tong Object-Oriented JavaScript

# Chapter 1

1. JavaScript there are no classes;
   JavaScript has the notion of prototypes, which are also objects (we'll discuss them later in detail). In a classic OO language, you'd say something like "create me a new object called Bob which is of class Person". In a prototypal OO language, you'd say, "I'm going to take this object Person that I have lying around and reuse it as a prototype for a new object that I'll call Bob".

2. In JavaScript, all methods and properties are public, but we'll see that there are ways to protect the data inside an object and achieve privacy

3. In classical OOP, classes inherit from other classes, but in JavaScript, because there are no classes, objects inherit from other objects.

# Chapter 2 JavaScript Basic

Everything that is not a primitive data type is an object

1. Five primitive types:
   1. Number: float or int
   2. String
   3. Boolean
   4. Undefined
   5. Null
2. Infinity: a big/small number that JavaScript can’t represent
   1. Used in [a = 6 / 0 ]
   2. Infinity – Infinity = 0
   3. Every other arithmetic on infinity will give back infinity.
3. NaN
   1. Typeof NaN = number
   2. NaN is Returned whenever there’s a failed arithmetic operation.
   3. NaN cannot be used to directly check if a value is a valid number
      NaN === NaN // false
4. String conversion.
   1. var s=”1”; s++; ➔ doing arithmetic operation on number-like strings will automatically convert them to number.
   2. failed conversion -→ get NaN
   3. special char
      1. \\ : backslash
      2. \r : carriage return
      3. \u: Unicode representation
      4. \t: tab
5. Boolean
   1. two value: true or false

2. quoted Boolean value will be regarded as a string
3. the following thing will be viewed as false : 0, NaN, undefined, null, “”, false

6. Logical operators
    1. ! > && > ||, in operator precedence
    2. best practice: using parentheses instead of operator precedence

7. Lazy evaluation: If you have several logical operations one after the other, but the result becomes clear at some point before the end, the final operations will not be performed because they don't affect the end result
    1. If JavaScript encounter a non-logical operator, it will be return that.
       Console: True && “something”
       Result : “something”
    2. So this is very useful when initializing a variable but not sure if the customized value is given. You can give a default value behind || operator
       Console: var a = settings.aaaa || 10

8. comparison
    1. == : compare value, operand will be transferred into same type:
    2. ===: compare value **and** type
    3. notice that NaN is not equal to even itself

9. Undefined and null
    1. If you define a variable but did not provide a value, the following result will be received

```
Console: Var aaa;
         Typeof aaa;
result:  Undefined
```

2. The null value, on the other hand, is not assigned by JavaScript behind the scenes;

```
Console: var aaa = null;
          typeof aaa
Result:  object
```

3. There are some small differences between the two

```
Console: var a = 1 + undefined;   //Undefined will be converted to NaN
Result:  a = NaN                  //But null will be converted to 0
Console: var a = 1 + null;
Result:  a = 1
undefined == null
> true
```

10. Array
    1. declaration of an array(some as list in Python). Array can contain all type of value

```
Var a = []; a = [1,2,3];
a[3] = 4 ;          //a=[1,2,3,4]   dynamicly add more elements into array
a[6] = 4;           //a=[1,2,3,4,undefined*2, 4] gap will be filled using undefined
a[1] = “world”       //a=[1, “world”, 2,3,4,undefined * 2, 4]
delete a[1]         //a= [1, undefined, 2,3,4,undefined * 2, 4]
                    //Length doesn’t change. Only value becomes undefined
Var b=[[1,2,3], [1,2,3]] //nested array
```

Var c=”aaa”            //c[1] = “a”, string is also kind of an array

11. Code block
    1. Best practice tips
        1. Use end-of-line semicolons, as discussed previously in the chapter. For best readability, the individual expressions inside a block should be placed one per line and separated by semicolons.
        2. Indent any code placed within curly brackets.
        3. Use curly brackets. When a block consists of only one expression, the curly brackets are optional, but for readability and maintainability, you should get into the habit of always using them, even when they're optional.
    2. Check if a value exists

```
If (typeof variable !== “undefined”)
{
      //do something
}
```

4. Simplified if syntax: syntax sugar. Don’t abuse it.

```
Var result = (a === 1) ? “yes” : “no”
```

5. Switch: same as C.

   Best practice:
   1. Don’t forget to add break after each sentence.
   2. Indent the code that follows the case lines.
   3. Use the default case. This helps you make sure you always have a meaningful result after the switch statement, even if none of the cases matches the value being switched.
   4. Sometimes, you may want to omit the break intentionally, but that's rare. It's called a fall-through and should always be documented because it may look like an accidental omission.

