

## 1 Seven (7) Types

### Six Primitive Types

1. String  
`"Any text"`
2. Number  
`123.45`
3. Boolean  
`true` or `false`
4. Null  
`null`
5. Undefined  
`undefined`
6. Symbol  
`Symbol('something')`
7. Object
  - Array  
`[1, "text", false]`
  - Function  
`function name() { }`

## 2 Basic Vocabulary

### Variable

A named reference to a value is a variable.

### Operator

Operators are reserved-words that perform action on values and variables. Examples: `+` `-` `=` `*` `in` `===` `typeof` `!=` ...

```
var a = 7 + "2";
```

### Keyword / reserved word

Any word that is part of the vocabulary of the programming language is called a keyword (a.k.a reserved word).

Examples: `var` `=` `+` `if` `for`...

### Expression

A reference, value or a group of reference(s) and value(s) combined with operator(s), which result in a single value.

### Statement

A group of words, numbers and operators that **do a task** is a statement.

## 3 Object

An object is a data type in JavaScript that is used to store a combination of data in a simple key-value pair. That's it.

### Key

These are the keys in `user` object.

```
var user = {
  name: "Aziz Ali",
  yearOfBirth: 1988,
  calculateAge: function(){
    // some code to calculate age
  }
}
```

### Value

These are the values of the respective keys in `user` object.

### Method

If a key has a function as a value, it's called a method.

## 4 Function

A function is simply a bunch of code bundled in a section. This bunch of code ONLY runs when the function is called. Functions allow for organizing code into sections and code reusability.

Using a function has ONLY two parts. (1) Declaring/defining a function, and (2) using/running a function.

### Name of function

That's it, it's just a name you give to your function. Tip: Make your function names descriptive to what the function does.

### Return (optional)

A function can optionally spit-out or "return" a value once it's invoked. Once a function returns, no further lines of code within the function run.

### Invoke a function

Invoking, calling or running a function all mean the same thing. When we write the function name, in this case `someName`, followed by the brackets symbol `()` like this `someName()`, the code inside the function gets executed.

```
// Function declaration / Function statement
```

```
function someName(param1, param2) {
```

```
// bunch of code as needed...
```

```
var a = param1 + "love" + param2;
```

```
return a;
```

```
}
```

```
// Invoke (run / call) a function
```

```
someName("Me", "You")
```

### Parameters / Arguments (optional)

A function can optionally take parameters (a.k.a arguments). The function can then use this information within the code it has.

### Code block

Any code within the curly braces `{ ... }` is called a "block of code", "code block" or simply "block". This concept is not just limited to functions. "if statements", "for loops" and other statements use code blocks as well.

### Passing parameter(s) to a function (optional)

At the time of invoking a function, parameter(s) may be passed to the function code.

## 5 Vocabulary around variables and scope

```
var a;
```

### Variable Declaration

The creation of the variable.

```
a = 12;
```

### Variable Initialization

The initial assignment of value to a variable.

```
a = "me";
```

### Variable Assignment

Assigning value to a variable.

```
console.log(a);
```

```
var a = "me";
```

### Hoisting

Variables are declared at the top of the function automatically, and initialized at the time they are run.

### Scope

The limits in which a variable exists.

#### Global scope

The outer most scope is called the Global scope.

#### Functional scope

Any variables inside a function is in scope of the function.

#### Lexical Environment (Lexical scope)

The physical location (scope) where a variable or function is declared is its lexical environment (lexical scope).

#### Rule:

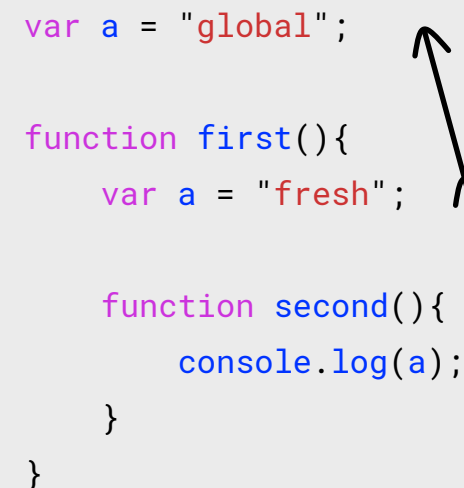
(1) Variables in the outer scope can be accessed in a nested scope; But variables inside a nested scope CANNOT be accessed by the outer scope. (a.k.a private variables.)

