目录

[Introduction 2](#_Toc522142496)

[Program Structure 4](#_Toc522142497)

[Function 8](#_Toc522142498)

# Introduction

JS中的值：

JS中的值可以是数字，字母，文字，函数 等等。每一个值都有一个对应的名称。值被储存在某处，可以通过名称来调用。

1. 数字。
2. 正常的数字；
3. 有浮点的数字；
4. 如果数字太大，用e添加： 2.998e8
5. Arithmetic & Operator

数字的运算

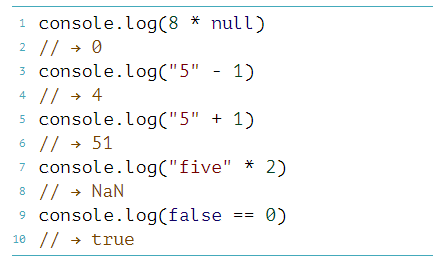
1. 特殊的数字，比如Infinity & -Infinity , NaN
2. 字符串
3. 字符串中若有\这个标记，说明后面有特殊意义，比如\n
4. 如果想使用\标记，那么就直接[\\ 代表一根](file:///\\代表一根)
5. “+”代表直接连接字符串
6. 重音符号还有一些其他的特殊用法：`half of 100 is ${100 / 2}`

🡺“half of 100 is 50”

1. 不是所有的运算符都是直接运算，比如 一元运算“typeof ”
2. 布尔值： 显示true/false。例外：NaN == NaN 🡺false.
3. 逻辑运算：二元运算 and, or, not. &&, ||, !

三元运算： true? 1:2;

1. 特殊值： null and undefined
2. 自动类型 转换：



逻辑语句的使用：|| 和&&

||的特点：

Console.log(null || “user”) 🡺 user （第一个值为假，就使用第二个值，这个可以用来给某个可能为empty的值设定默认值）

Console.log(“angela”|| “user”) 🡺 angela （第一个值为真，就使用第一个值）

&&的特点:

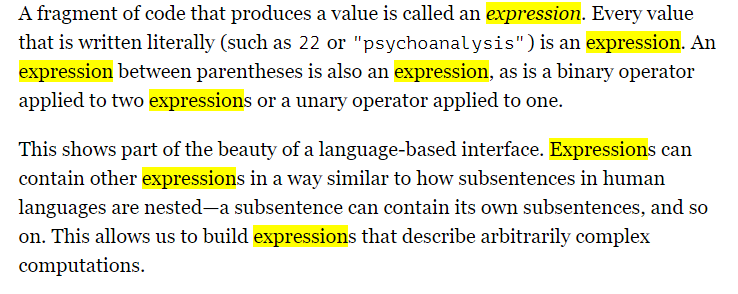
Console.log(null && “user”) 🡺null

Console.log(“angela” && “user”) 🡺user.

Programming

1. 养成良好的习惯，每一句话后面都要加分号；

Expression:



Binding:

现在在定义一个变量的时候，主要使用let. 跟之前var的用法一样

还有另外一个也可以定义变量： const (stands for constant) 意思就是一个不变的变量，不像let可以随时重新赋值，const 的变量，随时调用都是你设定的原来那个值。

变量名称命名注意：

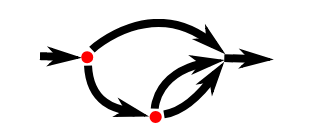
1. 数字不能放在前面；
2. 可以有特殊符号$和\_，其他的不可以。

Arguments: 参数

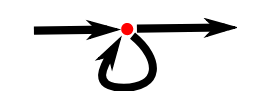
# Program Structure

流程控制：

If（建立很多分支）



While(可以设定一个循环)



Do while;

附录：

Syntax

Number.isNaN(value) ：判断要要检测值是否为NaN值： 返回布尔值

注意：

和全局函数 [isNaN()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/isNaN) 相比，该方法不会强制将参数转换成数字，只有在参数是真正的数字类型，且值为 NaN 的时候才会返回 true。