5. Loops: same as C, regarding do…While loop
   1. For-in loops: used to iterate over the elements of an array or an object

```
Console: var a = [1,2,3,4,5,6];
              For (var i in a)
              {
                  //do something
              }
```

Notice: this is for informational purposes only, as for-in is mostly suitable for objects, and the regular for loop should be used for arrays

# Chapter 3 Function

1. Basic of function
    a) The return statement. A function always returns a value. If it doesn't return a value explicitly, it implicitly returns the value undefined.
    b) You can return an array for multiple values.
    c) JavaScript is not picky at all when it comes to accepting arguments. If you pass more than the function expects, the extra ones will be silently ignored.

d) argument: an array contains all parameters passed to it and create automatically

```
function args() {
return arguments;
}
> args();
[]
> args( 1, 2, 3, 4, true, 'ninja');
[1, 2, 3, 4, true, "ninja"]
```

e) The following function is used to add arbitrary numbers

```
function sumOnSteroids() {
    var i,
    res = 0,
    number_of_params = arguments.length;
    for (i = 0; i < number_of_params; i++) {
        res += arguments[i];
    }
    return res;
}
```

f) Built-in functions

1. parseInt(string, [base]):

   Description: takes any type of input (most often a string) and tries to make an integer out of it. If it fails, it returns NaN

   Notice: **always specify the base, otherwise you may fail to get the desired result**

   Example:
   ```
   parseInt("FF", "16")
   parseInt("0388")      //recognized as octal number
   parseInt("0x88")      //recognized as hex number
   ```

2. parseFloat(string):

   Description: looks for decimals when trying to figure out a number from your input.

   Notice: 1. Function will stop and return when it encounters non-number chars
   2. Function recognize exponential representation

   Example:
   ```
   parseFloat('1e10');       //   10000000000
   parseFloat('abc123');     //   NaN
   ```

3. isNaN(string/number):

   Description: check if an input value is a valid number that can safely be used in arithmetic operations.

   Notice: **if string is a number, false will be returned.**

4. isFinite(string/number):
   Description: checks whether the input is a number that is neither Infinity nor NaN.

   Notice: **if input is a finite number, true will be returned.**
5. encodeURl(string) & encodeURLComponent(string)
   Description: If you want to "escape" those characters, you can use the functions.

   Example:
   var url = 'http://www.packtpub.com/script.php?q=this and that';
   encodeURI(url);
   ➢ http://www.packtpub.com/scr%20ipt.php?q=this%20and%20that
   encodeURIComponent(url);
   ➢ "http%3A%2F%2Fwww.packtpub.com%2Fscr%20ipt.php%3Fq%3Dthis%20and%

   Notice: **encodeURLComponent will encode all chars including http://**

6. Eval()
   Description: takes a string input and executes it as a JavaScript code:

   Notice: **You should not use eval() if there are other options. Eval() is neither security nor efficient**

g) Variable scope
   1. Function scope: This means that if a variable is defined inside a function, it's not visible outside of the function. But all variables defined in a function are seen all through the function.
   2. Global variable: all variables defined outside any functions
   3. If you don’t use **var** to declare a variable, you will always get a **global variable.**
   4. Best practice
      1. Minimize the number of global variables in order to avoid naming collisions.
      2. Always declare your variables with the var statement.
      3. Consider a "single var" pattern. Define all variables needed in your function at the very top of the function so you have a single place to look for variables and hopefully prevent accidental global variables.

h) Variable hoisting
   See the following example:

```
var a = 123;
function f() {              //When executing, var a; will be hoisted to the top of
    alert(a);               //the functi
    var a = 1;
    alert(a);
}
f();
```

   The first alert will give “undefined” since local variable will cover and overwhelm all global variable, however, which still existed in the local space due to **the special behavior called hoisting**

1. When your JavaScript program execution enters a new function, all the variables declared anywhere in the function are moved (or elevated, or hoisted) to the top of the function

i) Functions are data

1. **Function literal notation**

   Example:
```
var f = function () { return 1;}
typeof f
> function
var add = sum;              // functions can be assigned as variables
typeof add;
> function
add(1, 2);
> 3
```

   1. You can assign a name but that's rare and may cause an error in IE
   2. Difference between a named function expression and a function declaration
      **Look at the context in which they are used.** Function declarations may only appear in program code

2. **Anonymous functions**
   1. Two ways of using anonymous functions
      a) You can pass an anonymous function as a parameter to another function.
      The receiving function can do something useful with the function that you pass.
      b) You can define an anonymous function and execute it right away.