(2) Variables are picked up from the lexical environment.

```
var a = "global";

function first(){
  var a = "fresh";

  function second(){
    console.log(a);
  }
}
```



### Scope chain

The nested hierarchy of scope is called the scope chain. The JS engine looks for variables in the scope chain upwards (it its ancestors, until found)

Full list of JavaScript operators <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators>

## 6 Operators

Operators are reserved-words that perform action on values and variables.

### Arithmetic

.. + .. Add  
.. - .. Subtract  
.. \* .. Multiply  
.. / .. Divide  
.. % .. Remainder  
.. \*\* .. Exponential

### Assignment

.. = .. Assign value  
.. += .. Add then assign  
.. -= .. Subtract then assign  
.. \*= .. Multiply then assign

### Logical

.. || .. Or  
.. && .. And

### Equality

.. === .. Equality  
.. == .. Equality with coercion

### Conversion

+ .. Convert to number  
- .. Convert to number then negate it  
! .. Convert to boolean then inverse it

### Relational / Comparison

.. >= .. Greater than or equal to  
.. <= .. Less than or equal to  
.. != .. Not equal after coercion  
.. !== .. Not equal

### Increment / Decrement

.. ++ Postfix increment  
.. -- Postfix decrement

++ .. Prefix increment  
-- .. Prefix decrement

### Others

typeof ..  
.. instanceof ..  
(..)  
...spread-operator  
..  
..[..]  
new ..  
delete ..  
(.. ? .. : ..)

### Operator Precedence

Given multiple operators are used in an expression, the "Operator Precedence" determines which operator will be executed first. The higher the precedence, the earlier it will get executed.

### Operator Associativity

Given multiple operators have the same precedence, "Associativity" determines in which direction the code will be parsed.

See the **Operator Precedence and Associativity table** here:

<http://bit.ly/operatortable>

## 7 Coercion

When trying to compare different "types", the JavaScript engine attempts to convert one type into another so it can compare the two values.

### Type coercion priority order:

1. String
2. Number
3. Boolean

```
2 + "7"; // "27"  
true - 5 // -4
```

### Coercion in action

Does this make sense?

## 8 Conditional Statements

Conditional statements allow our program to run specific code only if certain conditions are met. For instance, let's say we have a shopping app. We can tell our program to hide the "checkout" button if the shopping cart is empty.

**If-else Statement:** Run certain code, "if" a condition is met. If the condition is not met, the code in the "else" block is run (if available.)

```
if (a > 0) {  
    // run this code  
}  
else if (a < 0) {  
    // run this code  
}  
else {  
    // run this code  
}
```

**Ternary Operator:** A ternary operator returns the first value if the expression is truthy, or else returns the second value.

```
(expression)? ifTrue: ifFalse;
```

**Switch Statement:** Takes a single expression, and runs the code of the "case" where the expression matches. The "break" keyword is used to end the switch statement.

```
switch (expression) {  
    case choice1:  
        // run this code  
        break;  
  
    case choice1:  
        // run this code  
        break;  
  
    default:  
        // run this code  
}
```

## 9 Truthy / Falsy

There are certain values in JavaScript that return true when coerced into boolean. Such values are called **truthy** values. On the other hand, there are certain values that return false when coerced to boolean. These values are known as **falsy** values.

### Truthy Values

```
true  
"text"  
72  
-72  
Infinity  
-Infinity  
{  
[]
```

### Falsy Values

```
false  
"  
0  
-0  
NaN  
null  
undefined
```

## 10 Loop Statements

Loops are used to do something repeatedly. For instance lets say we get a list of 50 blog posts from the database and we want to print their titles on our page. Instead of writing the code 50 times, we would instead use a loop to make this happen.