Number.isNaN(NaN); // true

Number.isNaN(Number.NaN); // true

Number.isNaN(0 / 0) // true

// 下面这几个如果使用全局的 isNaN() 时，会返回 true。

Number.isNaN("NaN"); // false，字符串 "NaN" 不会被隐式转换成数字 NaN。

Number.isNaN(undefined); // false

Number.isNaN({}); // false

Number.isNaN("blabla"); // false

// 下面的都返回 false

Number.isNaN(true);

Number.isNaN(null);

Number.isNaN(37);

Number.isNaN("37");

Number.isNaN("37.37");

Number.isNaN("");

Number.isNaN(" ");

Loop: 通过for循环来实现。

--

如果是一个派遣的逻辑，通常使用if..else，虽然swith case也同样可以实现：

原因： 使用switch case需要添加break; 一旦忘记添加可能会导致其他问题。

There is a construct called switch that is intended to express such a “dispatch” in a more direct way. Unfortunately, the syntax JavaScript uses for this (which it inherited from the C/Java line of programming languages) is somewhat awkward—a chain of if statements may look better.

If. Else 语句：

**let** x=prompt("Who is your favorite actor in Harry Potter");  
**if**(x == "harry") {  
 console.log("Oh, you like Harry? You must also brave too!"); //if语句不需要用break,break是用来结束loop。  
}  
**else if**(x == "hermione") {  
 console.log("Are you a girl or a boy? Boys always like her:)");  
}  
**else if**(x == "ron" ) {  
 console.log("Great! I also like Wisley family, they are very friendly.");  
}  
**else**{  
 console.log("Everyone in this book has their own meaning. Hope you enjoy!");  
}

switch case

**switch** (prompt("Who is your favorite actor in Harry Potter")){  
 **case** "harry": //答案是大小写敏感的。  
 console.log("Oh, you like Harry? You must also brave too!");  
 **break**;  
 **case** "hermione":  
 console.log("Are you a girl or a boy? Boys always like her:)");  
 **break**;  
 **case** "ron":  
 console.log("Great! I also like Wisley family, they are very friendly.");  
 **break**;  
 **default**:  
 console.log("Everyone in this book has their own meaning. Hope you enjoy!");  
 **break**;  
}

Binding 的命名方法：

1. 一般的命名原则：大写除第一个单词以外的后面每一个单词首字母；
2. 一些少量的函数，例如数字函数 number function。连第一个单词首字母一起大写。这样使得大家明白这个函数是一个构造器（constructor）

总结：

Program即为 statement；

Statement里可以有很多的statement和expression

Function 则可以用来封装一个一个的program

If an expression corresponds to a sentence fragment, a JS statement correspond to a full sentence.

1. Similar:

Const x=5; =🡺 expression is a value;

Const y=getAnswer(); 🡺 in most times, will be a value;

1. Difference:

Statement is not always a value. It’s an instruction/action, for example； condition ： if/else; loop: for/while🡺 control action, invoke action, but not value. 🡺 they are statements.

<https://www.youtube.com/watch?v=WVyCrI1cHi8>

注意：

* 在使用for循环的时候，变量需要首先赋值，否则会出现错误:

Variable has not been initialized.

eg. For(let x=0; x<3; x++){};

* When use “\n” in a loop, keep in mind that this is a newline character and will be counted as one in length.

# Function

A function🡺 value of a binding

Function can have parameters or have no parameters;

Usually, functions have statements inside ready to be executed.

##pay attention##

1. Functions can produce a value or not;

**const** makeNoise = **function** () {  
 console.log("Pling!");  
}

This function does not have a value;

**const** power = **function** (base, exponent) {  
 **let** result = 1;  
 **for**(**let** count = 0; count < exponent; count++ ){  
 result \*=base;  
 }  
 **return** result;  
}  
console.log(power(2, 10));

This function produces a value.

1. ***Return*** is a statement, and it will cause the function jump out of current function and gives a return value;
2. ***Return*** without an expression after it will cause the function returns undefined;

***Return \_\_\_\_\_\_;***

1. A function without a return statement, will also has similar result.

(No return)

Var vs. let vs. const

1. Variable hoisting

If a variable is not just declared in one function, it will walk up the tree to become a variable in a boarder scope.