3. **Callback function**

   When you pass a function, A, to another function, B, and then B executes A, it's often said that A is a **callback** function.

   Example:
```
function invokeAdd(a, b) { return a() + b();}
invokeAdd(
    function () { return 1; },
    function () { return 2; }
  );
> 3
myarr = multiplyByTwo(1, 2, 3, addOne);
function multiplyByTwo(a, b, c, callback) {
    var i, ar = [];
    for (i = 0; i < 3; i++) {
        ar[i] = callback(arguments[i] * 2);
    }
    return ar;
}
multiplyByTwo(1, 2, 3, function (a) {
    return a + 1;
    });                              // Best practice: use anonymous function
```

1. They let you pass functions without the need to name them
2. You can delegate the responsibility of calling a function to another function
3. They can help with performance

4. **Immediate functions**
   1. When you want to have some work done without creating extra global variables. A drawback, of course, is that you cannot execute the same function twice. This makes immediate functions best suited for one-off or initialization tasks.
   2. Example:

```
(function () {
    // ...
})();
[or]
var result = (function () {
    // something complex with
    // temporary local variables...
    // ...
    // return something;
}());
```

   You don't know if result is a function or the return value of the immediate function.

5. **Inner (private) functions**
   1. Best practice:
      1. You keep the global namespace clean (less likely to cause naming collisions)
      2. Privacy—you expose only the functions you decide to the "outside world", keeping to yourself functionality that is not meant to be consumed by the rest of the application
   2. Example:

```
var outer = function (param) {
    var inner = function (theinput) {
        return theinput * 2;
    };
    return 'The result is ' + inner(param);
};
```

6. **Functions that return functions**
   1. You can return a function instead of a value.
   2. Example:

```
function a() {
    alert('A!');
    return function () {
        alert('B!');
    };
};
```

```
Var new = a();                    // say A!
New();                            // say B!
a()();                            // Directly invoke returned function
```

7. **Function rewrite**
   1. Inside one function, it can rewrite itself.
   2. Example:

```
function a() {
    alert('A!');
    a = function () {
        alert('B!');
    };
}
```

   3. Advanced example: an anonymous function

      If you know that the browser features won't change between function calls, you can have a function determine the best way to do the work in the current browser, then redefine itself so that the "browser capability detection" is done only once.

      Example:

```
var a = (function () {
        function someSetup() {
            var setup = 'done';
        }
        function actualWork() {
            alert('Worky-worky');
        }
        someSetup();
        return actualWork;
    }());
```

8. **Closure: a closure is created when a function keeps a link to its parent scope even after the parent has returned. Closure also can retain variables from return function.**
   1. **Scope chain**
      1. If you define a function inner() nested inside outer(), inner() will have access to variables in its own scope, plus the scope of its "parents".
      2. Example:

```
var global = 1;
function outer() {
    var outer_local = 2;
    function inner() {
        var inner_local = 3;
        return inner_local + outer_local + global;
    };
    return inner();
};
```

   2. **Breaking the chain with a closure**

By making itself global (omitting var) or by having F deliver (or return) it to the global space. Let's see how this is done in practice.

3. Closure example #1

```
var a = "global variable";
var F = function () {
        var b = "local variable";
        var N = function () {
                var c = "inner local";
                return b;
        };
        return N;
 };
 var inner = F();
inner();
```

4. Closure example #2: use external variable

A new function, N(),is defined inside F() and assigned to the global inner. During definition time, N() was inside F(), so it had access to the F() function's scope. inner() will keep its access to the F() function's scope, even though it's part of the global space:

```
var inner; // placeholder
var F = function () {
       var b = "local variable";
       var N = function () {
               return b;
       };
       inner = N;
};
F();
Inner()
```

