# BASICSOFJAVASCRIPT

## Based On ES5


### Abstract

No prior knowledge of JavaScript is required, But assumed that one has Basic Programing knowledge(Like C,C++,Java)



Haramohan Sahu

Hara.sahu@gmail.com

# Contents

## JavaScript Data Types

JavaScript has six primitive data types:

1. null
2. undefined
3. Boolean
4. number
5. string
6. symbol – available only from ES6
7. and one complex data **type called object**.

In JS, the same variable can hold values of different types at any time.

```
let foo = 120; // foo is a number
foo = false;   // foo is now a boolean
foo = "foo";   // foo is now a string
```

To get the current type of the value of a variable, you use the typeof operator.

**The undefined type**

The undefined type is a primitive type that has one special value undefined. By default, when a variable is declared but not initialized, it is assigned the value of undefined.

```
let foo;
console.log(foo);        // undefined
console.log(typeof foo); // undefined
```

In this example, foo is a variable. Since foo is not initialized, it is assigned the value of undefined. The type of foo is also undefined.

It is important to note that the typeof operator also returns undefined when you call it on a variable that has not been declared:

```
console.log(typeof bar); // undefined
```

**The null type**

The null type is the second primitive data type that has only one value: null. Javascript defines that null is an empty object pointer.

**See the following example:**

```
let obj = null;
console.log (typeof obj); // object
```

JavaScript defines that null is equal to undefined as shown in the following statement.

```
console.log (null == undefined); // true
```

**The number type**

JavaScript uses the IEEE-754 format to represent both integer and floating-point numbers.

**Integer numbers**

The following statement declares a variable that holds an integer.

```
let num = 100;
```

If you want to represent the octal (base 8) literals, you put the first digit as zero (0) followed by octal digit numbers (0 to 7) as follows:

```
let oct = 060; // octal for 48
```

If the literal of an octal number is out of the range, JavaScript treats it as a decimal as shown in the following example.

```
let d = 090; // intepreted as 90
```

To avoid the confusion, ES6 allows you to specify an octal literal by using the prefix 0o followed by a sequence of octal digits from 0 through 7:

```
let v = 0o45;
console.log(v); // 37
```

To create hexadecimal (base 16) literal, you must use 0x (in lowercase) as the first two characters followed by any number of hexadecimal digits (0 to 9, and A to F).

```
let h = 0xf; // same as 0xF hexadecimal for 15
```

Floating-point numbers

To represent a floating-point number, you include a decimal point followed by at least one number. See the following example:

```
let f1 = 12.5;
let f2 = .3;   // same as 0.3, also valid but not recommended
```

JavaScript converts a floating-point number into an integer number if the number appears to be the whole number. The reason is that Javascript always wants to use less the memory since a floating-point value uses twice as much memory as an integer value.

```
let f3 = 200.00; // interpreted as integer 200
```

JavaScript allows you to use the e-notation to represent very large or small numbers as in the following example.

```
let f4 = 2.17e6; // ~ 2170000
```

JavaScript provides the minimum and maximum values of a number that you can access using Number.MIN_VALUE and Number.MAX_VALUE. In addition, JavaScript uses Infinity and -Infinity to represent the finite numbers, both positive and negative.

See the following example:

```
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
console.log(Number.MIN_VALUE); // 5e-324
console.log(Number.MAX_VALUE + Number.MAX_VALUE); // Infinity
console.log(-Number.MAX_VALUE - Number.MAX_VALUE); // -Infinity
```

**NaN**

JavaScript has a special numeric value called NaN, which stands for Not a Number. In fact, it means an invalid number.

For example, the division of a string by a number returns NaN as in the following example.

```
console.log('a'/2); // NaN;
```

The NaN has two special characteristics:

Any operation with NaN returns NaN.
The NaN does not equal any value, including itself.
Here are some examples:

```
console.log(NaN/2); // NaN
console.log(NaN == NaN); // false
```

**The string type**

In JavaScript, a string is a sequence of zero or more characters. A literal string begins and ends with either single quote(') or double quotes (").  A string that starts with a double quote must end with a double quote and a string that begins with a single quote must end with a single quote.

Here are some examples:

```
let greeting = 'Hi';
let foo = "It's a valid string";
let bar = 'I'm also a string';
```

JavaScript strings are immutable. It means that you cannot modify a string once it is created. However, you can create a new string based on an operation on the original string, like this:

```
let foo = 'JavaScript';
foo = foo + ' String';
```

In this example:

First, declare the foo variable and initialize it to a string of 'JavaScript'.
Second, use the + operator to combine 'JavaScript' with ' String' to make its value as 'Javascript String'.
Behind the scene, JavaScript engine creates a new string that holds the new string 'JavaScript String' and destroys two other original strings 'JavaScript' and ' String'.

**When adding a number and a string, JavaScript will treat the number as a string.**

**The boolean type**

The boolean type has two values: true and false, in lowercase. The following example declares two variables that hold boolean values.

```
let inProgress = true;
let done = false;
console.log(typeof done); // boolean
```

JavaScript allows values of other types to be converted into boolean values of true or false.

To convert a value of another data type into a boolean value, you use the Boolean function. The following table shows the conversion rules:

| Type | true | false |
|---|---|---|
| string | non-empty string | empty string |
| number | non-zero number and Infinity | 0, NaN |
| object | non-null object | null |
| undefined | | undefined |

See the following demonstration.

```
console.log(Boolean('Hi'));// true
console.log(Boolean(''));  // false

console.log(Boolean(20));  // true
console.log(Boolean(Infinity));  // true
console.log(Boolean(0));  // false
```

```
console.log(Boolean({foo: 100}));  // true on non-empty object
console.log(Boolean(null));// false
```

**The symbol type**

JavaScript added a primitive type in ES6: the symbol. Different from other primitive types, the symbol type does not have a literal form.

To create a symbol, you call the Symbol function as follows:

```
let s1 = Symbol();
```

Note that Symbol is a function, not an object constructor, therefore, you cannot use the new operator. If you do so, you will get a TypeError.

The Symbol function creates a new unique value every time you call it.

```
console.log(Symbol() == Symbol()); // false
```

You can pass a descriptive string to the Symbol function for the logging and debugging purposes.

```
let s2 = Symbol('event.save');
```

When you call the toString() method on the symbol variable, it returns more descriptive name as shown below:

```
console.log(s2.toString()); // Symbol(event.save)
```

You can use symbols for many purposes. One of them is to create a string-like constant that can't clash with any other value. The following example creates a symbol that represents the click event.

```
const click = Symbol('click');
```

The string 'click' may be used for different purposes and not unique. However, the click symbol is absolutely unique.

Use Case: Symbols as keys of non-public properties

Note that symbols only protect you from name clashes, not from unauthorized access
Use symbols when your requirement is one of these:

**Enum: To allow you to define constants with semantic names and unique values.**

```
const directions = {  UP   : Symbol( 'UP' ),  DOWN : Symbol( 'DOWN' ),  LEFT : Symbol( 'LEFT' ),
RIGHT: Symbol( 'RIGHT' )};
```

**Name Clashes: when you wanted to prevent collisions with keys in objects**
Privacy: when you don't want your object properties to be enumerable
Protocols: To define how an object can be iterated.  Imagine, for instance, a library like dragula defining a protocol through Symbol.for(dragula.moves) .  You can add a method on that Symbol to any DOM element. If a DOM element follows the protocol,
then dragula could call the el[Symbol.for('dragula.moves')]() user-defined method to assert whether the element can be moved.

**Symbol usages**
**A) Using symbols as unique values**
Whenever you use a string or a number in your code, you should use symbols instead.
For example, you have to manage the status in the task management application. Prior to ES6, you would use strings such as open, in progress, completed, canceled, and on hold to represent different statuses of a task.
In ES6, you can use symbols as follows:

```
let statuses = {
    OPEN: Symbol('Open'),
    IN_PROGRESS: Symbol('In progress'),
    COMPLETED: Symbol('Completed'),
    HOLD: Symbol('On hold'),
    CANCELED: Symbol('Canceled')
};
// complete a task
console.log(statuses.COMPLETED.toString()); //"Symbol(Completed)"
```

Using symbol as the computed property name of an object
You can use symbols as computed property names. See the following example.

```
let status = Symbol('status');
let task = {
    [status]: statuses.OPEN,
    description: 'Learn ES6 Symbol'
};
console.log(task);

//{ description: 'Learn ES6 Symbol', [Symbol(status)]: Symbol(Open) }
```

Symbol.iterator
The Symbol.iterator specifies whether a function will return an iterator for an object.

The objects that have Symbol.iterator property are called iterable objects. In ES6, all collection objects (Array, Set and Map) and strings are iterable objects.  ES6 provides the for…of loop that works with the iterable object as in the following example.

```
var numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}

// 1
// 2
// 3
```

Internally, JavaScript engine first calls the Symbol.iterator method of the numbers array to get the iterator object. Then, it calls the iterator.next() method and copies the value property fo the iterator object into the num variable.  After three iterations, the done property of the result object is true, the loop exits.

You can access the default iterator object via System.iterator symbol as follows:

```
var iterator = numbers[Symbol.iterator]();

console.log(iterator.next()); // Object {value: 1, done: false}
console.log(iterator.next()); // Object {value: 2, done: false}
console.log(iterator.next()); // Object {value: 3, done: false}
console.log(iterator.next()); // Object {value: undefined, done: true}
```

**The object type**

In JavaScript, an object is a collection of properties, where each property is defined as a key-value pair.

The following example defines an empty object using the object literal form:

```
var emptyObject = {};
```

The following example defines the person object with two properties:

```
var person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

A property name of an object can by any string. You can use quotes around the property name if it is not valid JavaScript identifier.

For example, if you have a property first-name, you must use the quotes such as "first-name" but firstName is a valid JavaScript identifier so the quotes are optional.

If you have more than one property, you use a comma ( ,) to separate the pairs.

JavaScript allows you to nest object as shown in the following example:

```
var contact = {
    firstName: 'John',
    lastName: 'Doe',
    email: 'john.doe@example.com',
    phone: '(408)-555-9999',
    address: {
        building: '4000',
        street: 'North 1st street',
        city: 'San Jose',
        state: 'CA',
        country: 'USA'
    }
}
```

The contact object consists of firstName, lastName, email, phone, and address properties. The address property itself is also an object that consists of building,  street, city, state, and country properties. You can access the properties of an object by using two notations: the dot notation (.) and array-like notation ( []).The following example uses the dot notation (.) to access the firstName and lastName properties of the contact object.

```
console.log(contact.firstName);
console.log(contact.lastName);
```

To get property of a nested object, you use the following form:

```
console.log(contact.address.country);
```

If you refer to a non-existent property, you will get an undefined value as follows:

```
console.log(contact.age); // undefined
```

The following example uses the array-like notation to access the email and phone properties of the contact object.

```
console.log(contact['phone']); // '(408)-555-9999'
console.log(contact['email']); // 'john.doe@example.com'
```

Besides the object literal form, you can use the new keyword to create a new object as follows:

let customer = new Object();
And assign the property of the object a value:

customer.name = 'ABC Inc.';

## JavaScript Number

var aNumber= 10
console.log(aNumber.toStrring()); // "10"

Number is a primitive wrapper object used to represent and manipulate numbers like 37 or -9.25.
The Number constructor contains constants and methods for working with numbers. Values of other types can be converted to numbers using the Number() function.

In modern JavaScript, there are two types of numbers:

- Regular numbers in JavaScript are stored in 64-bit format IEEE-754, also known as "double precision floating point numbers".

These are numbers that we're using most of the time.

- BigInt numbers, to represent integers of arbitrary length. They are sometimes needed, because a regular number can't exceed 253 or be less than -253.

As bigints are used in few special areas, we devote them a special chapter BigInt.

JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63:

**Precision**
Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

Example
var x = 999999999999999;   // x will be 999999999999999
var y = 9999999999999999;  // y will be 10000000000000000
The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

If you add a number and a string, the result will be a string concatenation:

**Example**

```
var x = 10;
var y = "20";
var z = x + y;          // z will be 1020 (a string)


var aNumber= 10
console.log(aNumber.toString()); // "10"
console.log(12..toString(2));
```

**Two dots to call a method**

Please note that two dots in 123456..toString(36) is not a typo. If we want to call a method directly on a number, like toString in the example above, then we need to place two dots .. after it.

If we placed a single dot: 123456.toString(36), then there would be an error, because JavaScript syntax implies the decimal part after the first dot. And if we place one more dot, then JavaScript knows that the decimal part is empty and now goes the method.
Also could write (123456).toString(36).

example:

```
var x = 123e5;    // 12300000
var y = 123e-5;   // 0.00123
```

By default, JavaScript displays numbers as base 10 decimals. But you can use the toString() method to output numbers from base 2 to base 36. Hexadecimal is base 16. Decimal is base 10. Octal is base 8. Binary is base 2.

Example

```
var myNumber = 32;
myNumber.toString(10);  // returns 32
myNumber.toString(32);  // returns 10
myNumber.toString(16);  // returns 20
myNumber.toString(8);   // returns 40
myNumber.toString(2);   // returns 100000
```

Numbers Can be Objects, Normally JavaScript numbers are primitive values created from literals:

```
var x = 123;
```

But numbers can also be defined as objects with the keyword new:

```
var y = new Number(123);
```

Example

```
var x = 123;
var y = new Number(123);

// typeof x returns number
// typeof y returns object
```

Do not create Number objects. It slows down execution speed. The new keyword complicates the code. This can produce some unexpected results:

When using the == operator, equal numbers are equal:

Example

```
var x = 123;
var y = new Number(123);

if (x == y)
  console.log("true");
  else
   console.log("False") ;

// (x == y) is true because x and y have equal values

var aNumber = new Number(10);
console.log(aNumber.toStrring()); // "10"

console.log(aNumber.toString(2)); // "1010"
var x = 10;
console.log(x.toString(16)); // "a"

var numberObject = new Number(10);
```

```
var number = 10;

// typeof
console.log(typeof numberObject);
console.log(typeof number);
// instanceof
console.log(numberObject instanceof Number); // true
console.log(number instanceof Number); // false
```

Output:

```
-------
"object"
"number"
true
false
```

Some More example with Number Formatting:

```
var distance = 19.006
console.log(distance.toFixed(2)); // 19.01
var x = 10, y = 100, z = 1000;

console.log(x.toExponential());
console.log(y.toExponential());
console.log(z.toExponential());

// "1e+1"
// "1e+2"
// "1e+3"
var x = 9.12345;

console.log(x.toPrecision());    // '9.12345'
console.log(x.toPrecision(4));   // '9.123'
console.log(x.toPrecision(3));   // '9.12'
console.log(x.toPrecision(2));   // '9.1'
console.log(x.toPrecision(1));   // '9'
```

## JavaScript Number object vs. primitive number

The following table illustrates the differences between a Number object and a primitive number:

| Operator | Number object | Number value |
|---|---|---|
| typeof | "object" | "number" |
| instanceof Number | True | false |

```
var numberObject = new Number(10);
var number = 10;

// typeof
console.log(typeof numberObject); // object
console.log(typeof number); // number
// instanceof
console.log(numberObject instanceof Number); // true
console.log(number instanceof Number); // false
```

## JavaScript boolean primitive type:

JavaScript provides a boolean primitive type that has two values of true and false. The following example declares two variables that hold boolean values of false and true:

```
let isPending = false;
let isDone = true;
```

When you apply the  typeof operator to a variable that holds primitive boolean value, you get the boolean as the following example:

```
console.log(typeof(isPending)); //  boolean
console.log(typeof(isDone)); // boolean
```

**JavaScript Boolean object**

In addition to the boolean primitive type, JavaScript also provides you with the global Boolean() function, with the letter B in uppercase, to cast a value of another type to boolean.

The following example shows you how to use the Boolean() function to convert a string into a boolean value. Because the string is not empty, therefore, it returns true.

```
let a = Boolean('Hi');
console.log(a); // true
console.log(typeof(a)); // boolean
```

The Boolean is also a wrapper object of the boolean primitive type. It means that when you use the Boolean constructor and pass in either true or false, you create a Boolean object.

```
let b = new Boolean(false);
```

```
var x = false;
var y = new Boolean(false);

// (x == y) is true because x and y have equal values
```

But Comparing two JavaScript objects will always return false.

```
var x = new Boolean(false);
var y = new Boolean(false);

// (x == y) is false because objects cannot be compared
```

```
let a = new Boolean('Hi');
console.log(a); // true
console.log(typeof(a)); // Object
```

Here the output will be object, as we have created an object, But if we remove the new, then it will be considered as boolean.

```
let a =  Boolean('Hi');
console.log(a); // true
console.log(typeof(a)); // boolean
```

The Boolean is also a wrapper object of the boolean primitive type. It means that when you use the Boolean constructor and pass in either true or false, you create a Boolean object.

```
let b = new Boolean(false);
```

To get the primitive value back, you call the valueOf() method of the Boolean object as follows:

```
console.log(b.valueOf()); // false
```

```
let user = {
  name: "John",
  money: 1000,

  // for hint="string"
  toString() {
    return `{name: "${this.name}"}`;
  },

  // for hint="number" or "default"
  valueOf() {
    return this.money;
  }

};

alert(user); // toString -> {name: "John"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

```
let user = {
  name: "John",

  toString() {
    return this.name;
  }
};

alert(user); // toString -> John
alert(user + 500); // toString -> John500
```

All objects are true in a boolean context. There are only numeric and string conversions.
The numeric conversion happens when we subtract objects or apply mathematical functions.
For instance, Date objects (to be covered in the chapter Date and time) can be subtracted, and the result of date1 - date2 is the time difference between two dates. As for the string conversion – it usually happens when we output an object like alert(obj) and in similar contexts.

**The valueOf() method returns the primitive value of the specified object and The toString() method returns a string representing the object.**

1 + 1 // 2
'1' + 1 // '11' Both already primitives, RHS converted to string, '1' + '1', '11'
1 + [2] // '12' [2].valueOf() returns an object, so `toString` fallback is used, 1 + String([2]), '1' + '2', 12
1 + {} // '1[object Object]' {}.valueOf() is not a primitive, so toString fallback used, String(1) + String({}), '1' + '[object Object]', '1[object Object]'
2 - {} // NaN {}.valueOf() is not a primitive, so toString fallback used => 2 - Number('[object Object]'), NaN
+'a' // NaN `ToPrimitive` passed 'number' hint), Number('a'), NaN
+'' // 0 `ToPrimitive` passed 'number' hint), Number(''), 0
+'-1' // -1 `ToPrimitive` passed 'number' hint), Number('-1'), -1
+{} // NaN `ToPrimitive` passed 'number' hint', `valueOf` returns an object, so falls back to `toString`, Number('[Object object]'), NaN
1 + 'a' // '1a' Both are primitives, one is a string, String(1) + 'a'
1 + {} // '1[object Object]' One primitive, one object, `ToPrimitive` passed no hint, meaning conversion to string will occur, one of the operands is now a string, String(1) + String({}), `1[object Object]`
[] + [] // '' Two objects, `ToPrimitive` passed no hint, String([]) + String([]), '' (empty string)
1 - 'a' // NaN Both are primitives, one is a string, `ToPrimitive` passed 'number' hint, 1-Number('a'), 1-NaN, NaN
1 - {} // NaN One primitive, one is an object, `ToPrimitive` passed 'number' hint, `valueOf` returns object, so falls back to `toString`, 1-Number([object Object]), 1-NaN, NaN
[] - [] // 0 Two objects, `ToPrimitive` passed 'number' hint => `valueOf` returns array instance, so falls back to `toString`, Number('')-Number(''), 0-0, 0

**JavaScript boolean vs. Boolean**

let foo = true;
let bar = new Boolean(false);
First, bar is an object so you can add a property to the bar object:

bar.primitiveValue = bar.valueOf();
console.log(bar.primitiveValue); // false
However, you cannot do it with the primitive boolean variable like the foo variable:

foo.name = 'primitive';
console.log(foo.name); // undefined

Second, the typeof of Boolean object returns object, whereas the typeof of a primitive boolean value returns boolean.

```
console.log(typeof foo); // boolean
console.log(typeof bar); // object
```

Third, when applying the instanceof operator to a Boolean object, it returns true. However, it returns false if you apply the instanceof operator to a boolean value.

```
console.log(foo instanceof Boolean); // false
console.log(bar instanceof Boolean); // true
```

It is a good practice to never use the Boolean object because it will create many confusions especially when you use it in an expression.
See the following example.

```
let falseObj = new Boolean(false);
if (falseObj) {
    console.log('weird part of the Boolean object');
}
```

Output:
weird part of the Boolean object

Because **falseObj** is an object, and JavaScript engine coerces it to a boolean value of true . As a result, the statement inside the if block is executed.

| Operator | boolean | Boolean |
|---|---|---|
| typeof | boolean | object |
| instanceof Boolean | false | true |

Note: As a Best practice, It is recommended that you use the Boolean() function to convert a value of a different type to a Boolean type but you should never use the Boolean as a wrapper object of a primitive boolean value.