For instance:

var i = 99999; //Here is a declared global variable

(function () { //call a function.

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

console.log(i);

}

})()

console.log('after loop', i);

🡺0,1,2,3,4,5,6,7,8,9

🡺10 in *for* loop, variable “i” has not been declared, so it’s scope walks up the tree and changed value of var i = 99999.

var i = 99999; //Here is a declared global variable

(function () { //call a function.

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

console.log(i);

}

var i;

})()

console.log('after loop', i);

🡺0,1,2,3,4,5,6,7,8,9

🡺99999 (Although i is not declared in for loop, but it has been declared in function.

var i = 99999; //Here is a declared global variable

(function () { //call a function.

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

console.log(i);

}

})()

console.log('after loop', i);

🡺0,1,2,3,4,5,6,7,8,9

🡺99999

Conculsion: So it can be very dangerous that if you forget to declare one variable in a function, since it may become a global scope variable.

1. One thing should know about “var”

In the past, only function can create new scope. Therefore, old-style bindings that created with var keywords are visible throughout whole function that they appear in, or throughout the global scope if they are not in a function.

eg

let x = 10;

if (true) {

let y = 20;

var z = 30;

console.log(x + y + z);

// → 60

}

// y is not visible here

console.log(x + z);

// → 40

1. This can be revised☺

The value of *let* can be reassigned.

The value of *const* cannot be reassigned.

eg

let x = 2;

x = 1;

console.log(x);

🡺1

Const x = {

x = 2;

}

x.x = 3;

console.log(x.x) = 3 =🡺 the value can be updated.

Const x = {

y = 3;

}

🡺error

Principle: Minimize Mutable State

1. Each scope can “look out” into the scope around it. So a variable is visible inside a function without it. Like “x” in 2. Point.
2. Exception: when multiple bindings have same name, it can only see the innermost one.

For instance:

Const halve = function(n){

Return n/2;

}

let n = 10; //this is a global scope variable.

Console.log(halve(100)) 🡺 50 //they have same binding names and will only see it’s own.

Console.log(n) 🡺 10 // will not change the value of “n” global scope.

**Nested Scope**

There is not only global and local bindings. Local bindings can produce multiple degrees of locality. Because blocks and functions can be created inside other blocks and functions:

e.g.

1. Function A inside Function B can see bindings from Function B.
2. Function B cannot see bindings from function A;
3. All function A and B can see global bindings.

Conclusion:

Each local scope can see all the local scopes that contain it, and all scopes can see the global scope. This approach to binding visibility is called lexical scoping.

**Functions as value**

A function value 🡺 is a part of the regular “top-to-bottom” flow of control.

When use keyword”let””const” it strictly follow this regular rule. In a block or local scope, if one binding is defined with “var” keyword, and used before it’s definition, it will not show as” is not defined”, it will show as a value “undefined”.

Var is an exception.

**Declaration notation**

Function Test(){}

This is not a part of regular “top-to-bottom” flow of control.

The preceding code works, even though the function is defined below the code that uses it. They are conceptually moved to the top of their scope and can be used by all the code in that scope.

**Arrow Function**

Const power = (base, exponent) = >{ //means, this input(parameters) produces this result(the body)

let result = 1;

for(let count = 0; count < exponent; count++){

result \*=base;

}

return result;

};

If there is only one parameter, can omit parentheses around it. If body is a single expression, can return directly.

Const square = x = > x\*x;

**CALL STACK**

LIKE A PILE OF BOOKS.

“FIRST IN, LAST OUT”

Every time a function is called, the current content goes to the top of the stack. When function returns, it removes from the top context of the stack and uses that context to continue execution.

Can use call stack in Chrome to check this flow.

**Optional arguments**

One feature of JS is that if you pass too many arguments, the extra ones are ignored. If too few, then the missing parameters get assigned the value undefined.

Upside: it can be used to call a function with different numbers of arguments.

Downside: if you pass wrong arguments, it’s really hard to find it out.

**Closure**

A very important feature of Javascript. (Need to practice more about this)