5. Closure example #3:

   1. **Every function maintains access to the global scope, which is never destroyed.**
   2. **Function parameters behave like local variables to this function, but they are implicitly created (you don't need to use var for them).**
   3. **The functions don't remember values, they only keep a link (reference) to the environment where they were created.**

```
function F(param) {
        var N = function () {
                return param;
        };
        param++;
        return N;
 };
 Inner = F(123);    // Inner is external variable but can access to the parent space
```

6. How to remember the value:
   **The key is to use the middle function to "localize" the value of i at every iteration:**

```
function F() {
       var arr = [], i;
       for (i = 0; i < 3; i++) {
              arr[i] = (function (x) {
                     return function () {return x;};        //anonymous immediate function
              }(i));                                        //localize the value
       }
       return arr;
}
```

Other variant::

```
function F() {
       var arr = [], i;
       for (i = 0; i < 3; i++) {
              arr[i] = (function (x) {
                     return x;              // will return [1,2,3] directly since this
              }(i));                        // is not a function but immediate value.
       }
       return arr;
}
```

9. **Getter/Setter: use closure to protect private variable**

```
var getValue, setValue;
(function () {
        var secret = 0;
       getValue = function () {
              return secret;
       };
       setValue = function (v) {
              if (typeof v === "number") {
                     secret = v;
               }
       };
 }());
```

10. **Iterator: wrap the complicated "who's next" logic into an easy-to-use next() function**

```
function setup(x) {
       var i = 0;
       return function () {
              return x[i++];              //localize the variable && use an implicit setter
       };
}
var next = setup(['a', 'b', 'c']);
```

```
next();
> “a”
```

# Chapter 4 object

1. Object declaration
   Example:
   ```
   var hero = {occupation: 1};
   var hero = {"occupation": 1};
   var hero = {'occupation': 1};
   ```

   Best practice: **always quotes the key**.

2. Elements, properties, methods, and members
   ```
   var dog = {
       name: 'Benji',                  // define a property/member
       talk: function () {             // define a method
           alert('Woof, woof!');
       }
   };
   ```
3. Hashes and associative arrays: this is so-called “object” in JavaScript terminology.
4. Accessing an object's properties: [] / .
5. Calling an object's methods
   ```
   var hero = {
       breed: 'Turtle',
       occupation: 'Ninja',
       say: function () {
           return 'I am ' + hero.occupation;
           }
   };
   hero.say();
   ```
   **hero['say']();** //You can treat method as a property
   **hero[name] // You can access the propery using variable**
   delete hero.say //You can delete a property/method of a object
   best practice: **Always use dot notation to access the method and property.**

6. Altering properties/methods
   1. Blank object
      ```
      Var a = {};
      Typeof a.b
      > Undefined
      a.b = 1               // Add new property
      a.c = function() {    // Add new method
          return “123”;
      ```

```
}
```

2. Note: **A few properties of some built-in objects are not changeable (for example, Math.PI, as you'll see later).**

7. “this” pointer

```
var hero = {
    name: 'Rafaelo',
    sayName: function () {
    return this.name;                    // Same with Java
    }
};
```

8. Constructor functions
   1. Example:

```
function Hero() {                        // Like a class in Java
    this.occupation = 'Ninja';           // This is a template function
}                                        // only this.xxx is a property. Otherwise only a var
var hero = new Hero();                   // An object
hero.occupation;                         // If new is omitted, undefined will be returned.
"Ninja"
```

   2. A benefit of using constructor functions is that they **accept parameters**

```
function Hero(name) {
    this.name = name;
    this.occupation = 'Ninja';           // This is just like an parameter constructor
}                                        // in Java
```

   **3.** Best practice: **By convention, you should capitalize the first letter of your constructor functions so that you have a visual clue that this is not intended to be called as a regular function.**

9. Global object: the host environment provides a global object and all global variables are accessible as properties of the global object.
   1. If your host environment is the web browser, the global object is called **window.**
   2. Access the global object

```
var a = 1;
window.a;              // Give “1”
this.a;                // Give “1”, referring to the same variable
```

   3. The built-in functions, e.g. parseInt(), are also global function. Namely, they can be accessed using **window** or **this**

10. The constructor property
    1. When an object is created, a special property is assigned to it behind the scenes—the constructor property. It contains a reference to the constructor function used to create this object.

```
h2.constructor;                          // a method referring to the constructor function
```

```
> function Hero(name) {
      this.name = name;
  }
  var h3 = new h2.constructor('Rafaello');   // Reuse it.
```