## Java Script Operator:

The simplest operators in JavaScript are unary operators. A unary operator works on one operand. The unary operators in JavaScript are:

- Unary plus (+) – convert an operand into a number
- Unary minus (-) – convert an operand into a number and negate the value after that.
- prefix / postfix increments (++) – add one to its operand
- prefix / postfix decrements (--) – subtract one from its operand

**Unary plus / minus**

The unary plus operator is a simple plus sign + and the unary minus is the minus sign -. You can place the unary plus or minus in front of a variable as follows:

```
let a = 10;
a = +a; // 10
console.log(a);
a = -a; // -10
console.log(a);
```

If the value is a number, the unary plus operator does not take any effect whereas the unary minus negates the value.
In case you apply the unary plus or minus on a non-numeric value, it performs the same conversion as the Number() function.

```
let s = '10';
console.log(+s); // 10
```

In this example, s is a string. However, when we placed the unary plus operator in front of it, the string s is converted to a number.

```
let a = 'a';
a = +a; // 10
console.log(a);
a = -a; // -10
console.log(a); // NaN
```

The following example shows how the unary operator converts boolean values into numbers, false to 0 and true to 1.

```
let f = false,
```

```
    t = true;
console.log(+f); // 0
console.log(+t); // 1

let product = {
    valueOf: function () {
        return 60;
    }
};

console.log(+product); // 60
```

When you apply the unary plus or minus on an object that has the valueOf() method, the method is called to return the converted value. In case the returned value is NaN, the toString() method is called to get the converted value.

**Increment / Decrements operators**

This is same as C language/Java

```
let i = 10;

console.log(i--) // 10
console.log(--i) //8
```

Similar to the unary plus and minus, you can use the increment or decrement operator on a value of a string, Boolean, and object to convert these value into a number with the similar rules:

When used on a string that can be converted to a valid number, it converts the string to a number. If the string cannot be converted to a number, it returns NaN. When used on a Boolean, the variable is converted to a number, true becomes 1 and false becomes 0. When used on an object, the valueOf() is called first. If the result is NaN then the toString() is called to return the converted value.

## An Introduction to JavaScript Logical Operators

**JavaScript provides three logical operators:**

1. ! (Logical NOT)
2. || (Logical OR)
3. && (Logical AND)

The meaning  is same as C/JavaScript, But certain other thing need to be discussed.
When you apply the ! operator to a non-Boolean value. The ! operator first converts the value to a boolean value and then negates it. For example:

The following example shows how to use the ! operator:

!a
The logical ! operator works based on the following rules:

If a is undefined, the result is true.
If a is null, the result is true.
If a is a number other than 0, the result is false.
If a is NaN, the result is true.
If a is null, the result is true.
If a is an object, the result is false.
If a is an empty string, the result is true. In case a is a non-empty string, the result is false.

The following demonstrates the results of the logical ! operator when we apply it to a non-boolean value:

```
console.log(!undefined); // true
console.log(!null); // true
console.log(!20); //false
console.log(!0); //true
console.log(!NaN); //true
console.log(!{}); // false
console.log(!''); //true
console.log(!'OK'); //false
console.log(!false); //true
console.log(!true); //false
```

**Double-negation (!!)**
Sometimes, you may see the double negation (!!) in the code. The !! uses the logical NOT operator (!) twice to convert a value to its real boolean value. The result is the same as using the Boolean() function. For example:

```
let counter = 10;
```

console.log(!!counter); // true
The first ! operator returns a Boolean value of the counter variable. And the second one ! negates that result and returns the real boolean value of the counter variable.

**Chain of && operators**
The following expression uses multiple && operators:

let result = value1 && value2 && value3;
The && operator carry the following:

Evaluates values from left to right.
For each value, converts it to a boolean. If the result is false, stops and returns the original value. If all values are truthy values, returns the last value.

In other words, The && operator returns the first falsy value or the last value if none were found. If a value can be converted to true, it is so-called a truthy value. If a value can be converted to false, it is so-called falsy value.

**A chain of || operators**
The following example shows how to use multiple || operators in an expression:

let result = value1 || value2 || value3;
The || operator does the following:

Evaluates values from left to right.
For each value, converts it to a boolean value. If the result of the conversion is true, stops and returns the value. If all values have been evaluated to false, returns the last value.
In other words, the chain of the || operators return the first truthy value or the last one if no truthy value was found.

**Logical operator precedence**
The precedence of the logical operator is in the following order from the highest to lowest:

Logical NOT (!)
Logical AND (&&)
Logical OR (||)

## JavaScript Assignment Operators

| Operator | Meaning | Description |
|---|---|---|
| a = b | a = b | Assigns the value of b to a. |
| a += b | a = a + b | Assigns the result of a plus b to a. |
| a -= b | a = a - b | Assigns the result of a minus b to a. |
| a *= b | a = a * b | Assigns the result of a times b to a. |
| a /= b | a = a / b | Assigns the result of a divided by b to a. |
| a %= b | a = a % b | Assigns the result of a modulo b to a. |
| a &=b | a = a & b | Assigns the result of a AND b to a. |
| a \|=b | a =a \| b | Assigns the result of a OR b to a. |
| a ^=b | a = a ^ b | Assigns the result of a XOR b to a. |
| a <<= b | a = a << b | Assigns the result of a shifted left by b to a. |
| a >>= b | a = a >> b | Assigns the result of a shifted right (sign preserved) by b to a. |
| a >>>= b | a = a >>> b | Assigns the result of a shifted right by b to a. |

">>" is a signed right shift, while ">>>" is an unsigned right shift. What this means is that if you use a signed shift on a negative number, the result will still be negative. Signed right shifting by one is equivalent to dividing by two, even if the number is negative.

## A Comprehensive Look at JavaScript Comparison Operators

| Operator | Meaning |
|---|---|
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| == | equal to |

| Operator | Meaning |
| --- | --- |
| != | not equal to |

Same as Normal C/Java.

Comparing a number with a value of another type
If an operand is a number while the other is not, JavaScript converts the non-numeric operand to a number and performs comparison numerically.

```
console.log(10 < '20'); // true
```

In this example, the string '20'is converted to 20 and compared with the number 10. Here is an example:

```
console.log(10 == '10'); // true
```

In this example, JavaScript converts the string '10' to the number 10 and compares the result with the number 10 that results in true.

But If you  write like below: as Both are diferent.

```
console.log(10 == 'A'); // False
```

**Comparing an object with a non-object**

If an operand is an object, JavaScript calls the valueOf() method of that object to get the value for comparison. If the object doesn't have the valueOf() method, JavaScript then calls the toString() method and uses the returned value for comparison.
See the following example:

```
let apple = {
  valueOf: function() {
    return 10;
  }
};

let orange = {
  toString: function() {
    return '20';
  }
};
console.log(apple > 10); // false
console.log(orange == 20); // true
```

If your class does not have both, then you should use the  the one the class has. like below

```
let orange = {
  value: function() {
    return '20';
  }
};

console.log(orange.value() == 20); // true
```

**Comparing a Boolean with another value**

If an operand is a Boolean, JavaScript converts it to a number and compares the converted value with the other operand; true will convert to 1 and false will convert to 0.

```
console.log(true > 0); // true
console.log(false < 1); // true
console.log(true > false); // true
console.log(false > true); // false
console.log(true >= true); // true
console.log(true <= true); // true
console.log(false <= false); // true
console.log(false >= false); // true
```

**Comparing null and undefined**

In JavaScript, null equals undefined. It means that the following expression returns true.

```
console.log(null == undefined); // true
```

Comparing NaN with other values

If either operand is NaN, then the equal operator(==) returns false.

```
console.log(NaN == 1); // false
console.log(NaN == NaN); // false
```

**Strict equal (===) and not strict equal (!==)**

Besides the comparison operators above, JavaScript provides the strict equal ( ===) and not strict equal  ( !==) operators.

```
console.log("10" == 10); // true
console.log("10" === 10); // false
```

**Java Script if else is same , JavaScript Ternary Operator , JavaScript** switch case **statement ,while loop, do while loop, for loop, label statement same as C/Java.**

## The **label** statement

```
outer: for (let i = 0; i < 5; i++) {
   console.log(i); }
```

## JavaScript String

JavaScript strings are primitive values and immutable. Literal strings are delimited by single quotes ('), double quotes ("), or backticks (`) The length property returns the length of the string. Use the >, >=, <, <=, == operators to compare two strings.

examples:
I will use only exmples as, it is self explanatory, whenever i found new JS specific concept, will elaborate.

```
let name = 'John';
let str = 'Hello ' + name;
```

**Converting values to string**

To convert a non-string value to a string, you use one of the following:

```
String(n);
'' + n
n.toString()
```

Note that the toString() method doesn't work for undefined and null.

When you convert a string to a boolean, you cannot convert it back via the Boolean():

```
let status = false;
let str = status.toString(); // "false"
let back = Boolean(str); // true
```

In this example:

First, the status is a boolean variable.
Then, the toString() returns the string version of the status variable, which is false. Finally, the Boolean() converts the "false" string back to the Boolean that results in true because "false" is a non-empty string.
Note that only string for which the Boolean() returns false, is the empty string ('');

**JavaScript String type:**
The String type is object wrapper of the string primitive type and can be created by using the String constructor as follows:

```
let str = new String('JavaScript String Type');
```

In this example, the value of the length property is 22 that also is the number of characters in the 'JavaScript String Type'.

To get the primitive string value, you use one of the following methods of the string object: valueOf(), toString(), and toLocaleString().

```
console.log(str.valueOf());
console.log(str.toString());
console.log(str.toLocaleString());

console.log(str.charAt(0)); // J
'literal string'.toUpperCase();
let firstName = 'John';
let fullName = firstName.concat(' ','Doe');
let firstName = 'John';
let fullName = firstName + ' ' + 'Doe';
console.log(fullName); // "John Doe"
let str = "JavaScript String";

console.log(str.substr(0, 10)); // "JavaScript"
console.log(str.substr(11,6)); // "Stri
let str = "JavaScript String";
console.log(str.substring(4, 10)); // "Script"
let str = "This is a string";
console.log(str.indexOf("is")); // 2

//! Removing the whitespace
let rawString = ' Hi  ';
let strippedString = rawString.trim();
console.log(strippedString); // "Hi"
```

Note that the trim() method creates a copy of the original string with whitespace characters stripped, it doesn't change the original string.

Notice that the trim() method is only available since ES5.

ES6 introduced two new methods for removing whitespace characters from a string:

trimStart() returns a string with whitespace characters stripped from the beginning of a string.
trimEnd() returns a string with the whitespace characters stripped from the end of a string.

The localeCompare() returns one of three values: -1, 0, and 1.

If the first string comes before the second string alphabetically, the method returns -1.
If the first string comes after the second string alphabetically, the method returns 1.
If two strings are equal, the method returns 0.
For example:

```
console.log('A'.localeCompare('B')); // -1
console.log('B'.localeCompare('B')); // 0
console.log('C'.localeCompare('B')); // 1
```

Matching patterns
The match() method allows you to match a string with a specified regular expression and get an array of results.

The match() method returns null if it does not find any match. Otherwise, it returns an array containing the entire match and any parentheses-capture matched results.

If the global flag (g) is not set, the element zero of the array contains the entire match. Here is an example.

```
let expr = "1 + 2 = 3";
let matches = expr.match(/\d+/);
console.log(matches[0]); // "1"
```

In this example, the pattern matches any number in the expr string.

In case the global flag ( g) is set, the elements of the result array contain all matches as follows:

```
matches = expr.match(/\d+/g);
matches.forEach(function(m) {
    console.log(m);
});
// "1"
// "2"
// "3"
```

```
let str = "This is a test of search()";
let pos = str.search(/is/);
console.log(pos); // 2

let str = "the baby kicks the ball";
// replace "the" with "a"
let newStr = str.replace(/the/g, "a");
console.log(newStr); // "a baby kicks a ball"
console.log(str); // "the baby kicks the ball"
```

## JavaScript Arrays

JavaScript arrays are written with square brackets. Array items are separated by commas.
The following code declares (creates) an array called cars, containing three items (car names):
Array indexes are zero-based, which means the first item is [0], second is [1], and so on.
Example

```
var cars = ["Saab", "Volvo", "BMW"];
```

## The typeof Operator

You can use the JavaScript typeof operator to find the type of a JavaScript variable.
The typeof operator returns the type of a variable or an expression:

**Example**

```
typeof ""                 // Returns "string"
typeof "John"             // Returns "string"
typeof "John Doe"     // Returns "string"
typeof 0              // Returns "number"
typeof 314            // Returns "number"
```

```
typeof 3.14          // Returns "number"
typeof (3)           // Returns "number"
typeof (3 + 4)       // Returns "number"
```

**Empty Values**

An empty value has nothing to do with undefined. An empty string has both a legal value and a type.

Example

```
var car = "";    // The value is "", the typeof is "string"
```

## Difference Between Undefined and Null

undefined and null are equal in value but different in type:

```
typeof undefined          // undefined
typeof null               // object

null === undefined        // false
null == undefined         // true
```

**Complex Data**

The typeof operator can return one of two complex types:

1. function
2. object

The typeof operator returns "object" for objects, arrays, and null. The typeof operator does not return "object" for functions.

**Example**

```
typeof {name:'John', age:34}          // Returns "object"
typeof [1,2,3,4]                      // Returns "object" (not "array", see note below)
typeof null                           // Returns "object"
typeof function myFunc(){}            // Returns "function"
```

## JavaScript Variables:

Declare JavaScript variables using var keyword

**var message;**

A variable name can be any valid identifier. The message variable is declared and hold a special value undefined.

```
var message = "Hello",    counter = 100;
```

Empty Values
An empty value has nothing to do with undefined.
An empty string has both a legal value and a type.

Example
```
var car = "";    // The value is "", the typeof is "string"
```

## Difference Between Undefined and Null

undefined and null are equal in value but different in type:
```
typeof undefined          // undefined
typeof null               // object

null === undefined        // false
null == undefined         // true
```

**Complex Data**
The typeof operator can return one of two complex types:

1. function
2. object

The typeof operator returns "object" for objects, arrays, and null. The typeof operator does not return "object" for functions.

**Example**
```
typeof {name:'John', age:34}   // Returns "object"
typeof [1,2,3,4]               // Returns "object" (not "array", see note below)
typeof null                    // Returns "object"
typeof function myFunc(){}     // Returns "function"
```

## Undefined vs. undeclared variables:

An undefined variable is a variable that has been declared. Because we have not assigned it a value, the variable used the undefined as its initial value. In contrast, an undeclared variable is the variable that has not been declared.

See the following example:

```
var message;

console.log(message); // undefined
console.log(test); // ReferenceError: test is not defined
```

In this example, the message variable is declared but not initialized therefore its value is undefined whereas the test variable has not been declared hence accessing it causes a ReferenceError.

## Global and local variables

In JavaScript, all variables exist within a scope that determines the lifetime of the variables and which part of the code can access them.
JavaScript mainly has global and function scopes. ES6 introduced a new scope called block scope.

If you declare a variable in a function, JavaScript adds the variable to the function scope. In case you declare a variable outside of a function, JavaScript adds it to the global scope.
In JavaScript, you define a function as follows:

```
function functionName() {
  // logic
}
```

The following example defines a function named say that has a local variable named message.

```
function say() {
  var message = "Hi";
  return message;
}
```

The message variable is a local variable. In other words, it only exists inside the function
If you try to access the message outside the function as shown in the following example, you will get a ReferenceError because the message variable was not defined:

```
function say() {
    var message = 'Hi';
}
console.log(message); // ReferenceError
```

## Variable shadowing

See the following example:

```
// global variable
var message = "Hello";
function say() {
    // local variable
    var message = 'Hi';
    console.log(message); // which message?
}
say();// Hi
console.log(message); // Hello
```

## Accessing global variable inside the function

See the following example:

```
// global variable
var message = "Hello";
function say() {
    // local variable
    message = 'Hi';
    console.log(message); // which message?
}
say();// Hi
console.log(message); // Hi
```

## Non-strict mode:

The following example defines a function and declares a variable message. However, the var keyword is not used.

```
function say() {
    message = 'Hi'; // what?
    console.log(message);
}
say(); // Hi
console.log(message); // Hi
```

When you execute the script, it outputs the Hi string twice in the output.

Because when we call the say() function, the JavaScript engine looks for the variable named message inside the scope of the function. As a result, it could not find any variable declared with

that name so it goes up to the next immediate scope which is the global scope in this case and asks whether or not the message variable has been declared. Because the JavaScript engine couldn't find any of global variable named message, so it creates a new variable with that name and adds it to the global scope.

## strict mode

To avoid creating a global variable accidentally inside a function because of omitting the var keyword, you use the strict mode by adding the "use strict";  at the beginning of the JavaScript file (or the function) as follows:

```
"use strict";

function say() {
    message = 'Hi'; // ReferenceError
    console.log(message);
}
say(); // Hi
console.log(message); // Hi
```

## Using let and const keywords:

From ES6, you can use the let keyword to declare one or more variables. The let keyword is similar to the var keyword.

However, a variable is declared using the let keyword is block-scoped, not function or global-scoped like the var keyword

**#1: Variable scopes**

The var variables belong to the global scope or local scope if they are declared inside a function

The let variables are blocked scopes:

**#2: Creating global properties:**

The global var variables are added to the global object as properties. The global object is window on the web browser and global on Node.js:

The global var variables are added to the global object as properties. The global object is window on the web browser and global on Node.js:

```
var counter = 0;

console.log(window.counter); //  0
```

However, the let variables are not added to the global object:

```
let counter = 0;

console.log(window.counter); // undefined
```

Note: It is running in browser, But in node, it is showing undefined.

**#3: Redeclaration**

The var keyword allows you to redeclare a variable without any issue.

```
var counter = 10;

var counter;

console.log(counter); // 10
```

If you redeclare a variable with the let keyword, you will get an error:

```
let counter = 10;

let counter; // error
```

**#4: The Temporal dead zone**

The let variables have temporal dead zones while the var variables don't. To understand the temporal dead zone, let's examine the life cycles of both var and let variables, which have two steps: creation and execution.

**The var variables**

In the creation phase, the var variables are assigned storage spaces and immediately initialized to undefined. In the execution phase, the var variables are assigned the values specified by the assignments if there are ones.  If there aren't, the values of the variables remain undefined.

**The let variables**

In the creation phase, thelet variables are assigned storage spaces but are not initialized. Referencing uninitialized variables will cause a ReferenceError. The let variables have the same execution phase as the var variables.

The temporal dead zone starts from the block until the let variable declaration is processed. In other words, it is where you cannot access the let variables before they are defined.

Variables and constants declared with let or const are not hoisted!

**Global Scope**
Global variables can be accessed from anywhere in a JavaScript program.
Variables declared Globally (outside any function) have Global Scope.

**Function Scope**
Variables declared Locally (inside a function) have Function Scope.

Local variables can only be accessed from inside the function where they are declared.

**JavaScript Block Scope**

Variables declared with the var keyword cannot have Block Scope. Variables declared inside a block {} can be accessed from outside the block.

Example
```
{
  var x = 2;
}
// x CAN be used here
```

Variables declared with the let keyword can have Block Scope.
Variables declared inside a block {} cannot be accessed from outside the block:

**Example**
```
{
  let x = 2;
}
// x can NOT be used here
```

Redeclaring Variables

Redeclaring a variable using the var keyword can impose problems. Redeclaring a variable inside a block will also redeclare the variable outside the block:

Example
```
var x = 40;
```

```
// Here x is 10
{
  var x = 2;
  // Here x is 2
}
// Here x is 2
```

```
C:\Users\haramohan.sahu\Desktop\JavaScriptTraining\Practice>node let.js
X value Global scope  = 40
X value inside Block = 2
X value Outside block  = 2
```

But instead of var inside the block, if you declare with let, then It will have no impact on outside block. That means, you will get 40 as answer. But Variables declared with var and let are quite similar when declared outside a block.

They will both have Global Scope:

```
var x = 2;       // Global scope
let x = 2;       // Global scop
```

**example**:

```
var x=40;
let aa=88;
console.log("X value Global scope  = "+aa); // aa=88
{
var xa = 2;
aa=99;
console.log("X value inside Block = "+aa);//aa=99
}
console.log("X value Outside block  = "+aa);//aa=99
```

## Global Variables in HTML

With JavaScript, the global scope is the JavaScript environment.

