12; };
typeof bar();
```

输出是抛出异常，bar is not defined。
如果想让代码正常运行，需要这样修改代码：

```
var bar = function(){ return 12; };
typeof bar();
```

或者是

```
function bar(){ return 12; };
typeof bar();
```

明确说明这个下问题

```
var foo = function bar(){
    // foo is visible here
    // bar is visible here
    console.log(typeof bar()); // Work here :)
};
// foo is visible here
// bar is undefined here
```

问题17:两种函数声明有什么区别?

```
var foo = function(){
    // Some code
};
function bar(){
    // Some code
};
```

foo的定义是在运行时。想系统说明这个问题，我们要引入变量提升的这一概念。

我们可以运行下如下代码看看结果。

```
console.log(foo)
console.log(bar)

var foo = function(){
    // Some code
};
function bar(){
    // Some code
};
```

输出为

```
undefined
function bar(){
    // Some code
};
```

为什么那？为什么 foo 打印出来是 undefined，而 bar打印出来却是函数？

JavaScript在执行时，会将变量提升。

所以上面代码JavaScript 引擎在实际执行时按这个顺序执行。

```
// foo bar的定义位置被提升
function bar(){
    // Some code
};
var foo;

console.log(foo)
console.log(bar)

foo = function(){
    // Some code
};
```

原代码的输出合理解释了。

问题18:下面代码输出什么?

```
var salary = "1000$";

(function () {
    console.log("Original salary was " + salary);

    var salary = "5000$";

    console.log("My New Salary " + salary);
})();
```

输出是

```
Original salary was undefined
My New Salary 5000$
```

这题同样考察的是变量提升。等价于以下代码

```
var salary = "1000$";

 (function () {
     var salary ;
     console.log("Original salary was " + salary);

     salary = "5000$";

     console.log("My New Salary " + salary);
 })();
```

问题19:什么是 instanceof 操作符？下面代码输出什么?

```
function foo(){
  return foo;
}

console.log(new foo() instanceof foo);
```

instanceof操作符用来判断是否当前对象是特定类的对象。

如

```
function Animal(){
    //或者不写return语句
    return this;
}
var dog = new Animal();
dog instanceof Animal // Output : true
```

但是，这里的foo定义为

```
function foo(){
  return foo;
}
```

所以

```
// here bar is pointer to function foo(){return foo}.
var bar = new foo();
```

所以 new foo() instanceof foo 返回 false

问题20: 如果我们使用JavaScript的"关联数组"，我们怎么计算"关联数组"的长度?

```
var counterArray = {
    A : 3,
    B : 4
};
counterArray["C"] = 1;
```

其实答案很简单，直接计算key的数量就可以了。

```
Object.keys(counterArray).length // Output 3
```