2. If an object was created using the object literal notation, its constructor is the built-in Object() constructor function (there is more about this later in this chapter):

```
var o = {};
o.constructor;
function Object() { [native code] }
```

11. The instanceof operator: test if an object was created with a specific constructor function
    1. Example:

```
var h = new Hero();
h instanceof Hero;          // Here, Hero is a constructor function.
> true                      // Like a class in Java
```

12. Functions that return objects: this function is called **factory**
    1. Example:

```
function factory(name) {
      return {
            name: name                 // An object will be returned
              };
}
```

2. An peculiar example:

```
function C2() {              // Some logic as function hoisting
      this.a = 1;            // a statement [var this = {}; is hoisted to the top of the function]
      return {b: 2};
}
var c2 = new C2();
c2.a                         // This is possible only if the return value is an object.
> undefined                  // if you try to return anything that is not an object, the constructor
C2.b                         // will proceed with its usual behavior and return this
> 2
function F() {
      function C() {
            return this;
      }
      return C();
}                            // o is the global object. This will point to the owner of the function
var o = new F();             // If C is a method, this points to F.
```

13. Passing objects: objects are passed using reference by default

Example:

```
var original = {howmany: 1};
var mycopy = original;
```

mycopy.howmany = 100;        //original.howmany is also 100 now.

14. Comparing objects: get true only if you compare two references to the same object

15. Console function: console.log() is convenient when you want to quickly test something.

16. Built-in objects
    1. Data wrapper object: These are Object, Array, Function, Boolean, Number, and String. These objects correspond to the different data types in JavaScript.
    2. Utility object: These are Math, Date, and RegExp, and can come in handy.
    3. Error objects: generic & specific error object.

17. Object: the parent of all JavaScript objects, which means that every object you create inherits from it
    1. Example:

```
Var o = {};
Var o = new Object();        // They are equivalent
o[0] = 1;                    //o = {0 : 1}
```

    2. Method that a blank object have.
        1. o.constructor: a reference to the constructor function
        2. o.toString():returns a string representation of the object
        3. o.valueOf():returns a single-value representation of the object; often this is the object itself
        4. notice: **When you call alert, JavaScript will call toString() of the parameter automatically.**
        5. Example:

```
o.valueOf() === o;
> true
```

18. Array:
    1. Example:

```
var a = new Array();                  //This is equivalent to the array literal notation:
var a = new Array(1, 2, 3, 'four');   //Constructor can receive parameters
var a2 = new Array(5);                //This will be regarded as the length of the array
var a3 = [1,,2]                          //The second element will be undefined
a.toString();                         //会调用数组元素的 toString
> "1,2,3,four"
```

    2. Special property

```
a.length;                             //Returns the number of elements in the array.
//这个值不是一个只读的，可以修改来改变数组大小
> 4
a.prop = 2                            // length property ignore non-numeric properties.
a.length = 5;                         // You can set the size of the array using length
```

    3. Special method

a.reverse();                    // Reverse the array

栈方法：

a.push(1);                    // add element into the tail of the array

//This equals to a[a.length] = 1    , add 1 element

a.pop();                    // remove the last element

队列方法：

a.push() + a.shift()                    //队尾入队，队头出队

a.pop()+ a.unshift()                    //队尾入队，队头出队

a.sort();                    //sort and return the array

a.sort(comparator)                    //比较器接收两个参数，如果第一个值比第二个大，//返回 1，否则返回-1，相等返回 0.

a.join(“ ”)                    // returns a string containing the values of all the

> “1 2 3 “four””                    // elements in the array glued together

//不传参的时候用逗号

a.slice(1,2)                    // You can regard it as Python slice [:]

//可以使用负数，表示倒数

a.splice(1,2, 3,4,6)                    // It removes a slice, returns 【it】, and

//optionally fills the gap with new elements.

//splice() uses at least 2 parameter:

1st    the start index

2nd    the number of the elements removed

Optional    new elements to fill the gap

//主要用法：1. 删除，删除任意长度的数据，第二个参数传正数

2. 附加，附加任意长度数据，第二个参数传 0，之后正常传

3. 替换，综合 1,2

a.concat()                    //concat elements to the rear of the array.

a.indexOf()和 a.lastIndexOf()寻找参数并返回参数在数组里的位置

迭代方法：

a)

19. Function: also objects
    1. Create a function using Function() constructor:

    var sum = new Function('a', 'b', 'return a + b;');    // Same drawback as eval(), avoid it

    2. Parameters can be passed as a single commade limited list.
    3. Best practice: **Do not use the Function() constructor**. As with eval()  and setTimeout() (discussed later in the book), always try to stay away from passing JavaScript code as a string.