### In HTML, the global scope is the window object.

Global variables defined with the var keyword belong to the window object:

**example**:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Global Variables</h2>
<p>In HTML, global variables defined with <b>var</b>, will become window variables.</p>
<p id="demo"></p>
<script>
var carName = "Volvo";

// code here can use window.carName
document.getElementById("demo").innerHTML = "I can display " + window.carName;
</script>
</body>
</html>
```

But If I use **let** instead of VAR, then undefined error will come.

**Redeclaring**

Redeclaring a JavaScript variable with var is allowed anywhere in a program, But in the same scope var and let can not be used for the same variable name.

```
var x = 2;       // Allowed
let x = 3;       // Not allowed

{
  var x = 4;   // Allowed
  let x = 5   // Not allowed
}
```

That is why Redeclaring a let variable with let, in the same scope, or in the same block, is not allowed:

Hoisting: Variables defined with var are hoisted to the top and can be initialized at any time, But cannot be hoisted if it is declared as let.

Const variable:

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

But Declaring a variable with const is similar to let when it comes to Block Scope.

The x declared in the block, in this example, is not the same as the x declared outside the block:

```
var x = 10;
// Here x is 10
{
  const x = 2;
  // Here x is 2
}
// Here x is 10
```

JavaScript const variables must be assigned a value when they are declared:

**Incorrect**

```
const PI;
PI = 3.14159265359;
```

**Correct**

```
const PI = 3.14159265359;
```

The keyword const is a little misleading.

It does NOT define a constant value. It defines a constant **reference** to a value.

Because of this, we cannot change constant primitive values, but we can change the properties of constant objects.

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

You can change the properties of a constant object:

**Example**

```
// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

// You can add a property:
car.owner = "Johnson";
```

But you can NOT reassign a constant object:

**Example**

```
const car = {type:"Fiat", model:"500", color:"white"};
car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

Similarly, you can change the elements of a constant array:

**Example**

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

**But you can NOT reassign a constant array:**

**Example**

```
const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"];    // ERROR

var x = 2;         // Allowed
const x = 2;       // Not allowed
{
  let x = 2;     // Allowed
  const x = 2;   // Not allowed
}

const x = 2;       // Allowed
const x = 3;       // Not allowed
x = 3;             // Not allowed
var x = 3;         // Not allowed
let x = 3;         // Not allowed
```

**Variables defined with const are not hoisted to the top. A const variable cannot be used before it is declared:**

**Example**

```
carName = "Volvo";    // You can NOT use carName here
const carName = "Volvo";
```

## Web Development Tools:

how to open the Console tab of web development tools to view messages issued by JavaScript.
Press F12 to view the Console tab of the dev tool

## Difference between Non-strict mode and strict mode;

The following example defines a function and declares a variable message. However, the var keyword is not used.

```
function say() {
    message = 'Hi'; // what?
    console.log(message);
}
say(); // Hi
console.log(message); // Hi
```

When you execute the script, it outputs the Hi string twice in the output.
Because when we call the say() function, the JavaScript engine looks for the variable named message inside the scope of the function. As a result, it could not find any variable declared with that name so it goes up to the next immediate scope which is the global scope in this case and asks whether or not the message variable has been declared.

Because the JavaScript engine couldn't find any of global variable named message so it creates a new variable with that name and adds it to the global scope.

But if you "use strict", then it will throw error.
Reason, JavaScript in strict mode does not allow variables to be used if they are not declared.

To avoid creating a global variable accidentally inside a function because of omitting the var keyword, you use the strict mode by adding the "use strict", at the beginning of the JavaScript file (or the function)

You should always use the strict mode in your JavaScript code to eliminate some JavaScript silent errors and make your code run faster.

## JavaScript variable hoisting

When executing JavaScript code, the JavaScript engine goes through two phases:

- Parsing
- Execution

In the parsing phase, The JavaScript engine moves all variable declarations to the top of the file if the variables are global, or to the top of a function if the variables are declared in the function.

In the execution phase, the JavaScript engine assigns values to variables and execute the code.

Hoisting is a mechanism that the JavaScript engine moves all the variable declarations to the top of their scopes, either function or global scopes. If you declare a variable with the var keyword, the variable is hoisted to the top of its enclosing scope, either global or function scope.
As a result, if you access a variable before declaring it, the variable evaluates to undefined.

**Example**:

```
console.log(message); // undefined
var message;
```

The JavaScript engine moves the declaration of the message variable to the top, so the above code is equivalent to the following:

```
var message;
console.log(message); // undefined
```

If there were no hoisting, you would get a **ReferenceError** because you referenced to a variable that was not defined.

Note: **The JavaScript engine moves only the declaration of the variables to the top. However, it keeps the initial assignment of the variable remains intact**

**Consider another example:**

```
console.log(counter);
var counter = 100;
```

The above code is transformed to below code by Java Script Engine.

```
var counter;
console.log(counter); // undefined
counter = 100;
```

Reason for the above is JavaScript only hoists declarations, not initializations.

### The hoisting uses redundant `var` declarations without any penalty:

```
var counter;
var counter;
counter = 1;
console.log(counter); // 1
```

### The **var** variables

- In the creation phase, the `var` variables are assigned storage spaces and immediately initialized to `undefined`.
- In the execution phase, the `var` variables are assigned the values specified by the assignments if there are ones. If there aren't, the values of the variables remain `undefined`.

## JavaScript Identifiers:

All JavaScript variables must be identified with unique names. These unique names are called identifiers.

## Objects

As we know from the chapter Data types, there are eight data types in JavaScript. Seven of them are called "primitive", because their values contain only a single thing (be it a string or a number

or whatever). In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

An object can be created with figure brackets {…} with an optional list of *properties*. A property is a "key: value" pair, where key is a string (also called a "property name"), and value can be anything.
We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It's easy to find a file by its name or add/remove a file.

An empty object ("empty cabinet") can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```

## Literals and properties of Object

We can immediately put some properties into {...} as "key: value" pairs:

```
let user = {                 // an object
  name: "John",              // by key "name" store value "John"
  age: 30                    // by key "age" store value 30
};
```

Property values are accessible using the dot notation:

```
// get property values of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

The value can be of any type. Let's add a boolean one:

```
let user = {                 // an object
  name: "John",              // by key "name" store value "John"
  age: 30                    // by key "age" store value 30
};
// get property values of the object:
console.log( user.name ); // John
console.log( user.age ); // 30
user.isAdmin = true;
delete user.age;
console.log( user.age ); // undefined
console.log(user.isAdmin) //true
user.isAdmin = true;
```

We can also use multiword property names, but then they must be quoted:

```
let user = {
  name: "John",
```

```
  age: 30,
  "likes birds": true  // multiword property name must be quoted
};


let user = {
  name: "John",
  age: 30,
}
```

The last property in the list may end with a comma:

```
let user = {
  name: "John",
  age: 30,
}
```

## Object with const can be changed

Please note: an object declared as const can be modified.

For instance:

```
const user = {
  name: "John"
};

user.name = "Pete"; // (*)
alert(user.name); // Pete
```

It might seem that the line (*) would cause an error, but no. The const fixes the value of user, but not its contents.

The const would give an error only if we try to set user=... as a whole.

Same like Python in the above example.

Square brackets is used for multiword properties, the dot access doesn't work:

```
let user = {};
// set
user["likes birds"] = true;
// get
alert(user["likes birds"]); // true
```

```
// delete
delete user["likes birds"];
```

## Loops in JS

1. Whie
2. Do… while
3. For loop

```
while (expression) {
   // statement
}
For (;;)
{
}
Do{
}while();
```

## JavaScript String

JavaScript strings are primitive values. JavaScript strings are also immutable. It means that if you process a string, you will always get a new string. The original string doesn't change.

To create literal strings in JavaScript, you use single quotes or double quotes:

```
let str = 'Hi';
let greeting = "Hello";
```

### scaping special characters

To escape special characters, you use the backslash `\` character. For example:

- Windows line break: `'\r\n'`
- Unix line break: `'\n'`
- Tab: `'\t'`
- Backslash `'\'`

## Converting values to string

To convert a non-string value to a string, you use one of the following:

- String(n);
- ” + n
- n.toString()

Note that the `toString()` method doesn’t work for `undefined` and `null`. When you convert a string to a boolean, you cannot convert it back via the `Boolean()`:

```
let status = false;
let str = status.toString(); // "false"
let back = Boolean(str); // true
```

In this example:

- First, the `status` is a boolean variable.
- Then, the `toString()` returns the string version of the `status` variable, which is `false`.
- Finally, the `Boolean()` converts the `"false"` string back to the Boolean that results in true because `"false"` is a non-empty string.

Note that only string for which the `Boolean()` returns false, is the empty string (”);

## JavaScript String type

The `String` type is object wrapper of the string primitive type and can be created by using the `String` constructor as follows:

```
let str = new String('JavaScript String Type');
```

To get the primitive string value, you use one of the following methods of the string object: `valueOf()`, `toString()`, and `toLocaleString()`.

```
console.log(str.valueOf());
console.log(str.toString());
console.log(str.toLocaleString());
```

- trim(), trimStart(), and trimEnd() – remove whitespace characters from a string.
- padStart() and padEnd() – pad a string with another string until the result string reaches the given length.
- concat() – concatenate multiple strings into a new string.
- split() – split a string into an array of substrings.
- indexOf() – get the index of the first occurrence of a substring in a string.
- lastIndexOf() – find the index of the last occurrence of a substring in a string.
- substring() – extract a substring from a string.
- slice() – extract a part of a string.
- includes() – check if the string contains a substring.

## JavaScript Arrays

In JavaScript, an array is an ordered list of values. Each value is called an element specified by an index.

**Creating JavaScript arrays**

```
let scores = new Array();
let scores = Array(10); // array with size 10 , Notice the NEW operator

let scores = new Array(9,10,8,7,6); // this will create the array with these value
let athletes = new Array(3); // creates an array with initial size 3
let scores = new Array(1, 2, 3); // create an array with three numbers 1,2 3
let signs = new Array('Red'); // creates an array with one element 'Red'
```

***Getting the array size***

Typically, the `length` property of an array returns the number of elements. The following example shows how to use the `length` property:

```
let scores = Array(10);
let t = [1];
console.log(t.length); // 1
```

```
console.log(scores.length);//10
```

## Javascript stack using array

```
let stack = [];
stack.push(1);
console.log(stack); // [1]
stack.push(2);
console.log(stack); // [1,2]
stack.push(3);
console.log(stack); // [1,2,3]
stack.push(4);
console.log(stack); // [1,2,3,4]
stack.push(5);
console.log(stack); // [1,2,3,4,5]
```

Array length – show you how to use the length property of an array effectively.
Stack – implement the stack data structure using the Array's methods: push() and pop().
Queue – implement the queue data structure using the Array's methods: push() and shift()
splice() – manipulate elements in an array such as deleting, inserting, and replacing elements.
slice() – copy elements of an array.
index() – locate an element in an array.
every() – check if every element in an array passes a test.
some() – check if at least one element in an array passed a test.
sort() – sort elements in an array.
filter() – filter elements in an array
map() – transform array elements.
forEach() – loop through array elements.
reduce() – reduce elements of an array to a value.
join() – concatenate all elements of an array into a string separated by a seperator.
Multidimensional Array – learn how to work with multidimensional arrays in JavaScript.

## JavaScript Function

In JavaScript, functions are first-class objects, because they can have properties and methods just like any other object. What distinguishes them from other objects is that functions can be called. In brief, they are Function objects.

## Adding properties to primitives doesn't work:

```
var str = "heyhey";
var num = 25;
var bool = true;
str.prop = "a";
num.prop = "b";
bool.prop = "c";
console.log([str.prop, num.prop, bool.prop]);
```

It will throw error, It does not work, as it is not an object.

Every function in JavaScript is an instance of the Function constructor:

## Functions can also be created by a function expression.

example: 1

```
// x, y is the argument. 'return x + y' is the function body, which is the last in the argument list.
var add = new Function('x', 'y', 'return x + y');
var t = add(1, 2);
console.log(t);  // 3
```

example : 2

## The add function above may also be defined using a function expression:

```
var add = function(x, y) {
  return x + y;
};
var t = add(1, 2);
console.log(t); // 3
```

example : 3

```
function add(x, y) {
  return x + y;
```

```
}
var t = add(1, 2);
console.log(t); // 3
```

Example : 4

```
var add = ((x, y) => {
  return x + y;
});
// or
var add = ((x, y) => x + y);

var t = add(1, 2);
console.log(t); // 3
```

Since function is an object in Javascript, so we can  add properties to object.

example : 5

```
var funcObj = new Function();
funcObj.ename="Haramohan";
funcObj.eid=11;
funcObj.fun=(a) => {
  if (a == 1)
  {
    console.log(a);
  }else if (a==2){
    console.log(a);
  }
  else {
    console.log(a);
  }
}
funcObj.fun(1);
console.log(funcObj );
```

**A function instance has properties and methods.**

Functions must be in scope when they are called, but the function declaration can be hoisted

```
console.log(square(5));
/* ... */
function square(n) { return n * n }
```

But below code will not work as variable are hoisted but initialised with undefined, but this is not true with function.

```
console.log(square)    // square is hoisted with an initial value undefined.
console.log(square(5)) // Uncaught TypeError: square is not a function
const square = function(n) {
  return n * n;
}
```

## Adding properties to objects, including functions, works:

```
  var obj = {}; // same as "new Object()"
  var arrObj = []; // same as "new Array()"
  var strObj = new String("byebye"); // not typically used
  var numObj = new Number(50); // not typically used
  var boolObj = new Boolean(true); // not typically used
  var funcObj = new Function(); // not typically used
  var funcObj2 = function() {
      console.log("I can have properties too.");
  };
  obj.prop = "a";
  arrObj.prop = "b";
  strObj.prop = "c";
  numObj.prop = "d";
  boolObj.prop = "e";
  funcObj.prop = "f";
  funcObj2.prop = "g";
  console.log([obj.prop, arrObj.prop, strObj.prop, numObj.prop, boolObj.prop, funcObj.prop,
  funcObj2.prop]); // outputs ["a", "b", "c", "d", "e", "f", "g"]


  (function(){}).constructor === Function // true
```

Every JavaScript function is actually a Function object. This can be seen with the code (function(){}).constructor === Function, which returns true.
Constructor:
---
Function() -> This one Creates a new Function object.The Function constructor creates functions that execute in the global scope only.

**Instance Properties:**
---
1)Function.displayName -->eprecated
The display name of the function.

2)Function.length
Specifies the number of arguments expected by the function.
3)Function.name
The name of the function.

**Function.prototype.call():**
---
call() provides a new value of this to the function/method.
With call(), you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

```
function Product(name, price) {
  this.name = name;
  this.price = price;
}

function Food(name, price) {
  Product.call(this, name, price);
  this.category = 'food';
}
var t = new Food('cheese', 5);
console.log(t.name);
// expected output: "cheese"
```

Note: While the syntax of this function is almost identical to that of apply(), the fundamental difference is that call() accepts an argument list,

while apply() accepts a single array of arguments.

**Using call to chain constructors for an object:**

```
-------------------------------------------
function Product(name, price) {
  this.name = name;
  this.price = price;
}

function Food(name, price) {
  Product.call(this, name, price);
  this.category = 'food';
}

function Toy(name, price) {
  Product.call(this, name, price);
  this.category = 'toy';
}

const cheese = new Food('feta', 5);
const fun = new Toy('robot', 40);
```

**Using call to invoke a function and specifying the context for 'this'**

```
-----------------------------------------------------------------
function greet() {
  const reply = [this.animal, 'typically sleep between', this.sleepDuration].join(' ');
  console.log(reply);
}

const obj = {
  animal: 'cats',
  sleepDuration: '12 and 16 hours'
};

greet.call(obj);  // cats typically sleep between 12 and 16 hours
```

another example:

```
//! 4th way of calling function.
function employee( age, name)
```

```
{
  this,name=name;
  this.age=age;
  this.fun=function()
  {
    this.age+=10;
  }
};
var  tt = new employee(15,"hara");
//! if I call tt.fun(), this will increase the age value
function Mechanic(n)
{
  this.name=n;
}
var obj=new Mechanic("HaraMohan");

obj.myFunc=tt.fun;
//! Can I call obj.myFunc() ? , If i call it actualy  try to call fun & fun () will try to increase the age ( but here thisis used
// which is not pointing obj rather employee. so we need to bind employee obj that is tt to Mechanic object obj with the help of call())
console.log(tt.age);
obj.myFunc.call(tt);
```

**function expression:**

---

```
const getRectArea = function(width, height) {
  return width * height;
};

console.log(getRectArea(3, 4));
// expected output: 12
```

**Function() constructor:**

---

```
const sum = new Function('a', 'b', 'return a + b');

console.log(sum(2, 6));
// expected output: 8
```

```
function show()
{

}

var tt = new show();
```

Note:
Every object created out of a function say example  show(), then  that object has constructor and __proro__.
example:
tt.constructor ; -->  function foo(){}
tt.__proro__;  -->{constructor: ƒ} , this is an object

But show.prototype ; --> {constructor: ƒ} same as __proto__
This prototype poperty will have 2 thing
1) constructor: ƒ foo()
    constructor: ƒ foo()
    arguments: null
    caller: null
    length: 0
    name: "foo"
    prototype: {constructor: ƒ}
    __proto__: ƒ ()
    [[FunctionLocation]]: VM3291:1
    [[Scopes]]: Scopes[2]

2) __proto__: Object

show.constructor ; --> ƒ Function() { [native code] }
show.__proro__;  --> ƒ () { [native code] }

---
t.__proto__ ;
    {constructor: ƒ}

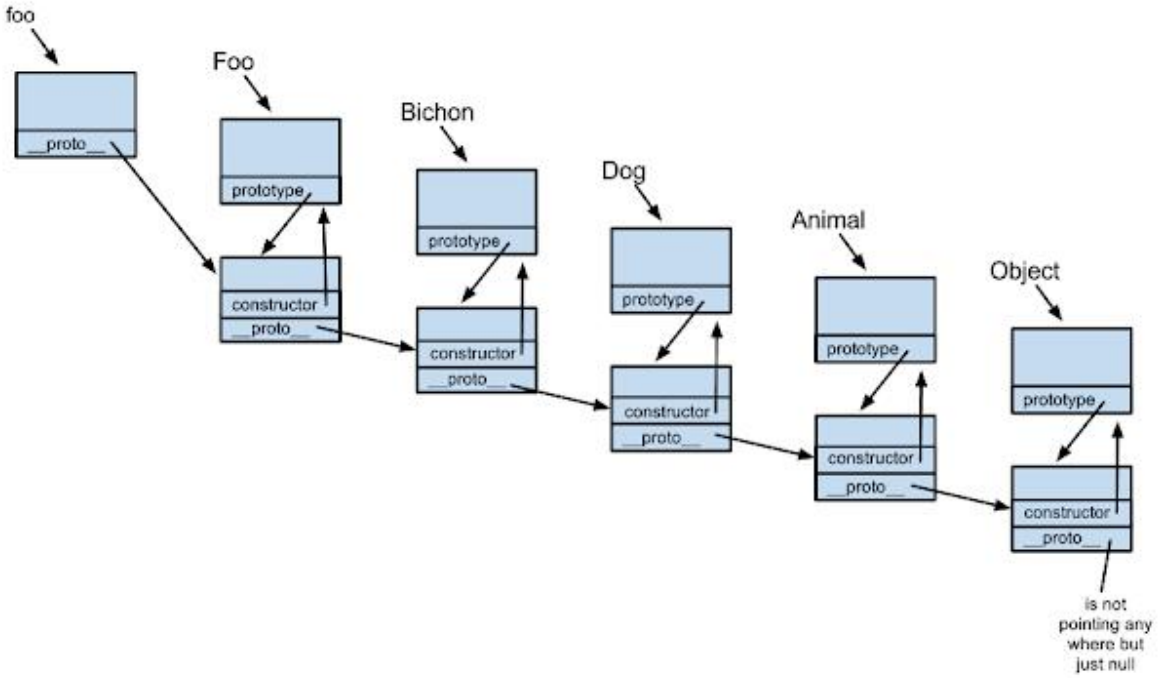
constructor: ƒ foo()
arguments: null
caller: null
length: 0
name: "foo"
prototype: {constructor: ƒ}
__proto__: ƒ ()
__proto__: Object