### For loop

```
for (initial-expression; condition; second-expression){  
  // run this code in block  
}
```

**Step 1:** Run the initial expression.

**Step 2:** Check if condition meets. If condition meets, proceed; or else end the loop.

**Step 3:** Run the code in block.

**Step 4:** Run the second-expression.

**Step 5:** Go to Step 2.

### While loop

```
while (i<3){  
  // run this code in block  
  i++;  
}
```

**Step 1:** If the condition is true, proceed; or else end the loop.

**Step 2:** Run the code in block.

**Step 3:** Go to Step 1.

### Do while loop

```
do {  
  // run this code in block  
  i++;  
} while (i<3);
```

**Step 1:** Run the code in block.

**Step 2:** If the condition is true, proceed; or else end the loop.

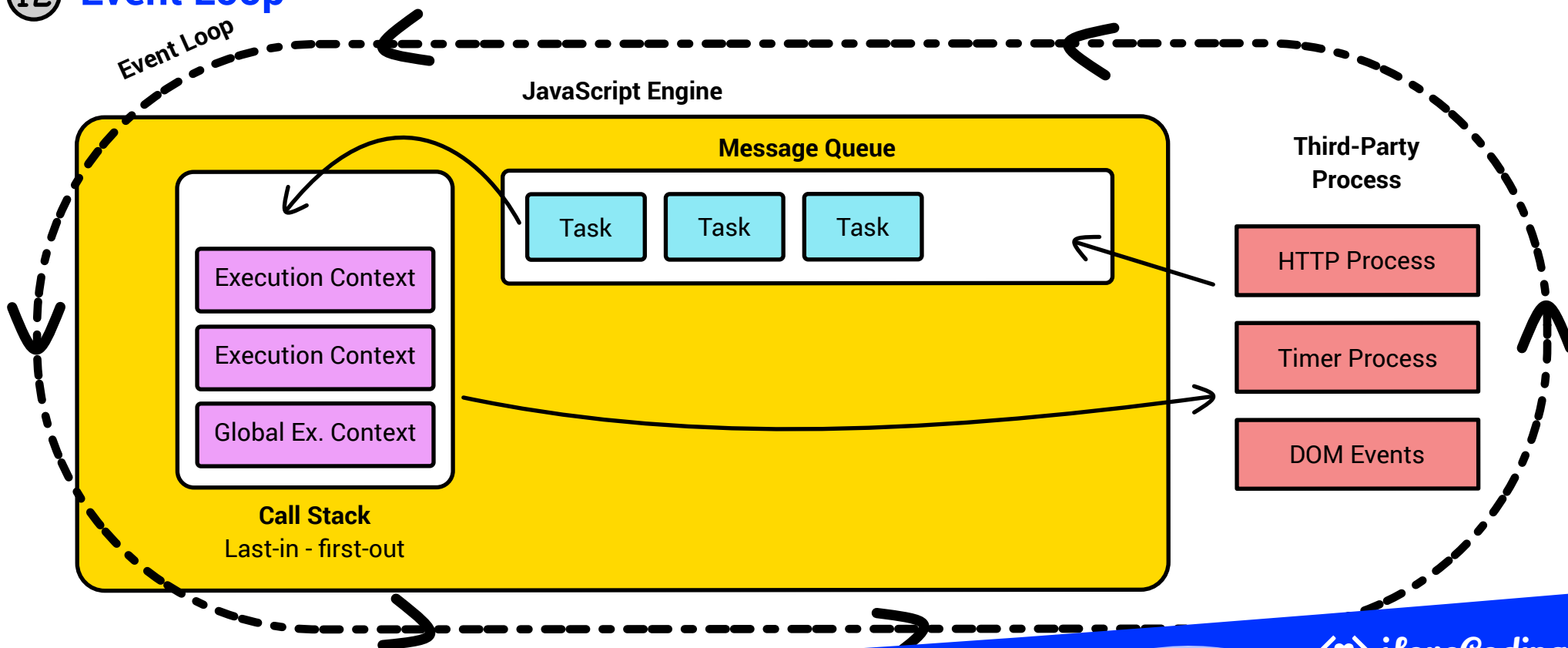
**Step 3:** Go to Step 1.