20. Properties of function objects
    1. Constructor
    2. function myfunc(a, b, c) {

```
        return true;
}
Myfunc.length
> 3                    // contains the number of formal parameters the function expects.
```

3. Prototype: One of the most widely used properties of function objects
   1. The prototype property of a function object points to another object
   2. **Its benefits shine only when you use this function as a constructor**
   3. All objects created with this function keep a reference to the prototype property and can use its properties as their own
   4. Use of prototype

```
var ninja = {
     name: 'Ninja',
     say: function () {
           return 'I am a ' + this.name;
     }
};
function F() {};
F.prototype = ninja;                  // Now F turns to be a constructor function
var baby_ninja = new F();             // Will return a copy of ninja
```

4. Methods of function objects
   1. toString(): returns the source code of the function

```
function myfunc(a, b, c) {
      return a + b + c;
}
myfunc.toString();
"function myfunc(a, b, c) {
return a + b + c;
}"
```

   2. Call() and apply():These methods also allow your objects to "borrow" methods from other objects and invoke them as their own.

```
var some_obj = {
      name: 'Ninja',
      say: function (who) {
        return 'Haya ' + who + ', I am a ' + this.name;
      }
};
var my_obj = {name: 'Scripting guru'};
some_obj.say.call(my_obj, 'Dude');
> "Haya Dude, I am a Scripting guru"
```

      1. Analysis: You invoked the call() method of the say() function object passing two parameters: the object my_obj and the string 'Dude'. The result is that **when say() is invoked, the references to the this value that it contains point to my_obj**. This way, this.name doesn't return Ninja, but Scripting guru instead.
      2. The method apply() works the same way as call(), but with the difference that all parameters you want to pass to the method of the other object are passed as an array.

   3. The arguments object revisited
      1. arguments looks like an array, but it is actually an array-like object.

a) contains indexed elements and a length property
b) arguments doesn't provide any of the array methods, such as sort() or slice().
c) you can convert arguments to an array and benefit from all the array goodies.

21. Inferring object types
    1. Example:

```
Object.prototype.toString.call({});
> "[object Object]"
Object.prototype.toString.call([]);
> "[object Array]"
(function () {
        return toStr.call(arguments);
}());
> "[object Arguments]"
toStr.call(document.body);
> "[object HTMLBodyElement]"
```

22. Boolean

```
var b = new Boolean();            // this creates a new object, b, and not a primitive Boolean value.
typeof b;
> "object"
typeof b.valueOf();
> "boolean"
b.valueOf();
> false
```

    1. Overall, objects created with the Boolean() constructor are not too useful, as they don't provide any methods or properties other than the inherited ones.
    2. Best practice: **sticky to the original primitive type.**

23. Number:
    1. Number() function includes:
        a) A constructor function (with new) to create objects.
        b) A normal function in order to try to convert any value to a number.
    2. The Number() function has constant built-in properties that you cannot modify:

```
Number.MAX_VALUE;
> 1.7976931348623157e+308
Number.MIN_VALUE;
> 5e-324
Number.POSITIVE_INFINITY;
> Infinity
Number.NEGATIVE_INFINITY;
> -Infinity
Number.NaN;
> NaN
```

3. The Number function() provides three built-in method: toFixed(), toPrecision(), and toExponential()
   ```
   (12345).toExponential();
   > "1.2345e+4"
   ```
4. toString() method accepts an optional radix parameter (10 being the default):
   ```
   var n = new Number(255);
   n.toString();
   > "255"
   n.toString(16);
   > "ff"
   ```