```
function foo(){}
var t = new foo();
console.log(t.__proto__ == foo.prototype) // true
```

## Core Java script concept:

## Object:

## Inheritance in JS:



the term **[[Prototype]]** is used. And that's the same as **__proto__**. It is often mentioned as the "**[[Prototype]]** internal property". And don't confuse this with a function's **prototype** property. One of the key points regarding **[[Prototype]]** is: "All objects have an internal property called **[[Prototype]]**. The value of this property is either **null** or a reference to an object and is used for implementing inheritance."

Object.getPrototypeOf(xx) == xx.__proto__

## Advantages of using prototype, vs defining methods straight in the constructor?

1. **Constructor approach:**

```
var Class = function () {

   this.calc = function (a, b) {
      return a + b;
   };

};
```

2. **Prototype approach:**

```
function Class () {};

Class.prototype.calc = function (a, b) {
   return a + b;
};
```

**Prototype approach:** Methods that inherit via the prototype chain can be changed universally for all instances, for example:

```
function Class () {}
Class.prototype.calc = function (a, b) {
   return a + b;
}

// Create 2 instances:
var ins1 = new Class(),
var ins2 = new Class();

// Test the calc method:
console.log(ins1.calc(1,1), ins2.calc(1,1));
// -> 2, 2

// Change the prototype method
```

```
Class.prototype.calc = function () {
   var args = Array.prototype.slice.apply(arguments),
      res = 0, c;

   while (c = args.shift())
      res += c;

   return res;
}

// Test the calc method:
console.log(ins1.calc(1,1,1), ins2.calc(1,1,1));
// -> 3, 3
```

- Notice how changing the method applied to both instances? This is because ins1 and ins2 share the same calc() function. In order to do this with public methods created during construction, we would have to assign the new method to each instance that has been created, which is an awkward task. This is because ins1 and ins2 would have their own, individually created calc() functions
- Another side effect of creating methods inside the constructor is poorer performance. Each method has to be created every time the constructor function runs. Methods on the prototype chain are created once and then "inherited" by each instance.

**Advantages of prototype methods:**

1. When you define methods via prototype, they are shared among all Your Class instances. As a result, the total size of such instances is < than if you define methods in constructor; There are tests that show how method definition via prototype decrease the total size of html page and as a result a speed of its loading.

2. Another advantage of methods, defined via prototype - is when you use inherited classes, you may override such methods and in the overridden method of the derived class you may invoke the method of base class with the same name, but with methods defined in constructor, you cannot do this.

3. The prototype also allows us to add a new method at a later point that can be accessed by already-existing objects of that type or globally modify a method for all objects of a type.

4. Adding methods in the prototype gives our code better abstraction.

5. For example, since the methods are not tied to the instances, we are able to call them on non-instances which are compatible enough. Like this: Array.prototype.slice.call({0:'a', 1:'b', length:2}); // ["a", "b"]

6. If we only defined your methods on the instances, it would be necessary to create an useless instance in order to borrow the method: [].slice.call({0:'a', 1:'b', length:2}); // ["a", "b"]

7. Additionally, defining a method inside the constructor means that each instance will receive a different copy. new Class (1,1).cal === new Class (1,1).cal // false
8. Faster to create instances.
9. Uses less memory
10.

**Constructor approach:**

With this approach, it takes more memory, that is the constructor way, a new set of functions is created every time the Class constructor is called, using more memory.

Every objects has it's own copy of the function. It ends up taking up more memory as the number of instances goes up and if for some reason you want to modify *all* **objects** at run time, we have to change every single instances function instead of modifying the prototype.

It's the difference between everyone looking at the same "copy" of Wikipedia versus everyone saving all of Wikipedia to their hard drive and reading that. It pointlessly uses up extra hard drive space and if Wikipedia is updated, everyone is wrong until they download the new version.

Whereas If we are creating lots of Class objects, we should use the prototype approach. This way, all "instances" (i.e. objects created by the Class constructor) will share one set of functions.

Better approach is, if our methods that need access to local private constructor variables are defined in the constructor while other methods are assigned to the prototype.
So prototype methods cannot access private variables.
A constructor's prototype provides a way to share methods and values among instances via the instance's private **[[Prototype]]** property.

**For example**,
The code below uses a local variable in the constructor to keep track of the number of times this dog has barked while keeping the actual number private, so the barking-related methods are defined inside the constructor. Tail wagging does not require access to the number of barks, therefore that method can be defined on the prototype.

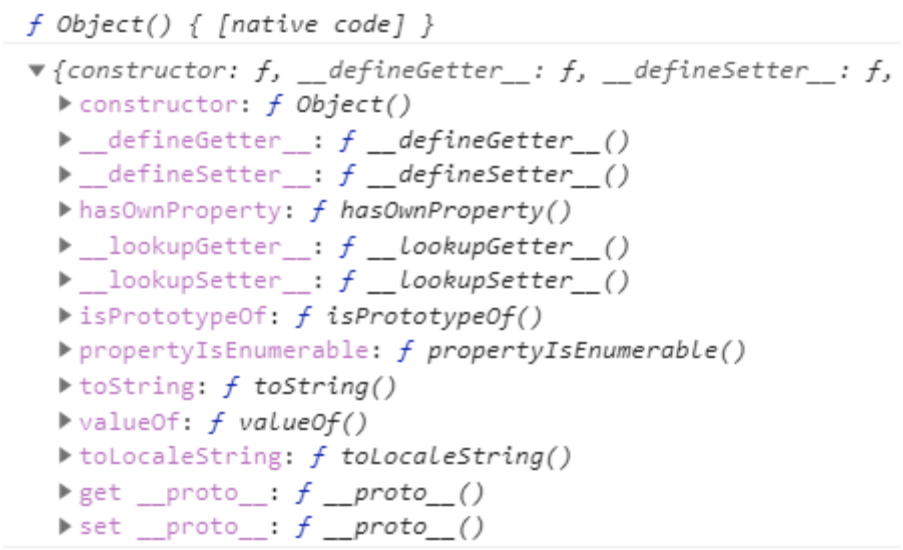
```
var Dog = function(name) {
   this.name = name;

   var barkCount = 0;

   this.bark = function() {
      barkCount++;
      alert(this.name + " bark");
   };
```

```
    this.getBarkCount = function() {
        alert(this.name + " has barked " + barkCount + " times");
    };
};

Dog.prototype.wagTail = function() {
    alert(this.name + " wagging tail");
};

var dog = new Dog("Dave");
dog.bark();
dog.bark();
dog.getBarkCount();
dog.wagTail();
```

## JavaScript prototype

By default, the JavaScript engine provides the `Object()` function and an anonymous object that can be referenced via the `Object.prototype`.

```
console.log(Object);
console.log(Object.prototype);
```




The `Object.prototype` object has many built-in methods and properties such as `toString()`, `valueOf()`, etc.

And it has a property named `constructor` that references the `Object()` function:

```
console.log(Object.prototype.constructor === Object); // true
```

## Is there any benefit of using a function literal to define a "class", over just function definition?

**function Class () {};** → hoisted to the top of the current scope
vs
**var Class = function () {};** → variable declaration is hoisted, but not the assignment

The former is "hoisted" to the top of the current scope before execution. For the latter, the variable declaration is hoisted, but not the assignment.

**For example:**

```
// Error, fn is called before the function is assigned!
fn();
var fn = function () { alert("test!"); }

// Works as expected: the fn2 declaration is hoisted above the call
fn2();
function fn2() { alert("test!"); }
```

## How does prototypal inheritance in JavaScript really works?

JavaScript provides by default a specific case of prototypal inheritance with the new operator.
When accessing the properties of an object, JavaScript will traverse the prototype chain upwards until it finds a property with the requested name.

Most JavaScript implementations use **__proto__** property to represent the next object in the prototype chain. Let see the difference between __proto__ and prototype. Though __proto__ is non-standard and should not be used in your code.

## What is prototype inheritance chain?

When accessing the properties of an object, JavaScript will traverse the prototype chain upwards until it finds a property with the requested name.
Most JavaScript implementations use **__proto__** property to represent the next object in the prototype chain.

### Prototype and Object.getPrototypeOf

Any function in JavaScript has a special property called prototype. This special property works with the JavaScript new operator. The reference to the prototype object is copied to the internal [[Prototype]] property of the new instance. Or the deprecated __proto__ ( same as [[prototype]] internal property.

**For example:**

When you do var a1 = new A(), JavaScript (after creating the object in memory and before running function A() with this defined to it) sets a1.[[Prototype]] = A.prototype. When you then access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in [[Prototype]]. This means that all the stuff you define in prototype is effectively shared by all instances, and you can even later change parts of prototype and have the changes appear in all existing instances, if you wanted to.

```
function A(){
};

A.prototype.doSomething=function(){};
```

If, in the example above, you do var a1 = new A(); then a1.doSomething would actually refer to Object.getPrototypeOf(a1).doSomething, which is the same as the A.prototype.doSomething you defined, i.e. Object.getPrototypeOf(a1).doSomething == Object.getPrototypeOf(a2).doSomething == A.prototype.doSomething.

So, when we call
Function Foo(){}
var o = new Foo();

JavaScript actually just does

var o = new Object();

o.[[Prototype]] = Foo.prototype;

Foo.call(o);

## Object.getPrototypeOf

We have heavily seen one property in JS that is __proro__ ( though it is deprecated, But is still in use & most browser support it.)

It's a quick-and-dirty way of accessing the original prototype property of the object's constructor function.

For example, the following is true:

```
 "test".__proto__ === String.prototype  // true

// Another alternative, not using __proto__
// Only works when constructor isn't changed
"test".constructor.prototype === String.prototype  // true
```

**Note:**

- **__proto__** is non-standard But all the browser except IE supports it and should not be used in our code. It is used here to explain how JavaScript inheritance works.

- Since ECMAScript 2015, the [[Prototype]] is accessed using the accessors Object.getPrototypeOf(). This is equivalent to the JavaScript property __proto__ which is non-standard but de-facto implemented by many browsers.

- It should not be confused with the *func*.prototype property of functions, which instead specifies the [[Prototype]] to be assigned to all *instances* of objects created by the given function when used as a constructor. The **Object.prototype** property represents the Object prototype object.

## Performance

- The lookup time for properties that are high up on the prototype chain can have a negative impact on the performance, and this may be significant in the code where performance is critical.
- Additionally, trying to access nonexistent properties will always traverse the full prototype chain.
- To check whether an object has a property defined on *itself* and not somewhere on its prototype chain, it is necessary to use the hasOwnProperty method which all objects inherit from Object.prototype
- hasOwnProperty is the only thing in JavaScript which deals with properties and does **not** traverse the prototype chain.
- Note: It is **not** enough to check whether a property is undefined. The property might very well exist, but its value just happens to be set to undefined.

## An Example of a Closure

Two one sentence summaries:

- a closure is the local variables for a function - kept alive *after* the function has returned, or
- a closure is a stack-frame which is *not deallocated* when the function returns. (as if a 'stack-frame' were malloc'ed instead of being on the stack!)

The following code returns a reference to a function:

```
function sayHello2(name){
   var text = 'Hello ' + name; // local variable
   var sayAlert = function(){ alert(text); }
   return sayAlert;
}
```

In JavaScript, if you use the function keyword *inside* another function, you are creating a **closure**.

## What is the difference between method in a constructor function vs function's prototype property?

Take a look at Test2 and where foo and bar are located in the chain.

```
var Test2 = function() {                  //foo attached via constructor
   this.foo = 'foo';                      //    each instance has "it's own" foo
}
Test2.prototype.bar = function() {};   //bar attached via prototype
                                       //    all instances "share" the same bar
var mytest2 = new Test2();
```

(3) <-------- (2) <---------- (1) <- prototype chain
Object -> Test2 prototype -> mytest2
'--> bar '--> bar (from prototype)
'--> foo (from constructor)

Basically, anything attached via constructor appears at every instance, but "belongs to that instance". Modifying that property from an instance only modifies it at the current instance.

On the other hand, the ones attached via the prototype is appears on all instances of that object and are "shared". Modifying that property changes this property for all instances.

By attaching a method to a prototype, each instance of the object doesn't have to store the exact same method, thus reducing the memory footprint.

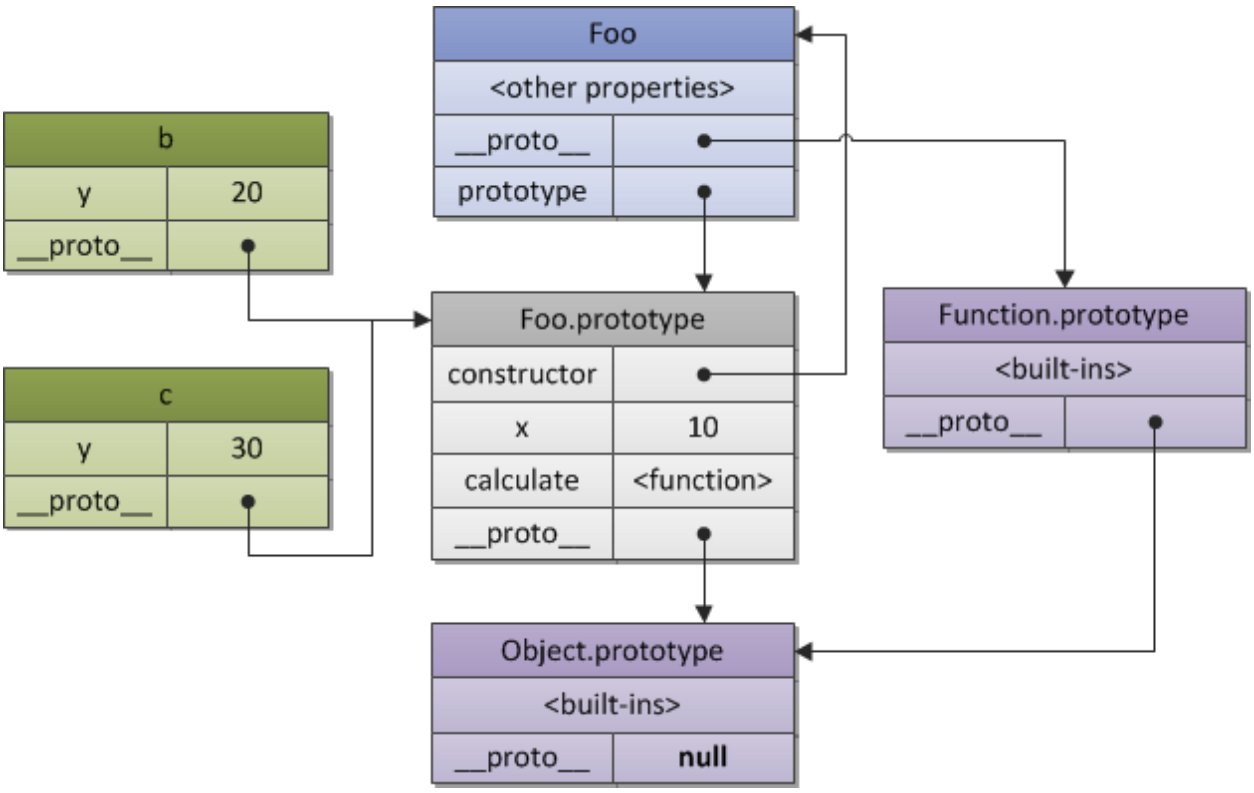
As per Normal OOP and Since methods (behaviors) don't often vary from instance to instance, individual instances don't need to store the same behavior. By putting it on the prototype, all instances will just inherit it.

And, when setting the method on the prototype, do so outside of the constructor function, otherwise each instance will just re-create the same method on the one prototype, causing the last method to be overwritten by the exact same new method.

Example:

```
"use strict";

function Aircraft(a, b, c) {
   this.manufacturer = a;
   this.numberOfEngines = b;
   this.costPerEngine = c;
}

// Prototype modifications should happen outside of the
// constructor function so that they are processed just
// once and not every time the constructor is invoked.
Aircraft.prototype.totalCost = function() {
  return this.numberOfEngines * this.costPerEngine;
}

// At this point, the prototype is all set, so calling the
// constructor (which inherits from the prototype) creates
// a new object instance that has inherited the method.
let thunderbolt = new Aircraft('Republic', 1, 20000);

console.log(thunderbolt.totalCost());
```

## What are the differences between __proto__ and prototype?




**Explanation of above diagram.**

In the above picture from your question, you can see that Foo is a Function Object and therefore it has a __proto__ link to the Function.prototype which in turn is an instance of Object and has a __proto__ link to Object.prototype. The proto link ends here with __proto__ in the Object.prototype pointing to null.

Any object can have access to all the properties in its proto chain as linked by __proto__ , thus forming the basis for prototypal inheritance.
__proto__ is not a standard way of accessing the prototype chain, the standard but similar approach is to use Object.getPrototypeOf(obj).

__proto__ is the actual object that is used in the lookup chain to resolve methods, etc. prototype is the object that is used to build __proto__ when you create an object with new.

prototype is a property of a Function Foo. It is the prototype of objects constructed by that function Foo.

__proto__ is internal property of an object, pointing to its prototype. Current standards provide an equivalent Object.getPrototypeOf(O) method, though de facto standard __proto__ is quicker.

__proto__ is deprecated, but still in use & many popular browser supports it.

Please look at the above diagram & correlates with below program.

```
function Foo()
{
  this.name="Sahu";
};

var b = new Foo();

//! Foo.__proto__
console.log(Foo.__proto__);// [Function]

//! Foo.__proto__.__proto__,as points to Object.prototype
console.log(Foo.__proto__.__proto__);// {}

//! Foo.prototype
console.log(Foo.prototype);// Foo {}

//! Foo.prototype.constructor, as it points back to Foo
console.log(Foo.prototype.constructor);// [Function: Foo]

//! Foo.prototype.__proto__, as points to Object.prototype
console.log(Foo.prototype.__proto__);// {}

console.log(b.__proto__);// Foo {}

console.log(b.__proto__.constructor);// [Function: Foo]

//! points to Object.prototype
console.log(b.__proto__.__proto__);// {}

//! As object.protp.proto points to Object.prototype, hence true
console.log (b.__proto__.__proto__== Object.prototype);
```

## How prototype property of the Function works

To explainit, let us create a function

```
function Foo (name) {
 this.name = name;
}
```

When JavaScript executes this code, it adds prototype property to Foo, prototype property is an object with two properties to it.

1. constructor
2. __proto__

So when we do Foo.prototype, it returns

```
constructor: Foo  // function definition
```

```
__proto__: Object
```

Now as you can see constructor is nothing but the function Foo itself and __proto__ points to the root level Object of JavaScript.

Let us see what happens when we use Foo function with new key word.