## 11 Ways to create a variable

There are 3 ways to create variables in JavaScript: **var**, **let** and **const**. Variables created with **var** are in scope of the function (or global if declared in the global scope); **let** variables are block scoped; and **const** variables are like **let** plus their values cannot be re-assigned.

```
var a = "some value"; // functional or global scoped
let b = "some value"; // block scoped
const c = "some value"; // block scoped + cannot get new value
```

## 12 Event Loop



## 13 Browser

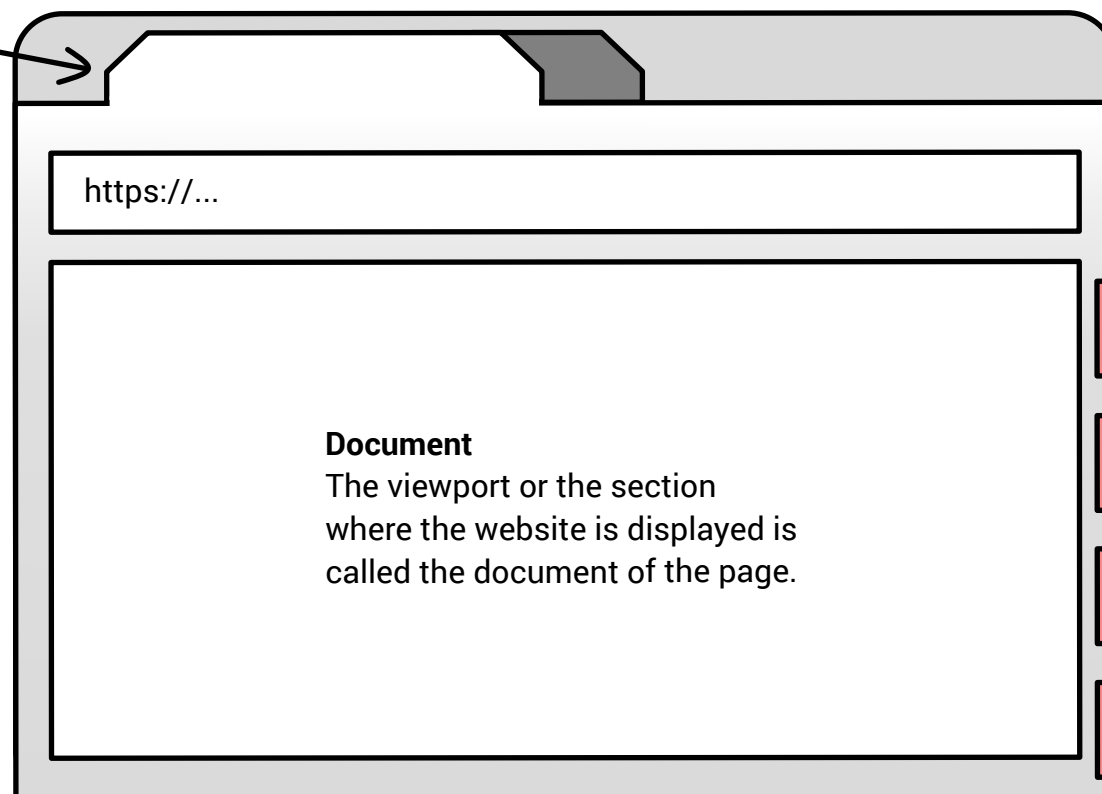
A web browser is a pretty advance piece of software which contains a lot of components. Many of these components are accessible to a web developer, so we can create complex web apps. At the same time a lot of components are kept out of reach of the web developer for security purposes. For instance, we as web developers can get access to the user's location, but we cannot get access to the user's saved passwords or browsing history. **Let's see below how a browser is structured:**

### Window

Each tab of a browser is considered the window.