24. String
    1. Create a string object
       ```
       var obj = new String('world');
       typeof obj;
       > "object"
       ```
    2. Primitive strings are not objects, so they don't have any methods or properties. But, JavaScript also offers you the syntax to treat primitive strings as objects
       ```
       "potato".length;
       > 6
       ```
    3. Last difference between string and string object:
       ```
       Boolean("");
       > false
       Boolean(new String(""));
       > true                              // all objects will be regarded as true
       ```
    4. If you use the String() function without new, it converts the parameter to a primitive:
       ```
       String(1);
       > "1"
       ```
       If you pass an object to String(), this object's toString() method will be called first
    5. A few methods of string objects
       1. toUpperCase() and toLowerCase() transforms the capitalization of the string:
          ```
          s.toUpperCase();
          >"COUCH POTATO"
          s.toLowerCase();
          > "couch potato"
          ```
       2. charAt() tells you the character found at the position you specify, which is the same as using square brackets
          ```
          s.charAt(0);
          > "C"
          ```
       3. indexOf() allows you to search within a string. If there is a match, the method returns the position at which the first match is found.
          ```
          s.indexOf('o');         // You must use[ if (s.indexOf('Couch') !== -1) {...} ] to check if
          > 1                     // there is a match
          s.indexOf('o', 2);      // You can optionally specify where (at what position) to start the
          ```

```
> 7                          // search.
s.indexOf('Couch');    // You can also search for strings, not only characters, and the
> -1                         // search is case sensitive:
s.lastIndexOf('o');      // lastIndexOf() starts the search from the end of the string
> 2
s.toLowerCase().indexOf('couch');    // a case-insensitive search
```

4. 
```
s.slice(1, 5);              // The difference between slice and substring is that
> "ouch"                    // substring() treats -1 as zeros, while slice() adds them to the
s.substring(1, 5);        //length of the string.
> "ouch"
s.slice(1, -1);
> "ouch potat"
s.split(" ");                // split() method creates an array from the string using another
> ["Couch", "potato"] // string that you pass as a separator
s.split(' ').join(' ');       // join(), which creates a string from an array:
> "Couch potato"
s.concat("es");            // concat() glues strings together, the way the + operator does for
> "Couch potatoes"    // primitive strings:
s.valueOf();                //while some of the preceding methods discussed return new
"Couch potato"          // primitivestrings, none of them modify the source string
```

25. Math object
    1. It's not a function, and therefore cannot be used with new to create objects.
    2. Math is a built-in global object that provides a number of methods and properties for mathematical operations. These method and properties can’t be modified
    3. Useful property and methods
        1. The constant π:
           ```
           Math.PI;
           > 3.141592653589793
           ```
        2. Square root of 2:
           ```
           Math.SQRT2;
           > 1.4142135623730951
           ```
        3. Euler's constant:
           ```
           Math.E;
           > 2.718281828459045
           ```
        4. Natural logarithm of 2:
           ```
           Math.LN2;
           > 0.6931471805599453
           ```
        5. Natural logarithm of 10:
           ```
           Math.LN10;
           > 2.302585092994046
           ```
        6. 
           ```
           Math.random();
           > 0.3649461670235814      // The random() function returns a number between 0 and 1
           ```
        7. floor() to round down

ceil() to round up
round() to round to the nearest
Example:
Math.round(Math.random()); //Will return 0 or 1

8. you can raise to a power using pow(), find the square root using sqrt(), and perform all the trigonometric operations—sin(), cos(), atan(), and so on.
   Example:
   Math.pow(2, 8);
   > 256
   Math.sqrt(9);
   > 3

26. Date object
    1. Date() is a constructor function that creates date objects. You can create a new object by passing:
       a) Nothing (defaults to today's date)
       b) A date-like string
       c) Separate values for day, month, time, and so on
       d) A timestamp

       Example:

       new Date();       //The current date
       >    Wed Feb 27 2013 23:49:28 GMT-0800 (PST)
       new Date('2015 11 12');    // not really a reliable way of defining a precise date
       > Thu Nov 12 2015 00:00:00 GMT-0800 (PST)
       new Date(2015, 0, 1, 17, 05, 03, 120);
       > Tue Jan 01 2015 17:05:03 GMT-0800 (PST)
       //suggested way, parameters are
       //Year
       **//Month: 0 (January) to 11 (December)**
       //Day: 1 to 31
       //Hour: 0 to 23
       //Minutes: 0 to 59
       //Seconds: 0 to 59
       //Milliseconds: 0 to 999
       new Date(2016, 1, 30);   //this is February. If you pass a greater than allowed value, your
       > Tue Mar 01 2016 00:00:00 GMT-0800 (PST)    // date "overflows" forward
       Date()      //If you call Date() without new, you get a string representing the current date
       > Wed Feb 27 2013 23:51:46 GMT-0800 (PST)
    2. Methods to work with date objects
       1. Most of the methods can be divided into set*() and get*() methods, for example, getMonth(), setMonth(), getHours(), setHours(), and so on.
       2. Static methods
          Date.parse('Jan 11, 2018'); // Date.parse() takes a string and returns a timestamp:
          > 1515657600000

```
Date.UTC(2018, 0, 11);     // Date.UTC() takes all the parameters for year,
> 1515628800000            //month, day and produces a timestamp in Universal Time:
Date.now();                // provides a more convenient way to get the timestamp
> 1362038353044
Date.now() === new Date().getTime();
> true
```