```
var b = new Foo ('JavaScript');
```

When JavaScript executes this code it does 4 things:

1. It creates a new object, an empty object // {}
2. It creates __proto__ on b and makes it point to Foo.prototype so b.__proto__ === Foo.prototype
3. It executes Foo.prototype.constructor (which is definition of function a ) with the newly created object (created in step#1) as its context (this), hence the name property passed as 'JavaScript' (which is added to this) gets added to newly created object.
4. It returns newly created object in (created in step#1) so var b gets assigned to newly created object.

Now if we add Foo.prototype.car = "BMW" and do b.car, the output "BMW" appears.
This is because when JavaScript executed this code it searched for car property on b, it did not find then JavaScript used b.__proto__ (which was made to point to ' Foo.prototype' in step#2) and finds car property so return "BMW".

## Prototype VS. __proto__ VS. [[Prototype]]

When creating a function, a property object called *prototype* is being created automatically (you didn't create it yourself) and is being attached to the function object (the constructor).
**Note**: This new *prototype* object also points to, or has an internal-private link to, the native JavaScript Object.

Example:

```
function Foo () {
   this.name = 'John Doe';
}

// Foo has an object property called prototype.
// prototype was created automatically when we declared the function Foo.
Foo.hasOwnProperty('prototype'); // true


var b = new Foo();

//b.[[Prototype]] === Foo.prototype  // true
Object.getPrototypeOf(b) === b.__proto__ // true
```

Note: Object.getPrototypeOf() …> gets it's value from [[prototype]]

The **private** linkage to that function's object called double brackets prototype or just [[Prototype]].
Many browsers are providing us a **public** linkage to it that called __proto__!

To be more specific, __proto__ is actually a getter function that belong to the native JavaScript Object. It returns the internal-private prototype linkage of whatever the this binding is (returns the [[Prototype]] of b):

```
b.__proto__ === Foo.prototype // true
Object.getPrototypeOf(b) === b.__proto__ // true
```

## How 'prototypal inheritance' works?

JavaScript has a mechanism when looking up properties on Objects which is called '**prototypal inheritance**', here is what it basically does:

- First, it's checked if the property is located on the Object itself. If so, this property is returned.
- If the property is not located on the object itself, it will 'climb up the protochain'. It basically looks at the object referred to by the __proto__ property. There, it checks if the property is available on the object referred to by __proto__.
- If the property isn't located on the __proto__ object, it will climb up the __proto__ chain, all the way up to Object object.
- If it cannot find the property anywhere on the object and its prototype chain, it will return undefined.

## Summary:

The __proto__ property of an object is a property that maps to the prototype of the constructor function of the object. In other words:

```
instance.__proto__ === constructor.prototype // true
```

This is used to form the prototype chain of an object. The prototype chain is a lookup mechanism for properties on an object. If an object's property is accessed, JavaScript will first look on the object itself. If the property isn't found there, it will climb all the way up to protochain until it is found (or not)

## Example:

```
function Person (name, city) {
  this.name = name;
}

Person.prototype.age = 25;

const willem = new Person('Willem');

console.log(willem.__proto__ === Person.prototype); // the __proto__ property on the instance refers to the prototype of the constructor

console.log(willem.age); // 25 doesn't find it at willem object but is present at prototype
console.log(willem.__proto__.age); // now we are directly accessing the prototype of the Person function
```

Run code snippet

Expand snippet

Our first log results to true, this is because as mentioned the __proto__ property of the instance created by the constructor refers to the prototype property of the constructor. Remember, in JavaScript, functions are also Objects. Objects can have properties, and a default property of any function is one property named prototype.

Then, when this function is utilized as a constructor function, the object instantiated from it will receive a property called __proto__. And this __proto__ property refers to the prototype property of the constructor function (which by default every function has).

**Why is this useful?**

JavaScript has a mechanism when looking up properties on Objects which is called '**prototypal inheritance**', here is what it basically does:

- First, it's checked if the property is located on the Object itself. If so, this property is returned.
- If the property is not located on the object itself, it will 'climb up the protochain'. It basically looks at the object referred to by the __proto__ property. There, it checks if the property is available on the object referred to by __proto__.
- If the property isn't located on the __proto__ object, it will climb up the __proto__ chain, all the way up to Object object.
- If it cannot find the property anywhere on the object and its prototype chain, it will return undefined.

For example:

```
function Person (name) {
  this.name = name;
}
let mySelf = new Person('Willem');
console.log(mySelf.__proto__ === Person.prototype);
console.log(mySelf.__proto__.__proto__ === Object.prototype);
```

## What is a prototype?

Objects in JavaScript have an internal property, denoted in the specification as[[Prototype]], which is simply a reference to another object. Almost all objects are given a non-nullvalue for this property, at the time of their creation.

prototype is an object automatically created as a special property of a **function**, which is used to establish the delegation (inheritance) chain, aka prototype chain.
When we create a function Foo, prototype is automatically created as a special property on Foo and saves the function code on as the constructor on prototype.

```
function Foo() {};
Foo.prototype // Object {constructor: function}
Foo.prototype.constructor === Foo // true
```

I'd love to consider this property as the place to store the properties (including methods) of a function object. That's also the reason why utility functions in JS are defined
like Array.prototype.forEach() , Function.prototype.bind(), Object.prototype.toString().

## How to get an object's prototype?

via __proto__or Object.getPrototypeOf

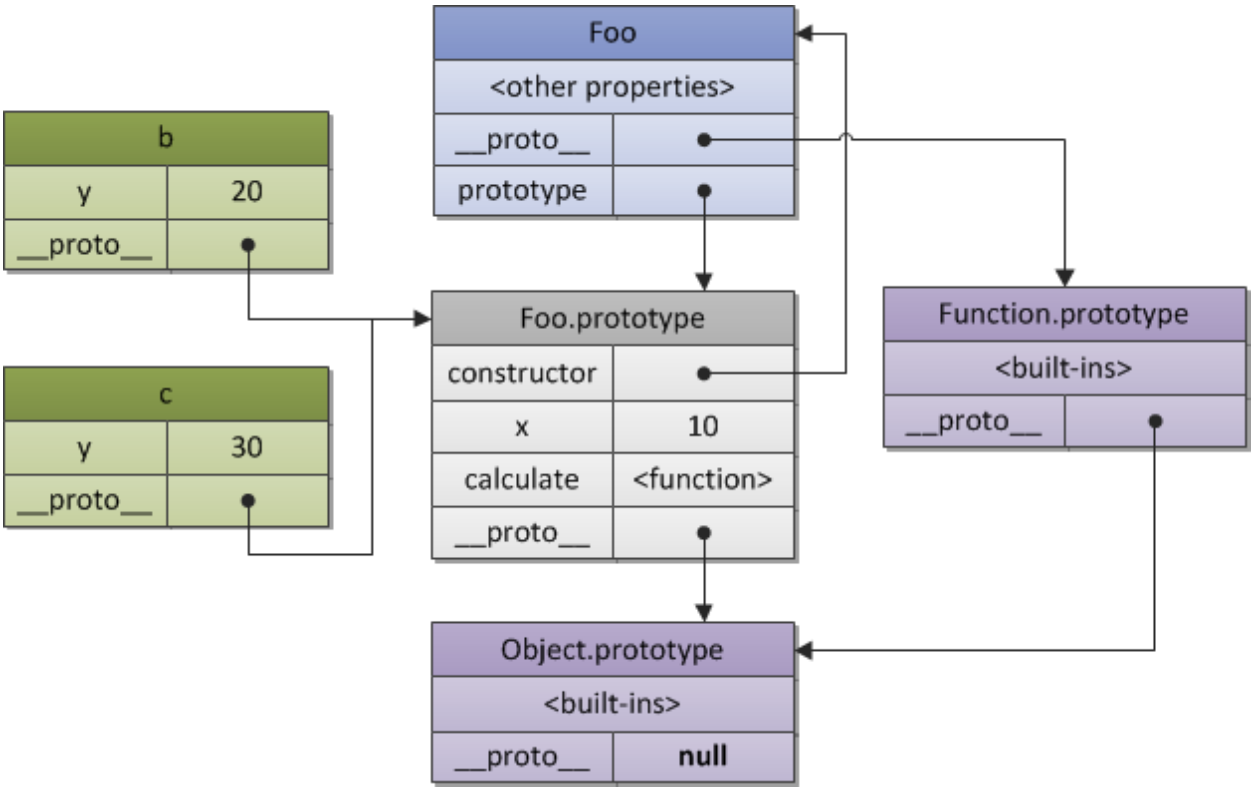
```
var a = { name: "wendi" };

a.__proto__ === Object.prototype // true
```

Object.getPrototypeOf(a) === Object.prototype // true

function Foo() {};
var b = new Foo();
b.__proto__ === Foo.prototype
b.__proto__.__proto__ === Object.prototype

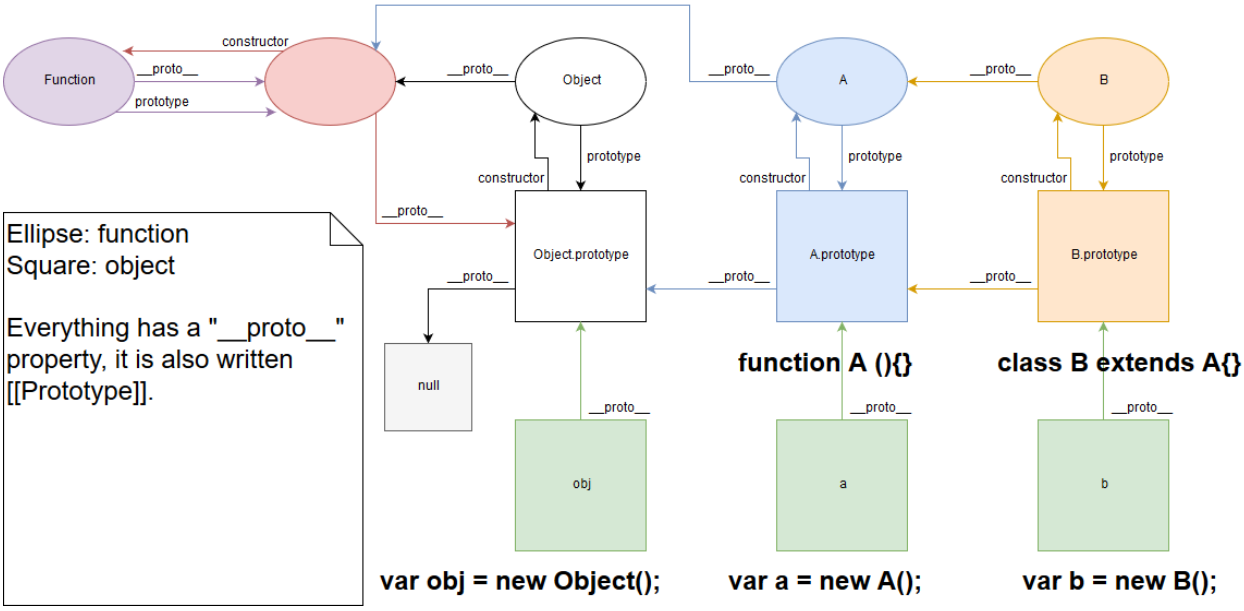
## What's the difference between **__proto__** and **prototype**?

__proto__a reference works on every **object** to refer to its [[Prototype]]property.
prototype is an object automatically created as a special property of a **function**, which is used to store the properties (including methods) of a function object.
With these two, we could mentally map out the prototype chain. Like this picture illustrates:




function Foo() {}
var b = new Foo();

b.__proto__ === Foo.prototype // true
Foo.__proto__ === Function.prototype // true
Function.prototype.__proto__ === Object.prototype // true




In JavaScript, Every object(function is object too!) has a __proto__ property, the property is reference to its prototype.
When we use the new operator with a constructor to create a new object, the new object's __proto__ property will be set with constructor's prototype property, then the constructor will be call by the new object, in that process "this" will be a reference to the new object in the constructor scope, finally return the new object.

Prototype chain actually is object's __proto__ property to reference its prototype, and the prototype's __proto__ property to reference the prototype's prototype, and so on, until to reference Object's prototype's __proto__ property which is reference to null.

For example:

```
console.log(a.constructor === A); // true
// "a" don't have constructor,
// so it reference to A.prototype by its ``__proto__`` property,
// and found constructor is reference to A
```

[[Prototype]] and __proto__ property actually is same thing.
We can use Object's getPrototypeOf method to get something's prototype.

```
console.log(Object.getPrototypeOf(a) === a.__proto__); // true
```

Another good way to understand it:

```
var foo = {}

/*
foo.constructor is Object, so foo.constructor.prototype is actually
Object.prototype; Object.prototype in return is what foo.__proto__ links to.
*/
```

```
console.log(foo.constructor.prototype === foo.__proto__);
// this proves what the above comment proclaims: Both statements evaluate to true.
console.log(foo.__proto__ === Object.prototype);
console.log(foo.constructor.prototype === Object.prototype);
```

## prototype

prototype is a property of a Function. It is the blueprint for creating objects by using that (constructor) function with new keyword.

## __proto__

__proto__ is used in the lookup chain to resolve methods, properties. when an object is created (using constructor function with new keyword), __proto__ is set to (Constructor) Function.prototype

```
function Robot(name) {
   this.name = name;
}
var robot = new Robot();

// the following are true
robot.__proto__ == Robot.prototype
robot.__proto__.__proto__ == Object.prototype
```

## Here is my (imaginary) explanation to clear the confusion:

Imagine, there is an imaginary class (blueprint/coockie cutter) associated with function. That imaginary class is used to instantiate objects. prototype is the extention mechanism to add things to that imaginary class.

```
function Robot(name) {
   this.name = name;
}
```

**The above can be imagined as:**

```
// imaginary class
class Robot extends Object{

   static prototype = Robot.class
   // Robot.prototype is the way to add things to Robot class
   // since Robot extends Object, therefore Robot.prototype.__proto__ == Object.prototype

   var __proto__;

   var name = "";

   // constructor
   function Robot(name) {

      this.__proto__ = prototype;
      prototype = undefined;

      this.name = name;
   }

}
```

**So,**

```
var robot = new Robot();

robot.__proto__ == Robot.prototype
robot.prototype == undefined
robot.__proto__.__proto__ == Object.prototype
```

**Which in turn,**

```
// imaginary class
class Robot{

   static prototype = Robot.class // Robot.prototype way to extend Robot class
   var __proto__;

   var name = "";

   // constructor
   function Robot(name) {

      this.__proto__ = prototype;
      prototype = undefined;

      this.name = name;
   }

   // added by prototype (as like C# extension method)
   function move(x, y){
      Robot.position.x = x; Robot.position.y = y
   };
}
```

So, **Prototype or Object.prototype** is a property of an object literal. It represents the *Object* prototype object which you can override to add more properties or methods further along the prototype chain.
**__proto__** is an accessor property (get and set function) that exposes the internal prototype of an object thru which it is accessed.

## [[Prototype]] :

[[Prototype]] is an internal hidden property of objects in JS and it is a reference to another object. Every object at the time of creation receives a non-null value for [[Prototype]]. Remember [[Get]] operation is invoked when we reference a property on an object like, myObject.a. If the object itself has a property, a on it then that property will be used.

```
let myObject= {
   a: 2
};

console.log(myObject.a);          // 2
```

But if the object itself directly does not have the requested property then [[Get]] operation will proceed to follow the [[Prototype]] link of the object. This process will continue until either a matching property name is found or the [[Prototype]] chain ends(at the built-in Object.prototype). If no matching property is found then **undefined** will be returned.

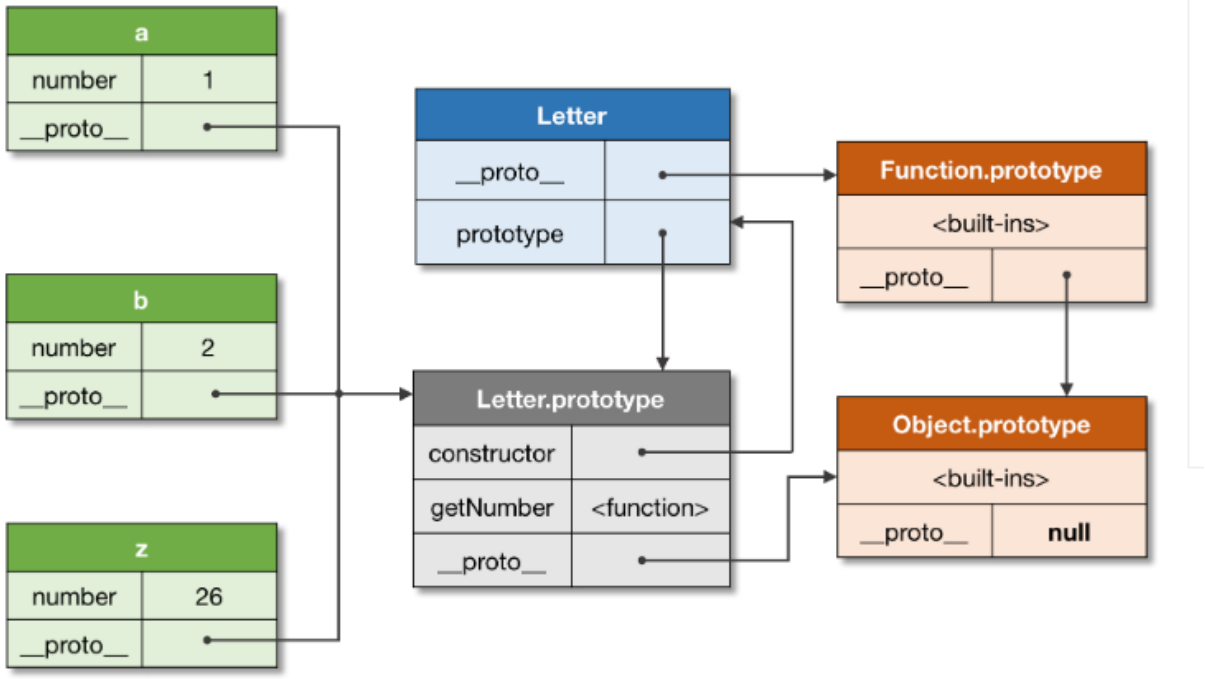
Both for..in loop and in operator use [[Prototype]] chain lookup process. So if we use for..in loop to iterate over the properties of an object then all the enumerable properties which can be reached via that object's [[Prototype]] chain also will be enumerated along with the enumerable properties of the object itself. And when using in operator to test for the existence of a property on an object then in operator will check all the properties via [[Prototype]] linkage of the object regardless of their enumerability.

```
// for..in loop uses [[Prototype]] chain lookup process
let anotherObject= {
   a: 2
};

let myObject= Object.create(anotherObject);

for(let k in myObject) {
   console.log("found: " + k);           // found: a
}

// in operator uses [[Prototype]] chain lookup process
console.log("a" in myObject);              // true
```

## .prototype :

.prototype is a property of functions in JS and it refers to an object having constructor property which stores all the properties(and methods) of the function object.

```
let foo= function(){}

console.log(foo.prototype);
// returns {constructor: f} object which now contains all the default properties

foo.id= "Walter White";

foo.job= "teacher";

console.log(foo.prototype);
// returns {constructor: f} object which now contains all the default properties and 2 more properties that we added
to the fn object
/*
{constructor: f}
   constructor: f()
      id: "Walter White"
      job: "teacher"
      arguments: null
      caller: null
      length: 0
      name: "foo"
      prototype: {constructor: f}
      __proto__: f()
      [[FunctionLocation]]: VM789:1
      [[Scopes]]: Scopes[2]
   __proto__: Object
*/
```




To retrieve the value of obj.__proto__ is like calling, obj.__proto__() which actually returns the calling of the getter fn, Object.getPrototypeOf(obj) which exists on Object.prototype object.
Although .__proto__ is a settable property but we should not change [[Prototype]] of an already existing object because of performance issues.

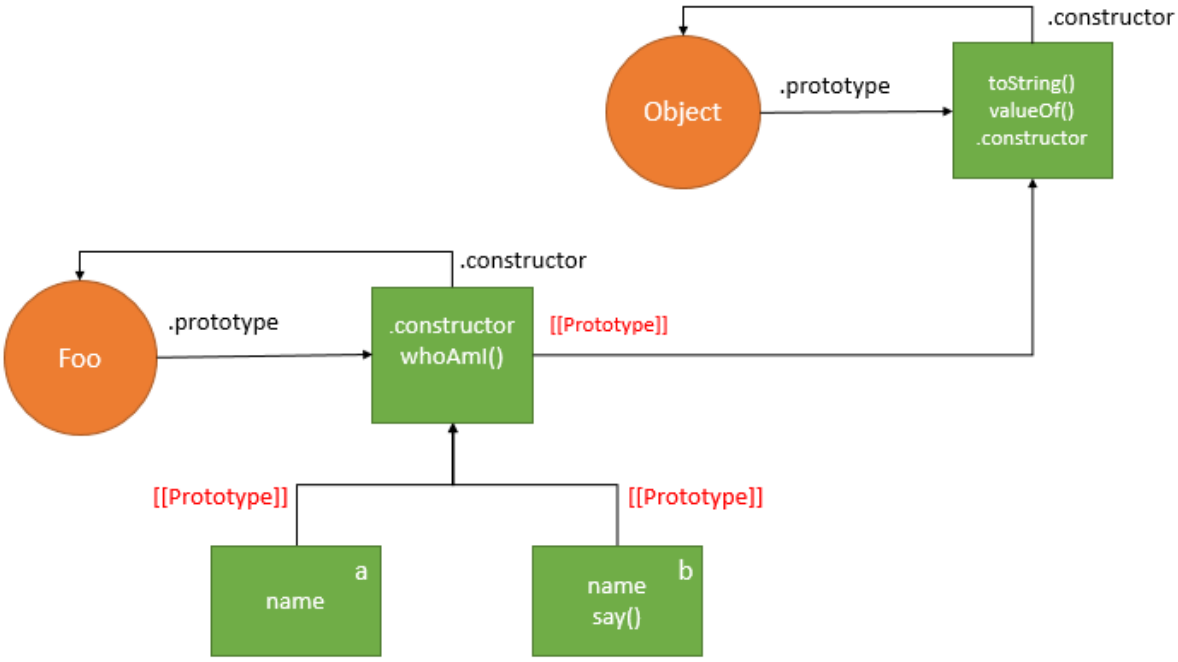
Using new operator if we create objects from a function then internal hidden [[Prototype]] property of those newly created objects will point to the object referenced by the .prototype property of the original function. Using .__proto__ property we can access the other object referenced by internal hidden [[Prototype]] property of the object. But __proto__ is not the same as [[Prototype]] rather a getter/setter for it. Consider below code :

```
let Letter= function() {}

let a= new Letter();

let b= new Letter();

let z= new Letter();