This is the outer most container that a web-app can access.

Notice: A website opened in one tab CANNOT access the window object of another tab. Pretty cool right?



The browser contains a lot of components that a Front-End Developer may need, such as **Navigator**, **JavaScript Engine** and **Dev Tools**.

Navigator

HTML / CSS Processor

JavaScript Engine

Dev Tools



## 14 DOM - Document Object Model

### Query/Get Elements

```
// Preferred way:  
document.querySelector('css-selectors')  
document.querySelectorAll('css-selectors', ...)
```

```
// Old ways, and still work:  
document.getElementsByTagName('element-name')  
document.getElementsByClassName('class-name')  
document.getElementById('id')
```

### Create / clone Element

```
document.createElement('div')  
document.createTextNode('some text here')  
node.cloneNode()  
node.textContent = 'some text here'
```

### Add node to document

```
parentNode.appendChild(nodeToAdd)  
parentNode.insertBefore(nodeToAdd, childNode)
```

### Get Element Details

```
node.nextSibling  
node.firstChild  
node.lastChild  
node.parentNode  
node.childNodes  
node.children
```

### Modify Element

```
node.style.color = 'red'  
node.style.padding = '10px',  
node.style.fontSize = '200%'  
  
node.setAttribute('attr-name', 'attr-value')  
node.removeAttribute('attr-name')
```

### Get and Modify Element Class

```
node.classList  
node.classList.add('class-name', ...)  
node.classList.remove('class-name', ...)  
node.classList.toggle('class-name')  
node.classList.contains('class-name')  
node.classList.replace('old', 'new')
```

### Remove Node

```
parentNode.removeChild(nodeToRemove)  
// Hack to remove self  
nodeToRemove.parentNode.removeChild(nodeToRemove)
```

### Events

```
node.addEventListener('event-name', callback-function)  
node.removeEventListener('event-name', callback-function)
```

List of Events: <https://developer.mozilla.org/en-US/docs/Web/Events>  
or google "Mozilla event reference"

### What is a "Node"? (in the context of DOM)

**Node:** Every item in the DOM tree is called a node. There are two types of node - A text node, and an element node:

**Text Node:** Node that has text.

**Element Node:** Node that has an element.

**Child Node:** A node which is a child of another node.

**Parent Node:** A node which has one or more child.

**Descendent Node:** A node which is nested deep in the tree.

**Sibling Node:** A node that share the same parent node.

## 15 Auto Inherited Properties

When you create a value in JavaScript, certain properties are automatically inherited by this value. This magic happens because every **type** has a **constructor** with a special property called **prototype**. All methods on the **prototype** gets automatically inherited by the new value created for that **type**. Take a look at some of these methods on the right.

```
const thing = "some text";
```

### String

Google 'Mozilla String' to find the docs

```
.concat()  
.charAt()  
.indexOf()  
.startsWith()  
.endsWith()  
.split()  
.slice()
```

```
const num = 123.45;
```

### Number

Google 'Mozilla Number' to find the docs

```
.toFixed()  
.toPrecision()  
.toString()
```

### Boolean

Google 'Mozilla Boolean' to find the docs

```
.toString()
```

### Array

Google 'Mozilla Array' to find the docs

```
.filter()  
.map()  
.find()  
.every()  
.some()  
.sort()  
.slice()  
.splice()  
.reduce()  
.forEach()
```

## 16 Built-in Objects

JavaScript gives us a ton of useful built-in objects to make our lives easier. The **Date** and **Math** objects are very useful on a regular basis. Take a look at some of their features on the right.