3. The internal representation of the date being an integer timestamp and all other methods being "sugar" on top of it.

27. Regular expression
    1. A regular expression consists of:
       a) A pattern you use to match text
       b) Zero or more modifiers (also called flags) that provide more instructions on how the pattern should be used
    2. Create an regex
       Example:

```
var re = new RegExp("j.*t"); // RegExp() constructor allows you to create regula
var re = /j.*t/;             // expression objects or regexp literal notation:
```

    3. Properties of RegExp objects
       1. Global: If this property is false, which is the default, the search stops when the first match is found. Set this to true if you want all matches.
       2. ignoreCase: When the match is case insensitive, the defaults to false (meaning the default is a case sensitive match).
       3. multiline: Search matches that may span over more than one line default to false.
       4. lastIndex: The position at which to start the search; this defaults to 0.
       5. source: Contains the regexp pattern.

       Example:

```
var re = new RegExp('j.*t', 'gmi');    // Once set, the modifier cannot be changed:
re.global;
> true
var re = /j.*t/ig; // To set any modifiers using the regex literal, you add them after the c
                   //losing slash:
```

    4. Methods of RegExp objects
       1. test() and exec().

```
/j.*t/i.test("Javascript");        // A case insensitive test. test() returns a Boolean
> true                             // (true when there's a match, false otherwise),
/j.*t/i.exec("Javascript")[0];     // the result is an array
> "Javascript"                     // exec() returns an array of matched strings
```

       2. String methods that accept regular expressions as arguments
          a) Search() and match()

```
s.match(/a/g);        // a global search
> ["a", "a"]          // match() returns an array of matches
s.search(/j.*a/i);    //The search() method gives you the position of the
> 5                   // matching string
```

b) replace(): allows you to substitute matched text with another string

1. normal use

```
s.replace(/[A-Z]/g, '');       // allows you to replace the matched text with some
> "elloavacriptorld"           // other string
s.replace(/([A-Z])/g, "_$1");  //use grouping and trap parentheses
> "_Hello_Java_Script_World"
```

2. Replace callbacks

Gives you the ability to implement any special logic you may need before specifying the replacements

```
s.replace(/[A-Z]/g, replaceCallback);
> "_hello_java_script_world"
```

The callback function receives a number of parameters (the previous example ignores all but the first one):

- The first parameter is the match
- The last is the string being searched
- The one before last is the position of the match
- The rest of the parameters contain any strings matched by any groups in

```
var callback = function () {
        glob = arguments;
        return arguments[1] + ' at ' +arguments[2] + ' dot ' +arguments[3];
};
"stoyan@phpied.com".replace(re, callback);
glob;
> ["stoyan@phpied.com", "stoyan", "phpied", "com", 0, "stoyan@phpied.com"]
```

c) split()

```
var csv = 'one, two,three ,four';
csv.split(/\s*,\s*/);                    //Noted that there are spaces between delimiter
> ["one", "two", "three", "four"]        //Use regex to remove the turbulence
```

d) Passing a string when a RegExp is expected

split(), match(), search(), and replace() can also take strings as opposed to regular expressions.

```
"pool".replace('o', '*');      //Noted that when an string is passed, the global
> "p*ol"                       //search is disabled by default
"pool".replace(/o/g, '*');     // This is the only way for global replacement
> "p**l"
```

28. Error object

a) Same as Java

**b)** Error objects are created by using one of these built-in constructors: **EvalError, RangeError, ReferenceError, SyntaxError, TypeError, and URIError**

c) e.name contains the name of the constructor that was used to create the error object. In different platforms, the name can also be different.

d) Example

```
throw new Error('Division by zero!');
throw {                                 // this gives you cross-browser control over the name
        name: "MyError",
        message: "OMG! Something terrible has happened"
}
```