// output in console
a.__proto__ === Letter.prototype;            // true

b.__proto__ === Letter.prototype;            // true

z.__proto__ === Letter.prototype;            // true

Letter.__proto__ === Function.prototype;     // true
```

```
Function.prototype.__proto__ === Object.prototype;        // true

Letter.prototype.__proto__ === Object.prototype;          // true
```

**Link:** https://hackernoon.com/understand-nodejs-javascript-object-inheritance-proto-prototype-class-9bd951700b29

```
function Foo(name) {
  this.name = name;
}
var b = new Foo('b');
var a = new Foo('a');
b.say = function() {
  console.log('Hi from ' + this.whoAmI());
}
```




## What is the difference between "__proto__" and "prototype"?

Short answer:

- `__proto__` is the actual prototype, but don't use it.
- `.constructor.prototype` was supposed to do the same thing as `__proto__` but its mostly broken.
- A function's `.prototype` is actually the prototype of things made by it, not its prototype.

## Functions

- Every **function declaration** immediately creates **TWO OBJECTS**:

    - the function object itself
    - the prototype object, owned by this function
    - That happens **before** any code execution even begins, just after code parsing
- function object can be accessed just using function name without parenthesis, for example: myFunction
- prototype object can be accessed using prototype property of function object, for example: myFunction.prototype
- prototype object is used by JavaScript, when function is invoked as a constructor (with new keyword) to initialize newly constructed object __proto__ property
- prototype object of constructor function is reminiscent of what is usually stored in class definition, in classical OOP languages like Java and C++
- constructor function and its prototype object are always come togehter
- prototype object does not used at all, if function is not intended to be used as a constructor

## Objects

- Every object has a built-in __proto__ property
- __proto__ property correspond to internal, hidden [[Prototype]] property of the object
- function object and its prototype object, **both**, also have __proto__ property
- __proto__ property as an accessor, standardized only in ES6. In ES5, existance of __proto__ property depends on implementation. In ES5 standard way to access value of [[Prototype]] property is Object.getPrototypeOf() method
- In ES6 __proto__ property can be set, it just holds reference to another object. In ES6 there is also a Object.setPrototypeOf() method
- It is possible to create object with __proto__ property set to null using var obj = Object.create(null)
- Object, which is referenced by __proto__ property of a given object, is called its parent. That parent object can also have __proto__ property to its own parent, thus forming prototype chain
- prototype chain of objects or prototypal inheritance chain is a way, how **inheritance** is implemented in JavaScript
- When JavaScript runtime looks for a property, with a given name, on an object, it first examines object itself, and then all objects down its prototype chain

## Built-in constructor functions

This is a list of most popular JavaScript built-in constructors. They are constructors, not just functions, objects or namespaces - this is **important!**

- Array
- Boolean
- Date
- Error, and its derrivatives
- Function
- Math
- Number
- Object
- RegExp

- String

Meanings of term "object":

- Built-in Object constructor
- Specific JavaScript object, referenced by some access path
- A constructor with its prototype
    - The problem here in the fact, that we have no classes, and JavaScript developers often call it object but with meaning type. Better not to use terms class, type, object, object type, but use only terms constructor or constructor function
- JSON object
    - JSON stands for "JavaScript Object Notation"".
    - Typical misuse and misunderstanding is, that JSON is not an object, it is always a string, which will become an object in the memory only after parsing
- POJO, which stands for "Plain Old Javascript Object", or just "simple object"
    - This is an object without any custom prototype chain, or any added "methods", just a container for data
    - Its __proto__ property refers directly to Object.prototype, or equal to null
    - Can be considered as a Hash

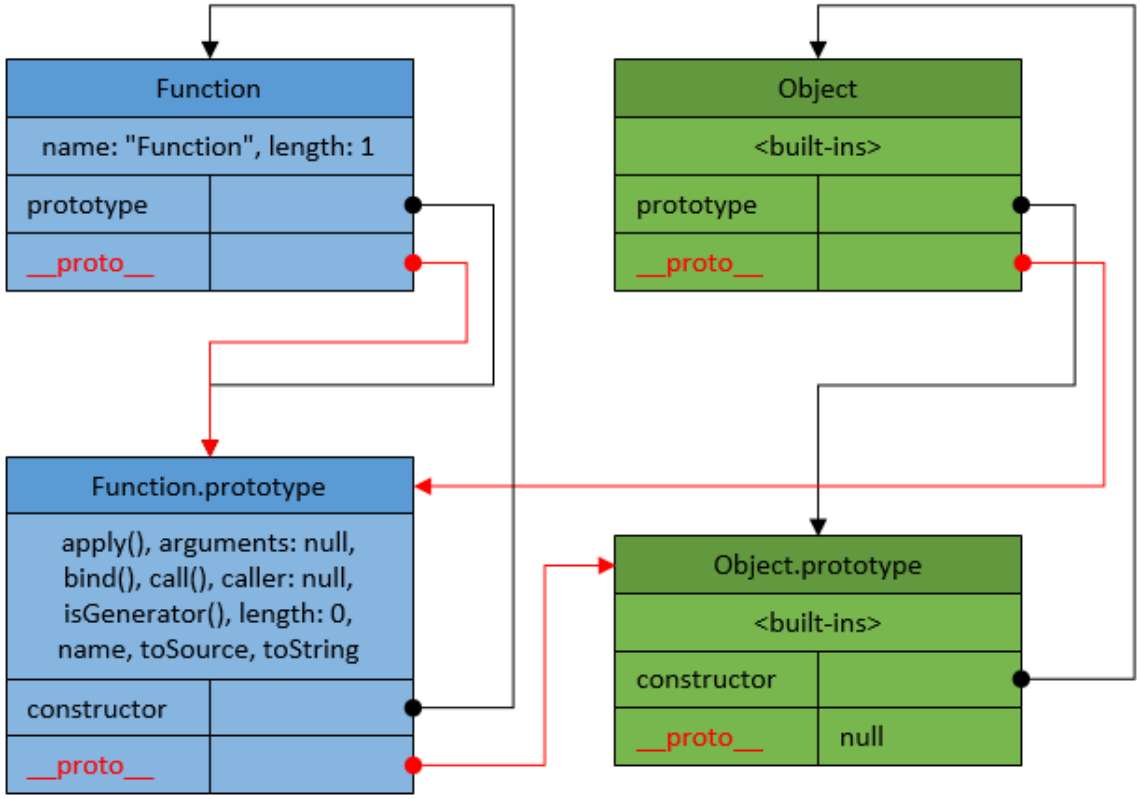
Meanings of term "function":

- Built-in Function constructor
- Specific named JavaScript function, referenced by name created with function declaration
- Specific anonymous JavaScript function, referenced by some access path

## Function and Object constructors relation

Relation between Function and Object constructors is very important. It plays major role in JavaScript inheritance.
To summarize:

- Every function in JS is an object, more exactly - two objects: function itself and its prototype
- There are two distinct built-in constructor functions Function and Object related to each-other
- Function.prototype object itself inherits from Object.prototype
- Every function in the system inherits from Function.prototype
- Every object in the system inherits from Object.prototype




The prototypal inheritance chain is drawn in red.

As you may see Function and Object are both functions, thus, they both have prototype property, which holds reference to respective prototype object.
Function and Object are both functions, thus their __proto__ property, refers to Function.prototype, which itself has __proto__ property referencing to Object.prototype.
Both prototype and __proto__ properties of a Function refer to the same Function.prototype object, which is a unique situation, valid only for built-in Function constructor.

## Meanings of term "prototype":

- A "prototype" of a given object
  - Its parent
  - Accessible with someObject.__proto__ property, not prototype property
  - The most confusing part is:
    - Parent of a someObject object, referenced by someObject.__proto__ property, is commonly called its **prototype**
- A prototype object of a given function, especially a constructor function
  - Accessible with SomeConstructor.prototype property
  - That should be called **constructor prototype**
- A built-in Function.prototype object
- A built-in Object.prototype object
- A prototype of any other built-in constructor, for example String.prototype

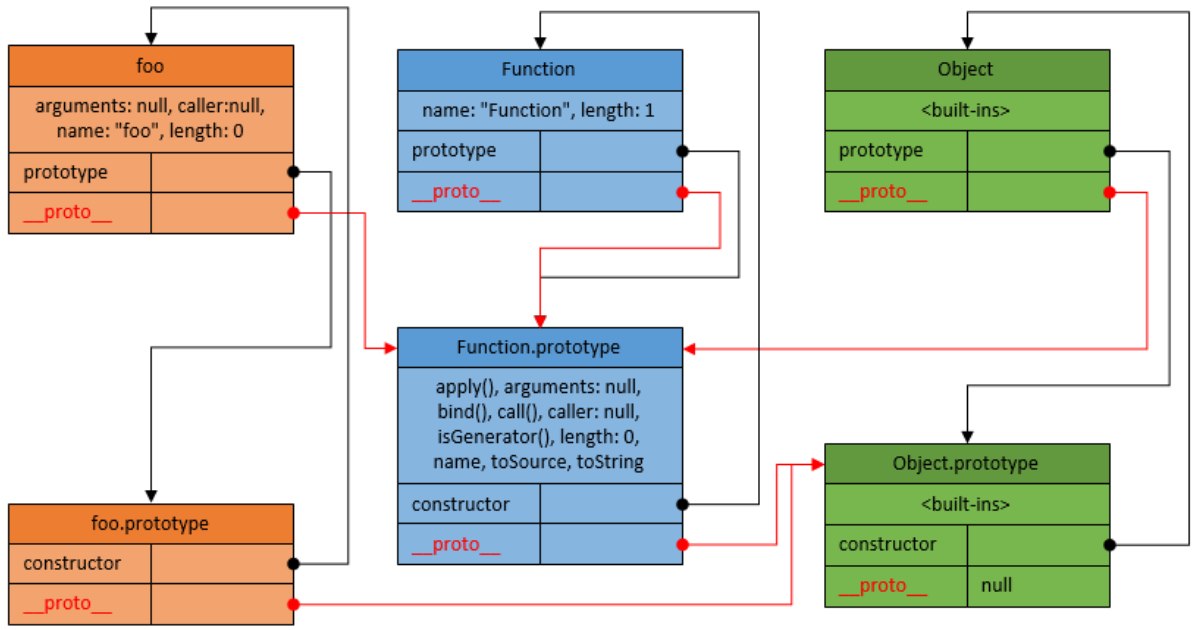
To summarize:

- Only a function may have prototype property
- Any function has prototype property
- prototype property of a function holds reference to an auxiliary object, which is used only, when the function is invoked as a constructor, with new keyword, and completely ignored for all other regular functions
- Any object has prototype chain
- **Prototype chain is built using __proto__ property, not prototype property**
- Functions are also objects, and thus have __proto__ property, referencing directly to Function.prototype built-in object. Their prototype chain is always someFunction -> Function.prototype -> Object.prototype
- All prototype chains ends with Object.prototype
- Object.prototype.__proto__ holds null. This is real end of prototype chain

## Function in JavaScript

Having simple function declaration, we get following inheritance.

```
function foo(){}
```




What we can see:

- function declaration creates two objects: foo itself and foo.prototype, even if foo does not going to be used as a constructor

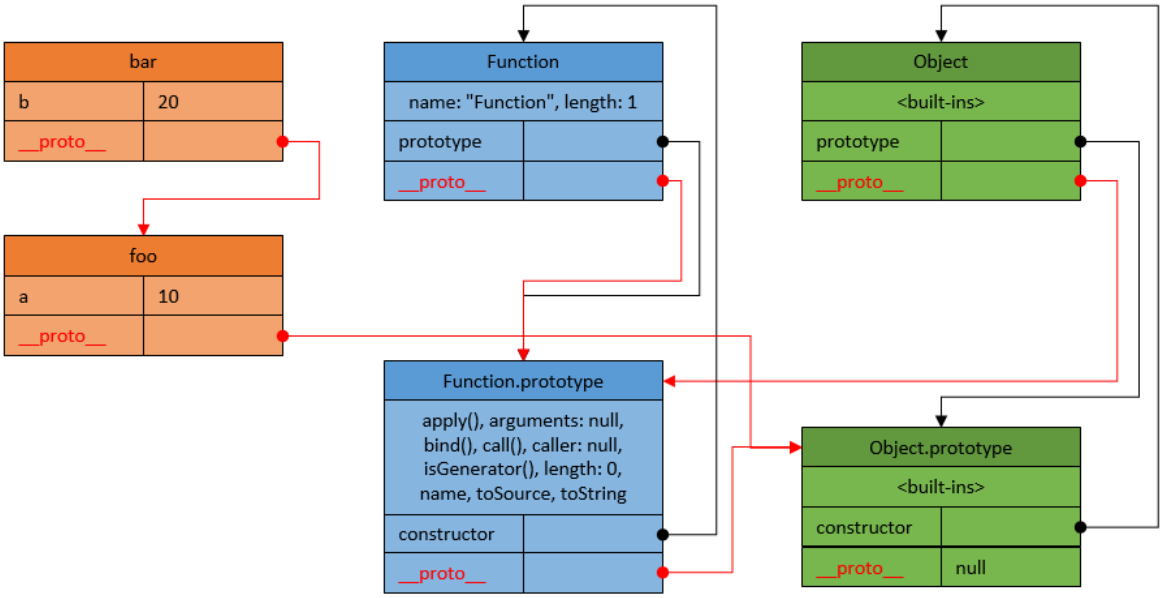
- foo inherits directly from Function.prototype
- foo.prototype inherits from Object.prototype
- this inheritance valid for any, even anonymous or arrow function.

What we don't see is that foo itself has internal [[Code]] property, which cannot be accessed but is used when we invoke it with foo().
When you use foo.someMethod(), all built-in methods come from Function.prototype and down the chain from Object.prototype. But foo.someMethod() never comes from foo.prototype. See Static methods.
foo.prototype typically does not used at all, if function is not a constructor, and vice versa, is used in the case of a constructor function.
foo.prototype can be set to any other object reference or primitive value. Setting it to a new object or define new properties on it is a common pattern to define a class.
ES6 arrow function do not have prototype object created by default. It cannot be used as a constructor because it has lexical this and cannot initialize new instance properties.

## Creating simple objects with inheritance

Simple objects in JavaScript created with object literal syntax or with Object.create function.

```
// Simple object literal
var foo = {
  a: 10
};

// foo object will be used as prototype for bar
var bar = Object.create(foo, {
    b: {
      value: 20
    }
  });

console.log(bar.__proto__ === foo); //true
console.log(bar.a); // 10 - Inherited property
console.log(bar.b); //20 - Own porperty
console.log(foo instanceof Object); // true, foo inherits from Object
console.log(bar instanceof Object); // true, bar inherits from Object
```



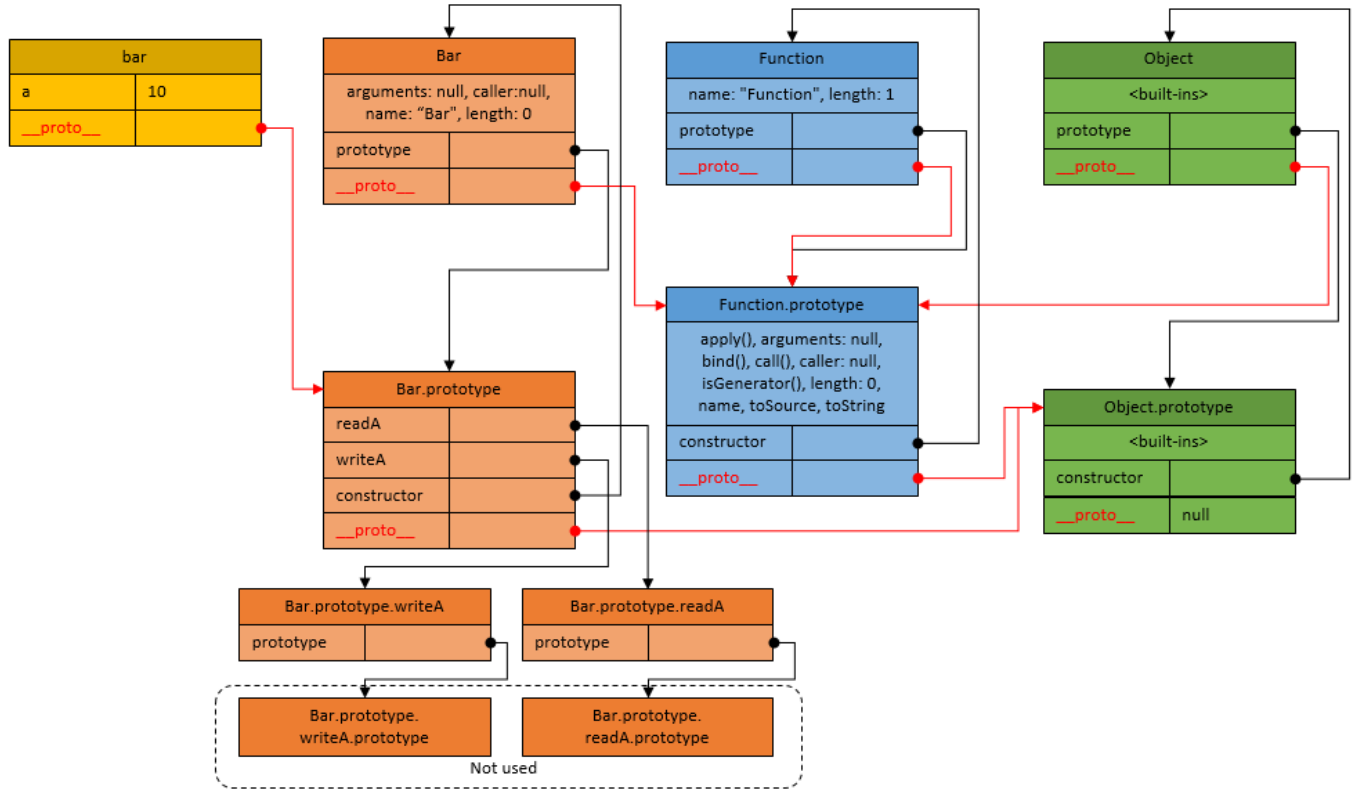

Important moment here is that in case of changing bar.a value, JavaScript automatically creates bar.a own property with new value, to prevent prototype pollution.
Even if prototype chain of foo and bar looks very simplistic, we can note, that both have an inherited constructor property, which points to the Object constructor, which itself inherits from Function.prototype. More of that, there are a lot of built-in methods in Object.prototype itself, not displayed for clarity. They all are accessible on foo and bar.

## Creating an object with constructor function

Now, let's declare a simple constructor function and create an object instance using it.

```
function Bar() {
  // "this" will point to newly created object
  this.a = 10;
}

Bar.prototype.readA = function () {
  // "this" will point to the object, in context of which, method is invoked
  return this.a;
}

Bar.prototype.writeA = function (a) {
  this.a = a;
}

// Constructor function invocation requires "new" keyword
var bar = new Bar();

console.log(bar.constructor === Bar); // true, "constructor" - inherited property
console.log(bar instanceof Bar); // true
console.log(bar instanceof Object); // true, bar inherits from Object
console.log(bar.readA()); // 10 - Invoking inherited method in context of "bar" object
bar.writeA(20);
```

```
console.log(bar.readA()); // 20
console.log(bar.a); // 20 - Reading own porperty of "bar" object
```



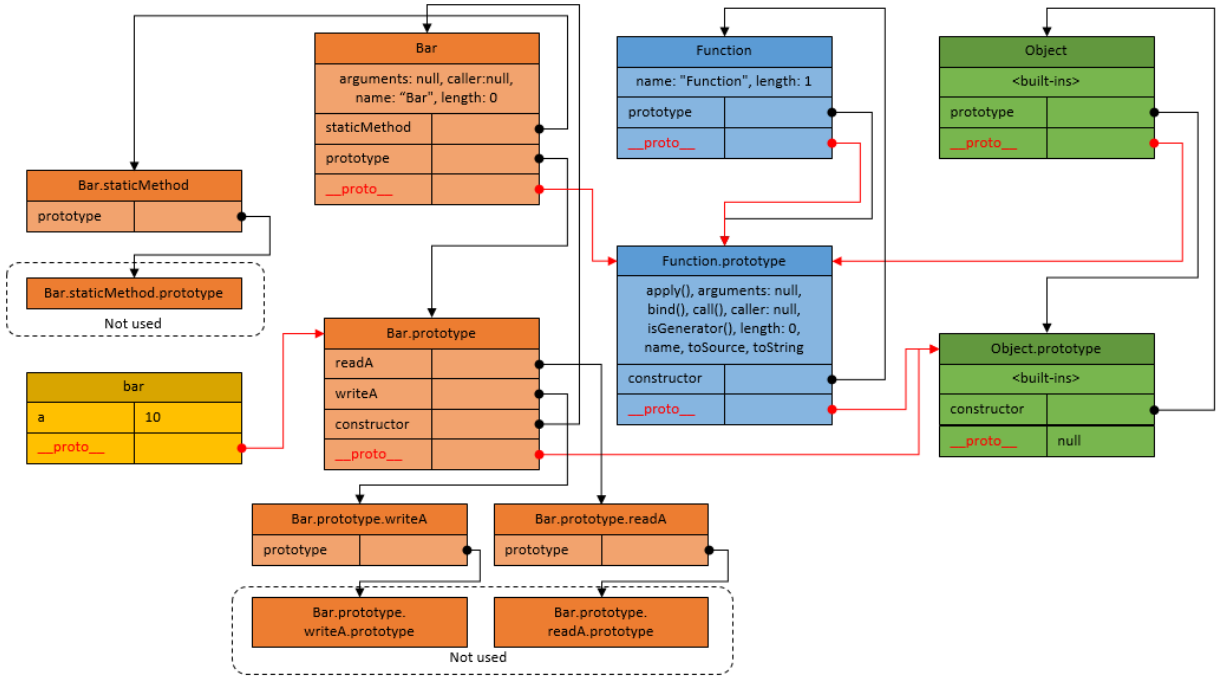

readA and writeA are just regular JS functions with similar references to Function and Object as Bar function itself. These references are not shown for clarity. Important difference between them and Bar function is, that their prototypes are not of any use.
bar object has its own property a, because this property created every time Bar constructor is invoked. This behavior allows to produce different objects with their own property a, but inheriting "methods" from Bar.prototype.

## Static methods

There is no such thing like static method in JavaScript spec at all, but this design pattern can easily be implemented by putting properties on a constructor function object itself, instead of its prototype object.