Full list of builtin objects in JavaScript visit [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)

### Math

Google 'Mozilla Math' to find the docs

```
Math.pow(2, 3) // 8  
Math.sqrt(16) // 4  
Math.min(7, 8, 6) // 6  
Math.max(7, 8, 6) // 8  
Math.floor(123.45) // 123  
Math.ceil(123.45) // 124  
Math.round(123.45) // 123  
Math.random() // 0.45..
```

### Date

Google 'Mozilla Date' to find the docs

```
const d = new Date('9/17/1988');  
d.getDay()  
d.getFullYear()  
d.getMonth()  
  
Date.now()  
Milliseconds since Jan 1, 1970
```

## 17 Promise

### What is a Promise?

Promise is an object that provides a useful construct when dealing with asynchronous tasks. A promise is called a "Promise" because it guarantees it will run upon success or failure of that task.

Working with a promise consists of two parts; (A) Creating a promise, and (B) Using a promise.

```
// (A) Create a promise
const p = new Promise((resolve, reject)=>{
  // Do some async task
  setTimeout(()=>{
    if(condition){
      resolve('Successful login');
    } else {
      reject('Login failed');
    }
  }, 2000)
})
```

### What is an Async task?

An async task is one in which a third-party process is doing the task.

Examples:

- Requesting/sending data to a database
- Requesting/sending data via HTTP protocol
- Working with the file system of the computer

```
// (B) Using a promise
p.then((res)=>{
  console.log(res)
})
.catch((err)=>{
  console.log(err)
})
```

**Note:** 90% of the time you will be working with pre-existing promises. The step of "Creating a promise" would be done for you either by a library, framework or environment you are using. Examples of promises: fetch

## ⑮ 'this' keyword

The **this** keyword is used inside a function. The **this** keyword is merely a reference to another object.

What the **this** keyword refers to depends on the scenario or the way the function is implemented. Here are the 3 scenarios to remember:

### Scenario #1: **this** inside a function

The **this** keyword points to global object.

### Scenario #2: **this** inside a method

The **this** keyword points to the object the method is in.

### Scenario #3: When function is run with **call**, **bind** or **apply**

When a function is called using the **.call(param)**, **.bind(param)** or **.apply(param)** method, the first **param** become the object that the **this** keyword refers to.

### **Important Note:**

In the browser, global is the **window** object.  
In Node.js, global is the **global** object.

```
var name = "Fatema";
```

```
function fun(){  
  // some code here  
  console.log(this.name);  
}
```

```
const user = {  
  name: "Marium",  
  yearOfBirth: 1999,  
  calcAge: function(){  
    const currentYear = (new Date()).getFullYear();  
    return currentYear - this.yearOfBirth;  
  }  
}
```

```
fun(); // 'this' is global. Logs "Fatema"
```

```
user.calcAge(); // 'this' is the user object
```

```
fun.call(user); // 'this' is the user object. Logs "Marium"
```

## 19 Constructor

### What is a constructor?

In JavaScript, a constructor is a special function that acts as a mold to create new objects.

There are numerous built-in constructors in JavaScript, such as [String](#), [Number](#), [Promise](#), [Date](#), [Array](#), [Object](#), and many more.

We can create our own custom constructors if need be.

A great place to use a constructor is when you are creating multiple objects of the same kind.

There are two parts to working with a constructor:

#### (1) Defining a constructor

When creating a custom constructor

#### (2) Using a constructor

with the "new" keyword

```
// Defining a Constructor
```

```
function Car(make, model, year){
  this.make = make;
  this.model = model;
  this.year = year;

  this.setMiles = function(miles){
    this.miles = miles
    return miles;
  }
}
```

```
// Using a constructor
```

```
const car1 = new Car('Toyota', 'Prius', 2016);
const car2 = new Car('Hyundai', 'Sonata', 2018);
```

```
// Adding method to the constructor prototype
```

```
Car.prototype.age = function(){
  return (new Date()).getFullYear() - this.year;
}
```

```
car1.age(); // 2
```

### Rule of thumb:

A) Set properties inside a constructor.

B) Set methods inside the prototype property.

### "new" keyword

The new keyword is used to create a new object (instance) from the constructor.

### "prototype" property

prototype is a special property on every object. Properties (methods or values) attached to the prototype property get inherited to every instance of the constructor.