```
function Bar() {
  this.a = 10;
}

Bar.staticMethod = function () {
  // can not use "this" here
  return "I am static";
}

Bar.prototype.readA = function () {
```

```
  return this.a;
}

Bar.prototype.writeA = function (a) {
  this.a = a;
}
var bar = new Bar();

console.log(bar.staticMethod); // undefined, method can not be invoked on instance
console.log(Bar.staticMethod()); // "I am static"
console.log(bar.constructor.staticMethod()); // "I am static", method available on instance through inherited
constructor property
```



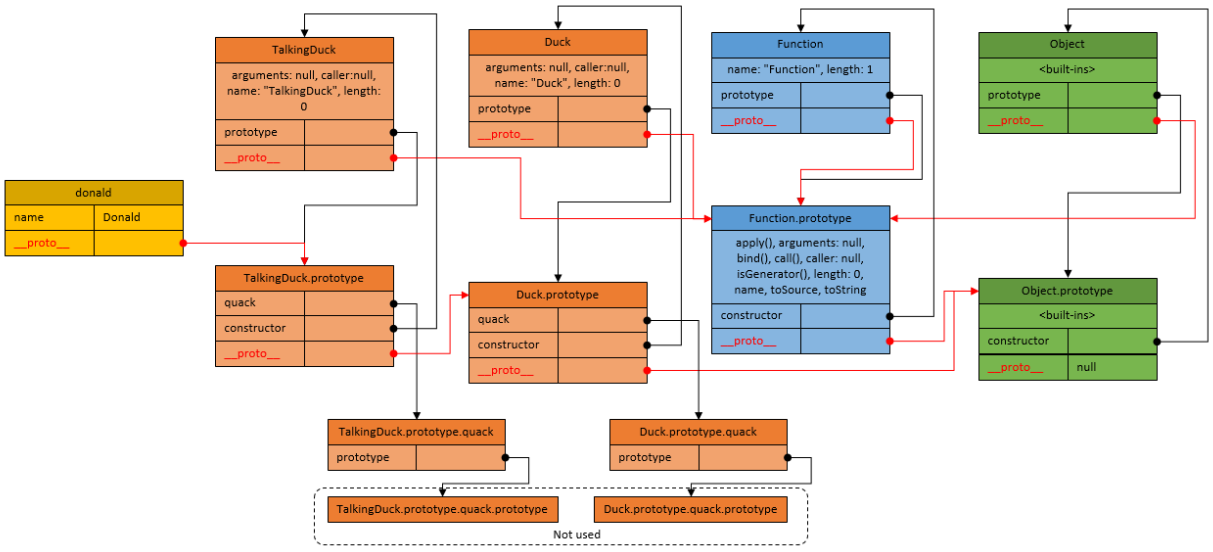

Static methods of constructor are not accessible on instances, created with this consrtuctor, they are available on a constructor itself.

A lot of useful design patterns in JavaScript are implemented putting methods on a constructor function object, for example factory functions. Such a constructor can be used as a namespace or singleton.

## Classical JavaScript inheritance and OOP

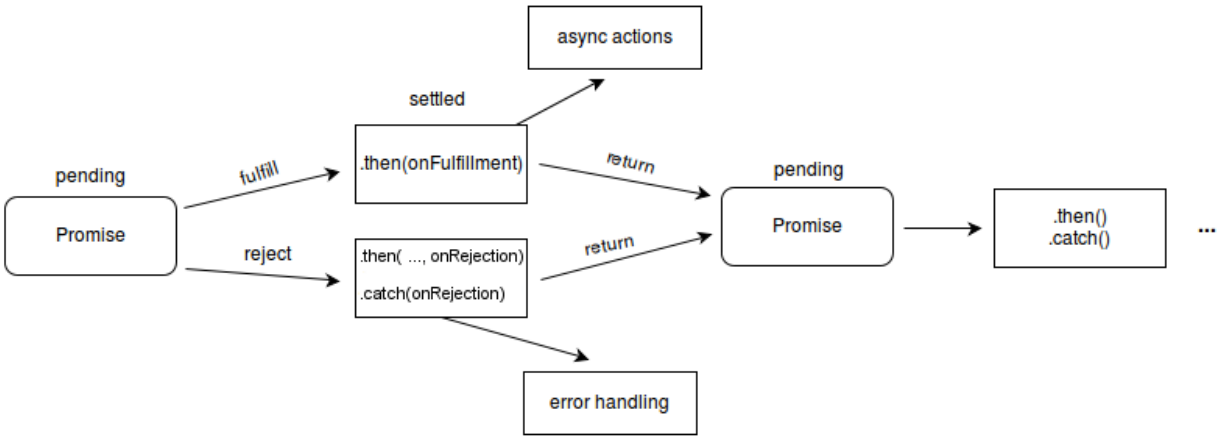
```
// Parent constructor
function Duck (name) {
  this.name = name;
};

// Parent method
Duck.prototype.quack = function () {
```

```
  return this.name + " Duck: Quack-quack!";
};

// Child constructor
function TalkingDuck (name) {
  // Call parent constructor with proper arguments
  Duck.call(this, name); // This is often forgotten
}

// Inheritance
TalkingDuck.prototype = Object.create(Duck.prototype);
TalkingDuck.prototype.constructor = TalkingDuck; // This is often forgotten

// Method overload
TalkingDuck.prototype.quack = function () {
  // Call parent method
  return Duck.prototype.quack.call(this) + " My name is " + this.name;
};

// Object instantiation
var donald = new TalkingDuck("Donald");
console.log(donald.quack()); // "Donald Duck: Quack-quack! My name is Donald"

console.log(donald.__proto__ === TalkingDuck.prototype); // true
console.log(TalkingDuck.prototype.__proto__ === Duck.prototype); // true, result of invoking Object.create()
console.log(Duck.prototype.__proto__ === Object.prototype); // true
console.log(donald.quack === TalkingDuck.prototype.quack); // true, method found by prototype chain
console.log(donald instanceof TalkingDuck); // true
console.log(donald instanceof Duck); // true
console.log(donald.name); // "Donald", reading own property
```



Prototype chain of donald is donald -> TalkingDuck.prototype -> Duck.prototype -> Object.prototype. Function.prototype does not taking part in this chain, since donald is not a function. Note that name is own property of donald, though it is created with Duck constructor. This is because Duck constructor is invoked with Duck.call(this, name), where this points to newly created object inside TalkingDuck constructor and then passed down as an invocation context to Duck constructor. See [MDN call() reference]

## Callbacks, Promises, and Async:




```
const promiseA = new Promise( (rejectionFunc,resolutionFunc) => {
   rejectionFunc("DONE");
});
// At this point, "promiseA" is already settled.
promiseA.then( (str) => console.log("rejectionFunc : asynchronous logging has val:", str) ).catch(
  (str) => console.log("resolutionFunc: asynchronous logging has val:", str)
);
console.log("immediate logging");
```

Here: First function will call first one & second will call second one.

In JavaScript, **Promises** are objects represent the eventual outcome of an asynchronous action. At a time, a promise can be one of 3 states:

- Pending: This is an initial state, the operation has not yet completed, neither fulfill or rejected.
- Fulfill: the operation has been completed successfully, the promise now has the resolved value.
- Rejected: the operation has been failed due to some kind of errors.

**Syntax**

```
var promise = new Promise(function(resolve, reject){
    //do something
});
```

**Parameters**

- Promise constructor takes only one argument, a callback function.

- Callback function takes two arguments, *resolve* and *reject*
- Perform operations inside the callback function and if everything went well then call resolve.
- If desired operations do not go well then call reject.

```
var promise = new Promise(function(resolve, reject) {
  const x = "geeksforgeeks";
  const y = "geeksforgeeks"
  if(x === y) {
    resolve();
  } else {
    reject();
  }
});

promise.
    then(function () {
        console.log('Success, You are a GEEK');
    }).
    catch(function () {
        console.log('Some error has occured');
    });
```

**Output:**

Success, You are a GEEK

## Promise Consumers

Promises can be consumed by registering functions using *.then* and *.catch* methods.

1. **then()**
   *then()* is invoked when a promise is either resolved or rejected.
   **Parameters:**
   *then()* method takes two functions as parameters.
   2. First function is executed if promise is resolved and a result is received.
   3. Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to hanlde error using *.catch() method*

**Syntax:**

```
.then(function(result){
        //handle success
    }, function(error){
        //handle error
    })
```

**Example: Promise Resolved**

```
var promise = new Promise(function(resolve, reject) {
    resolve('Geeks For Geeks');
})

promise
    .then(function(successMessage) {
      //success handler function is invoked
```

```
    console.log(successMessage);
  }, function(errorMessage) {
    console.log(errorMessage);
  })
```

**Output:**

Geeks For Geeks

**Examples: Promise Rejected**

```
var promise = new Promise(function(resolve, reject) {
  reject('Promise Rejected')
})

promise
  .then(function(successMessage) {
    console.log(successMessage);
  }, function(errorMessage) {
    //error handler function is invoked
    console.log(errorMessage);
  })
```

**Output:**

Promise Rejected

**catch()**

*catch()* is invoked when a promise is either rejected or some error has occured in execution.

**Parameters:**

*catch()* method takes one function as parameter.

1. Function to handle errors or promise rejections.(.catch() method internally calls .then(null, errorHandler), i.e. .catch() is just a shorthand for .then(null, errorHandler) )

**Examples: Promise Rejected**

```
var promise = new Promise(function(resolve, reject) {
  throw new Error('Some error has occured')
})

promise
  .then(function(successMessage) {
    console.log(successMessage);
  })
  .catch(function(errorMessage) {
    //error handler function is invoked
    console.log(errorMessage);
  });
```

Output:

Error: Some error has occurred

**Applications**

1. Promises are used for asynchronous handling of events.
2. Promises are used to handle asynchronous http requests.

**Reference**: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

The .then() method

The initial state of a promise is pending, but it is guaranteed to be settled in the future (either resolved or reject). What should we do next? We can use the `.then()` method to perform some actions afterward.

The `.then()` is a method taking 2 callback functions as arguments `onFulfilled` (*success handler*) and `onRejected`(*failed handler*). The `onFulfilled` function is invoked when the promise is fulfilled. If the promise is rejected, the `onRejected` function will be called. And this method always returns a **promise**.

```
promiseObject.then(onFulfilled, onRejected);
```

There is no obligation that we must pass all two methods `onFulfilled` or `onRejected` to this method, we can pass them both, either, or we can omit both functions. This allows for flexibility, but it can be trickier to debug. If in case there is no appropriate handler, the `.then()` method just returns the value as the settled value in the promise it was called.

**Example:**

```
function sweepTheFloorPromise(completed) {
   return new Promise(function(resolve, reject) {
      setTimeout(() => {
         if (completed) {
            resolve("I have done with sweeping, now the floor is clean!")
         } else {
            reject(Error("There is no broom!"));
         }
      }, 3000);
   });
}
var sweepTheFloor = sweepTheFloorPromise(true);
sweepTheFloor.then(
   success, failingReason
);

function success(message) {
   console.log(message);
}

function failingReason(reason) {
   console.log(reason);
}
```

Since the promise inside the `sweepTheFloorPromise` is resolved, hence the success handler function `sucess` will be **invoked** with **this promise's resolved value** (again, the promise inside the `sweepTheFloorPromise` function).

It also possible to either schedule a callback to handle the fulfilled or rejected case only, if we just want to run the fulfilled case:

```
sweepTheFloor.then(success);

Or we only want the rejected case:
sweepTheFloor.then(
   undefined, failingReason
);
// or
sweepTheFloor.then(
   null, failingReason
);
```

Instead of passing both handlers into one then, we can chain a second `.then()` with a failure handler to a first `.then()` with a success handler and both cases will be handled.

```
sweepTheFloor
   .then(success)
   .then(null, failingReason)
```

## The .catch() method

The `catch()` method accepts only one argument which is `onRejected` value. In the case of a rejected promise, this failure handler will be invoked with the reason for rejection.
Using `catch()` accomplishes the same thing as `then()` with only failed handlers:

```
sweepTheFloor
   .then(success)
   .catch(failingReason)
```

## The .finally() method

Sometimes you don't care whether a promise is fulfilled or rejected and want to execute a same piece of code, we can use the `.finally()` method, this method also takes a callback function as a parameter and this method also returns a promise:

```
sweepTheFloor
   .then(success)
   .catch(failingReason)
   .finally(doSomething)
```

## Async and Await:

An async function is a modification to the syntax used in writing promises. You can call it syntactic sugar over promises. It only makes writing promises easier.

An async function returns a promise -- if the function returns a value, the promise will be resolved with the value, but if the async function throws an error, the promise is rejected with that value. Let's see an async function:

```
async function myRide() {
  return '2017 Dodge Charger';
}
```

and a different function that does the same thing but in promise format:

```
function yourRide() {
  return Promise.resolve('2017 Dodge Charger');
}
```

From the above statements, `myRide()` and `yourRide()` are *equal* and will both resolve to `2017 Dodge Charger`. Also when a promise is rejected, an async function is represented like this:

```
function foo() {
  return Promise.reject(25)
}

// is equal to
async function() {
  throw 25;
}
```

## Await

Await is only used with an async function. The await keyword is used in an async function to ensure that all promises returned in the async function are synchronized, ie. they wait for each other. Await eliminates the use of callbacks in `.then()` and `.catch()`. In using async and await, async is prepended when returning a promise, await is prepended when calling a promise. `try` and `catch` are also used to get the rejection value of an async function. Let's see this with our date example:

```
async function myDate() {
  try {

    let dateDetails = await date;
    let message     = await orderUber(dateDetails);
    console.log(message);

  } catch(error) {
    console.log(error.message);
  }
}
```

Lastly we call our async function:

```
(async () => {
  await myDate();
```

```
})();
```

## Callback function:

When a function simply accepts another function as an argument, this contained function is known as a callback function. Using callback functions is a core functional programming concept, and you can find them in most JavaScript code; either in simple functions like setInterval, event listening or when making API calls.

Callback functions are written like below:

```
setInterval(function() {
  console.log('hello!');
}, 1000);
```

setInterval accepts a callback function as its first parameter and also a time interval. Another example using .map();

```
const list    = ['man', 'woman', 'child']

// create a new array
// loop over the array and map the data to new content
const newList = list.map(function(val) {
  return val + " kind";
});

// newList = ['man kind', 'woman kind', 'child kind']
```

In the example above, we used the .map() method to iterate through the array list, the method accepts a callback function which states how each element of the array will be manipulated. Callback functions can also accept arguments as well.

```
function greeting(name)
{
console.log(name);
}
function introduction(firstName, lastName, callback) {
  const fullName = `${firstName} ${lastName}`;

  callback(fullName);
}
introduction('Haramohan ','Sahu', greeting); // Hello Haramohan Sahu, welcome to Scotch!
```

## Key difference between callbacks and promises

A key difference between the two is that when using the callbacks approach we would normally *just* pass a callback *into a function* which will get called upon completion to get the result of something, whereas in promises you attach callbacks *on the returned promise object*.

# Callbacks:

```
function greeting(name)
{
console.log(name);
}


function getMoneyBack(money, callback) {
  if (typeof money !== 'number') {
    callback(null, new Error('money is not a number'))
  } else {
    callback(money)
  }
}

getMoneyBack(1200, greeting);
```

## JavaScript call() method

In JavaScript, a function is an instance of the `Function` type. For example:

```
function show(){
    //...
}

console.log(show instanceof Function); // true
```

The `Function` type has a method named `call()` with the following syntax:

```
functionName.call(thisArg, arg1, arg2, ...);
```

The `call()` method calls a function `functionName` with a given `this` value and arguments.

The first argument of the `call()` method `thisArg` is the `this` value. It allows you to set the `this` value to any given object.

The remaining arguments of the `call()` method `arg1`, `arg2`,... are the arguments of the function.
When you invoke a function, the JavaScript engine invokes the `call()` method of that function object.

Suppose that you have the `show()` function as follows:

```
function show() {
   console.log('Show function');
}
```

And invoke the show() function:

```
show();
```

It is equivalent to invoke the call() method on the show function object:

```
show.call();
```

```
function add(a, b) {
   return a + b;
}

let result = add.call(this, 10, 20);
console.log(result);
```

## Using the JavaScript call() method to chain constructors for an object

The `call()` method can be used for chaining constructors for an object.
Consider the following example:

```
function Box(height, width) {
   this.height = height;
   this.width  = width;
}

function Widget(height, width, color) {
   Box.call(this, height, width);
   this.color = color;
}

let widget = new Widget('red',100,200);
```

## Using the JavaScript **call()** method for function borrowing

The following defines two objects: `car` and `aircraft`:

```
const car = {
    name: 'car',
    start: function() {
        console.log('Start the ' + this.name);
    },
    speedup: function() {
        console.log('Speed up the ' + this.name)
    },
    stop: function() {
        console.log('Stop the ' + this.name);
    }
};

const aircraft = {
    name: 'aircraft',
    fly: function(){
        console.log('Fly');
    }
};
```

The `aircraft` object has the `fly()` method.
The following code uses the `call()` method to invoke the `start()` method of the `car` object on the `aircraft` object:

```
car.start.call(aircraft);
```

Outputt:
Start the aircraft

Technically, the `aircraft` object has borrowed the `start()` method of the `car` object for the `aircraft`.
When an object uses a method of another object is called the function borrowing.

The typical applications of function borrowing are to use the built-in methods of the Array type.

For example, the `arguments` object inside a function is an array-like object, not an array object. To use the `slice()` method of the Array object, you need to use the `call()` method:

```
function getOddNumbers() {
   const args = Array.prototype.slice.call(arguments);
   return args.filter(num => num % 2);
}

let oddNumbers = getOddNumbers(10, 1, 3, 4, 8, 9);
console.log(oddNumbers);
```

Output:

[ 1, 3, 9 ]

In this example, we passed any number of numbers into the function. The function returns an array of odd numbers.

The following statement uses the `call()` function to set the `this` inside the `slice()` method to the `arguments` object and execute the `slice()` method:
const args = Array.prototype.slice.call(arguments);

## JavaScript **apply()** method

The `Function.prototype.apply()` method allows you to call a function with a given `this` value and arguments provided as an array. Here is the syntax of the `apply()` method:

```
fn.apply(thisArg, [args]);
```

The `apply()` method accepts two arguments:

- The `thisArg` is the value of this provided for the call to the function `fn`.
- The `args` argument is an array that specifies the arguments of the function `fn`. Since the ES5, the `args` argument can be an array-like object or array object.

THe `apply()` method is similar to the `call()` method except that it takes the arguments of the function as an array instead of the individual arguments.

```
const person = {
   firstName: 'John',
   lastName: 'Doe'
}
```

```
function greet(greeting, message) {
   return `${greeting} ${this.firstName}. ${message}`;
}

let result = greet.apply(person, ['Hello', 'How are you?']);
console.log(result);
```

OUTPUT:
Hello John. How are you?

**Function borrowing**

```
const computer = {
   name: 'MacBook',
   isOn: false,
   turnOn() {
      this.isOn = true;
      return `The ${this.name} is On`;
   },
   turnOff() {
      this.isOn = false;
      return `The ${this.name} is Off`;
   }
};
```

... and the following `server` object:

```
const server = {
   name: 'Dell PowerEdge T30',
   isOn: false
};

let result = computer.turnOn.apply(server);
console.log(result);
```

OUTPUT:
The Dell PowerEdge T30 is On

## JavaScript Function type

All functions in JavaScript are objects. They are the instances of the `Function` type. Since functions are objects, they have properties and methods like other objects.

## Functions properties

Each function has two properties: `length` and `prototype`.

- The `length` property determines the number of named arguments specified in the function declaration.
- The `prototype` property references the actual function object.

See the following example:

```
function swap(x, y) {
   let tmp = x;
   x = y;
   y = tmp;
}

console.log(swap.length); // 2
console.log(swap.prototype); // Object{}
```

ES6 added a new property called target.new that allows you to detect whether a function ( `fn`) is called as a normal function ( `fn()`) or as a constructor using the new operator ( `new fn()`).

## JavaScript Primitive vs. Reference Values

**Accessing by value and reference**

In JavaScript, a variable may store two types of values: primitive and reference.

JavaScript provides six primitive types as `undefined`, `null`, `boolean`, `number`, `string`, and `symbol` , and a reference type `object`.

The size of a primitive value is fixed, therefore, JavaScript stores the primitive value on the stack.

On the other hand, the size of a reference value is dynamic so JavaScript stores the reference value on the heap.

When you assign a value to a variable, the JavaScript engine will determine whether the value is a primitive or reference value.

If the value is a primitive value, when you access the variable, you manipulate the **actual value** stored in that variable. In other words, the variable that stores a primitive value is **accessed by value**.

Unlike a primitive value, when you manipulate an object, you work on the **reference** of that object, rather than the actual object. It means a variable that stores an object is **accessed by reference**.

To determine the type of a primitive value you use the `typeof` operator. For example:

```
let x = 10;
console.log(typeof(x)); // number
```

and

```
let str = 'JS';
console.log(typeof(str)); // string
```

To find the type of a reference value, you use the `instanceof` operator:

```
let refType = variable instanceof constructor;
```

```
let str = 'JS';
function f(){}
var t = new f;

let refType = str instanceof constructor;
console.log(refType) //false
console.log(t) //true
```

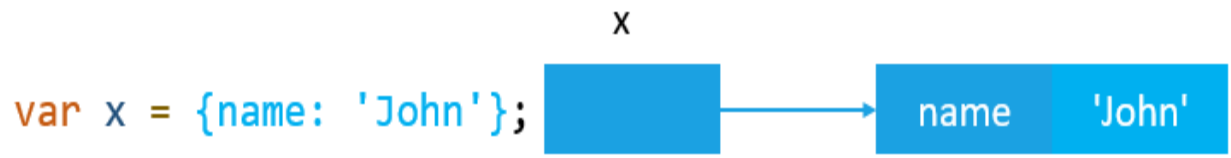
## Copying reference values

When you assign a reference value from one variable to another, the value stored in the variable is also copied into the location of the new variable. The difference is that the values stored in both variables now are the address of the actual object stored on the heap. As a result, both variables are referencing the same object.

Consider the following example.

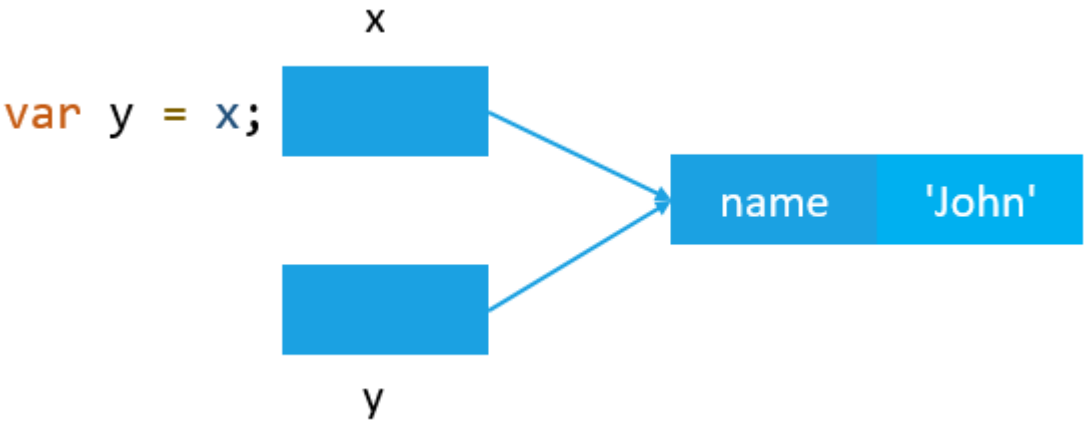
First, declare a variable `x` that holds an object whose name property is `'John'`.

```
var x = {name: 'John'};
```




Second, declare another variable y and assign the `x` variable to `y`. Both `x` and `y` are now referencing the same object on the heap.

```
var y = x;
```




## Enumerable Properties of an Object in JavaScript

In JavaScript, an object is an unordered list of key-value pairs. The key is usually a string or a symbol. The value can be a value of any primitive type (string, boolean, number, undefined, or null), an object, or a function.

The following example creates a new object using the object literal syntax:

```
const person = {
    firstName: 'John',
    lastName: 'Doe
};
```

An object property has several internal attributes including `value`, `writable`, `enumerable` and `configurable`

The `enumerable` attribute determines whether or not a property is accessible when the object's properties are enumerated using the `for...in` loop or `Object.keys()` method.

By default, all properties created via a simple assignment or via a property initializer are enumerable. For example:

```
const person = {
    firstName: 'John',
    lastName: 'Doe'
};

person.age = 25;

for (const key in person) {
    console.log(key);
}
```

To change the internal `enumerable` attribute of a property, you use the `Object.defineProperty()` method. For example:

```
const person = {
    firstName: 'John',
    lastName: 'Doe'
};

person.age = 25;

Object.defineProperty(person, 'ssn', {
    enumerable: false,    → True accessible
    value: '123-456-7890'
});

for (const key in person) {
    console.log(key);
}
```

Output:
firstName
lastName
age
ssn → will not be accessible if enumerable:false

## JavaScript try...catch Statement

To handle errors in JavaScript, you use the `try...catch` statement:

```
try {
  nonExistingFunction();
} catch(error){
    console.log(error.name + ":" + error.message);
}
```

### The `finally` clause

The `finally` clause is an optional clause of the `try...catch` statement. The code that you place in the `finally` block always executes whether the error occurs or not.

It means that if the code in the `try` block executes entirely, the code in `finally` block executes; In case an error occurs that causes the code in the catch block executes, the code in the `finally` block also executes.

```
function test(){
  try {
    return 1;
  } catch(error) {
    return 2;
  } finally {
    return 3;
  }
}
console.log(test());
```

**RangeError**

The `RangeError` occurs when a number is not in its range. See the following example:

```
try {
    let list = Array(Number.MAX_VALUE);
} catch (error) {
    console.log(error.name); // "RangeError"
    console.log(error.message); // "Invalid array length"
}
```

Error
RangeError
Invalid array length

### ReferenceError

The `ReferenceError` occurs when you reference a variable, a function, or an object that does not exist.

### SyntaxError

The `SyntaxError` occurs in a string that you pass to the `eval()` function.

### TypeError

The `TypeError` occurs when a variable is of an unexpected type or access to a nonexistent method

### URIError

The `URIError` error occurs when using the `encodeURI()` or `decodeURI()` with a malformed URI,

### Throwing errors

To throw an error, you use the `throw` operator. For example:
throw 'ABC';
throw 123;

### Custom Error

You can create a custom error that inherits from a built-in error as follows:

```
function InvalidCallError(message) {
   this.name = 'InvalidCallError';
   this.message = message;
}

InvalidCallError.prototype = Object.create(Error.prototype);
InvalidCallError.prototype.constructor = Error;
```

## JAVASCRIPT RUNTIME

### How JavaScript code gets executed.

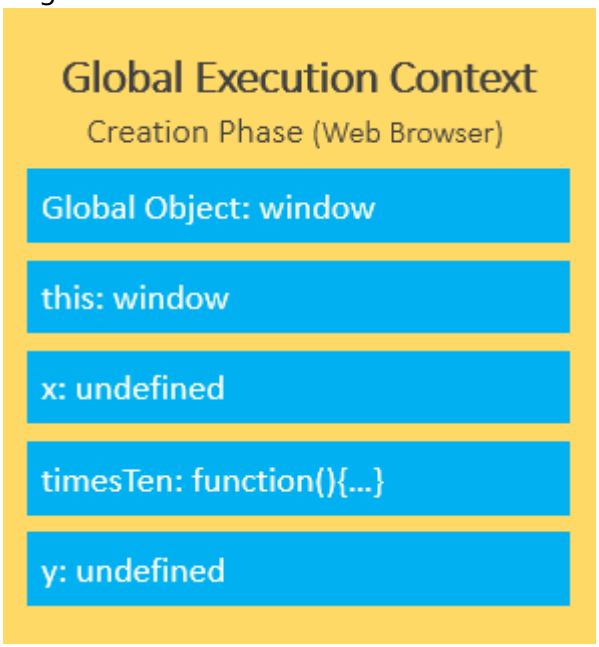
```
let x = 10;
function timesTen(a){
   return a * 10;
}
let y = timesTen(x);
console.log(y); // 100
```

When a JavaScript engine executes a script, it creates execution contexts.
An execution context is an abstract concept of an environment where the Javascript code is evaluated and executed. Whenever any code is run in JavaScript, it's run inside an execution context.

**There are three types of execution context in JavaScript**.

1. **Global Execution Context:**

This is the default or base execution context. The code that is not inside any function is in the global execution context. It performs two things: it creates a global object which is a window object (in the case of browsers) and sets the value of this to equal to the global object. There can only be one global execution context in a program. This is the default execution context in which JS code start its execution when the file first loads in the browser. It cannot be more than one because only one global environment is possible for JS code execution as the JS engine is single threaded.




2. **Functional Execution Context:**

Whenever the flow of execution enters a function body. Functional execution context is defined as the context created by the JS engine whenever it finds any function call. Each function has its own execution context. It can be more than one. Functional execution context has access to all the code of the global execution context though vice versa is not applicable. While executing the global execution context code, if JS engine finds a function call, it creates a new functional execution context for that function. In the browser context, if the code is executing in strict mode value of this is undefined else it is window object in the function execution context.

3. **Eval Function Execution Context:**

Text to be executed inside the internal eval function, But as eval isn't usually used by JavaScript developers.

**Each execution context has two phases:**

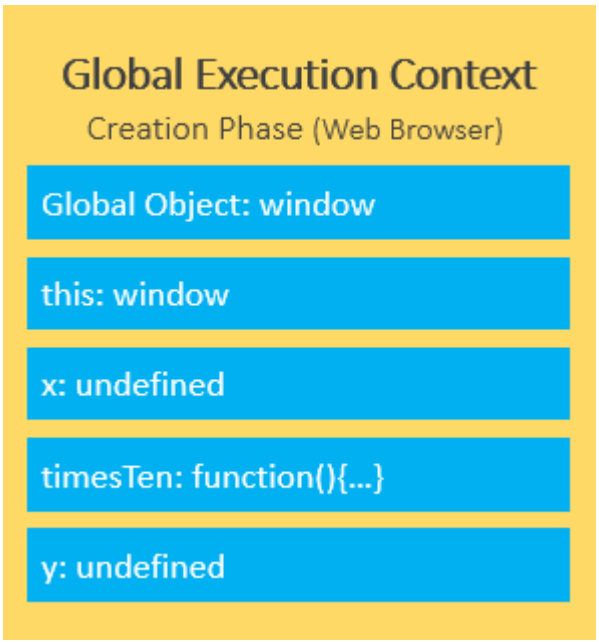
1. The creation phase.
2. The execution phase.

**The creation phase:**

**Creation phase** is the phase in which the JS engine has called a function but its execution has not started. In the creation phase, JS engine is in the compilation phase and it just scans over the function code to compile the code, it doesn't execute any code.
When a script executes for the first time, the JavaScript engine creates a **Global Execution Context.** During this creation phase, it performs the following tasks:

- Create a global object i.e., **window** in the web browser or **global** in Node.js.
    - Global object is a special object in JS which contain all the variables, function arguments and inner functions declaration information
- Create **this** object binding which points to the global object above.
    - The JavaScript engine initializes the value of this.
- Setup a memory heap for storing variables and function references.
- Store the function declarations in the memory heap and variables within the global execution context with the initial values as **undefined**.

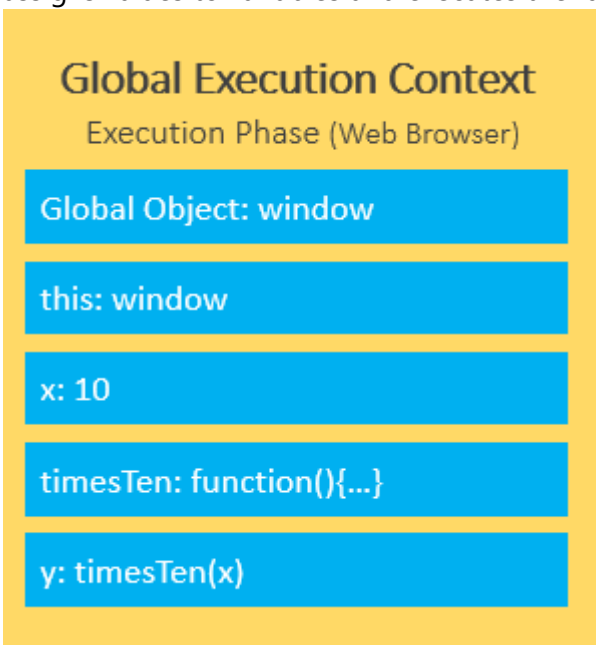
In our example, during the creation phase, the JavaScript engine stores the variables x and y and the function declaration timesTen() in the Global Execution Context. Besides, it initializes the variables x and y to **undefined**.




After the creation phase, the global execution context moves to the execution phase.

### The execution phase

During the execution phase, the JavaScript engine executes the code line by line. In this phase, it assigns values to variables and executes the function calls.




For every function call, the JavaScript engine creates a new **Function Execution Context**. The Function Execution Context is similar to the Global Execution Context, but instead of creating the global object, it creates the **arguments object** that contains a reference to all the parameters passed into the function:

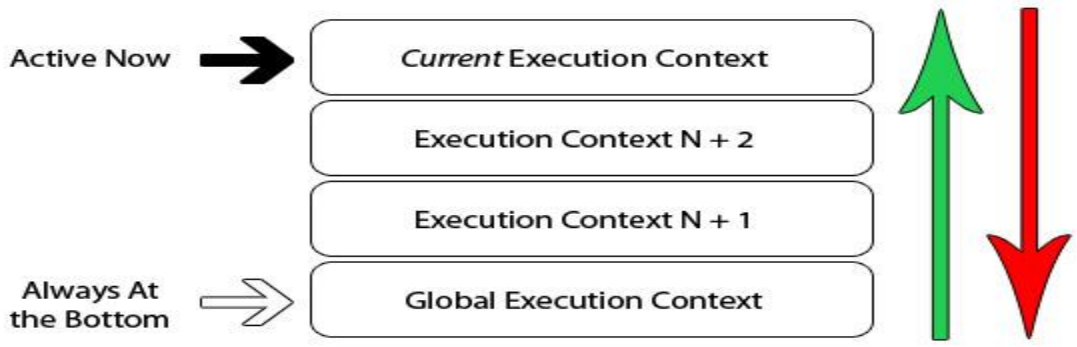

During the execution phase of the function execution context, it assigns 10 to the parameter **a** and returns the result (100) to the Global Execution Context:


To keep track of all the execution contexts including the Global Execution Context and Function Execution Contexts, the JavaScript engine uses a data structure named call stack.

This brings us to another important concept. How does engine know which execution context to enter or exit? The answer is the Call Stack.

**Execution context stack (ECS):** Execution context stack is a stack data structure, i.e. last in first out data structure, to store all the execution stacks created during the life cycle of the script. Global execution context is present by default in execution context stack and it is at the bottom of the stack. While executing the global execution context code, if JS engines find a function call, it creates a functional execution context for that function and pushes it on top of the execution context stack. JS engine executes the function whose execution context is at the top of the execution context stack. Once all the code of the function is executed, JS engines pop out that function's execution context and start's executing the function which is below it.

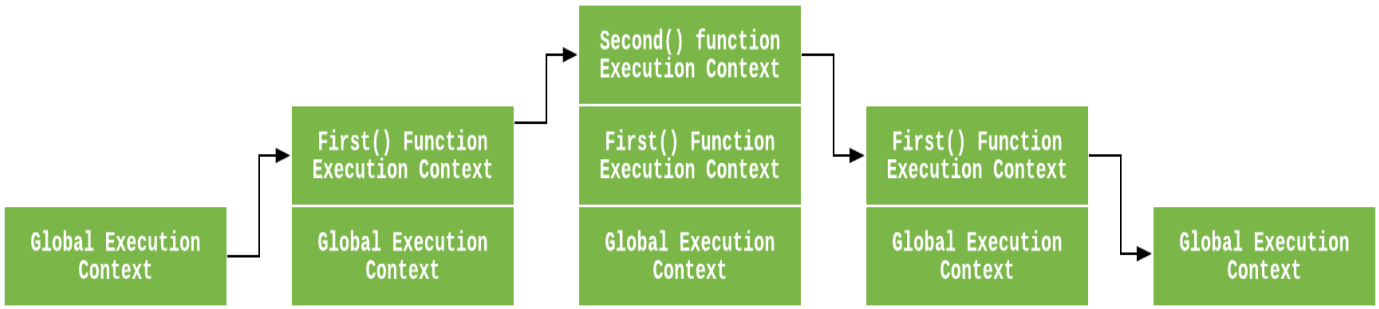
1. JavaScript engine uses a **call stack** to manage execution contexts: the Global Execution Context and Function Execution Contexts.
2. When you execute a script, the JavaScript engine creates a Global Execution Context and pushes it on top of the call stack.
3. Whenever a function is called, the JavaScript engine creates a Function Execution Context for the function, pushes it on top of the Call Stack, and starts executing the function.
4. If a function calls another function, the JavaScript engine creates a new Function Execution Context for the function that is being called and pushes it on top of the call stack.
5. When the current function completes, the JavaScript engine pops it off the call stack and resumes the execution where it left off in the last code listing.
6. The script will stop when the call stack is empty.




Let's understand this with a code example below:

```
let a = 'Hello World!';
```

```
function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}

function second() {
  console.log('Inside second function');
}

first();
console.log('Inside Global Execution Context');
```




When the above code loads in the browser, the JavaScript engine creates a global execution context and pushes it to the current execution stack. When a call to first () is encountered, the JavaScript engines creates a new execution context for that function and pushes it to the top of the current execution stack.

When the second () function is called from within the first () function, the JavaScript engine creates a new execution context for that function and pushes it to the top of the current execution stack. When the second () function finishes, its execution context is popped off from the current stack, and the control reaches to the execution context below it, that is the first () function execution context.

When the first() finishes, its execution stack is removed from the stack and control reaches to the global execution context. Once all the code is executed, the JavaScript engine removes the global execution context from the current stack.

## Stack overflow

The call stack has a fixed size, depending on the implementation of the host environment, either the web browser or Node.js. If the number of the execution contexts exceeds the size of the stack, a stack overflow will occur.

## Asynchronous JavaScript

JavaScript is the single-threaded programming language. The JavaScript engine has only one call stack so that it only can do one thing at a time.

When executing a script, the JavaScript engine executes code from top to bottom, line by line. In other words, it is **synchronous**.

Asynchronous is the opposite of synchronous, which means happening at the same time. So how does JavaScript carry asynchronous tasks such as callbacks, promises, and async/await?


Reference:
https://2ality.com/

https://github.com/rus0000/jsinheritance
http://speakingjs.com/es5/index.html
http://www.javascripttutorial.net/javascript-prototype/
http://stackoverflow.com/questions/9959727/proto-vs-prototype-in-javascript
http://dmitrysoshnikov.com/ecmascript/javascript-the-core/
https://www.quora.com/What-is-the-difference-between-__proto__-and-prototype
http://www.jisaacks.com/prototype-vs-proto/
http://kenneth-kin-lum.blogspot.tw/2012/10/javascripts-pseudo-classical.html