Introduction to JavaScript anonymous functions

An anonymous function is a function without a name. The following shows how to define an anonymous function:

```
let x = myFunction(4, 3);

function myFunction(a, b) {
// Function returns the product of a and b
  return a * b;
}
```

For example, the following shows an anonymous function that displays a message:

```
let show = function() {
    console.log('Anonymous function');
};

show();
```

JavaScript ES6

JavaScript **ES6** (also known as **ECMAScript 2015** or **ECMAScript 6**) is the newer version of JavaScript that was introduced in 2015.

ECMAScript is the standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.

JavaScript Arrow Function

In the **ES6** version, you can use arrow functions to create function expressions. For example,

```
// function expression
let x = function(x, y) {
   return x * y;
```

```
}
```

can be written as

```
// function expression using arrow function
let x = (x, y) => x * y;
```

Example 1: Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,

```
let greet = () => console.log('Hello');
greet(); // Hello
```

Example 2: Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses. For example,

```
let greet = x => console.log(x);
greet('Hello'); // Hello
```

Example 3: Arrow Function as an Expression

You can also dynamically create a function and use it as an expression. For example,

```
let age = 5;

let welcome = (age < 18) ?
  () => console.log('Baby') :
  () => console.log('Adult');

welcome(); // Baby
```

Example 4: Multiline Arrow Functions

If a function body has multiple statements, you need to put them inside curly brackets {}. For example,

```
let sum = (a, b) => {
    let result = a + b;
    return result;
}

let result1 = sum(5,7);
console.log(result1); // 12
```

Inside a regular function

```
function Person() {
    this.name = 'Jack',
    this.age = 25,
    this.sayName = function () {

        // this is accessible
        console.log(this.age);

        function innerFunc() {

            // this refers to the global object
            console.log(this.age);
            console.log(this);
        }

        innerFunc();

    }
}
```

```
let x = new Person();
x.sayName();
```

Output:
25
undefined
window{}

However, innerFunc() is a normal function and this.age is not accessible because this refers to the global object (Window object in the browser). Hence, this.age inside the innerFunc() function gives undefined.

Inside an arrow function

```
function Person() {
    this.name = 'Jack',
    this.age = 25,
    this.sayName = function () {

        console.log(this.age);
        let innerFunc = () => {
            console.log(this.age);
        }

        innerFunc();
    }
}

const x = new Person();
x.sayName();
```

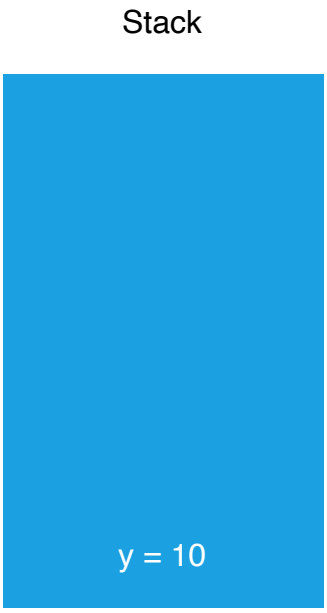
Output:
25
25

Pass-by-value of primitives values

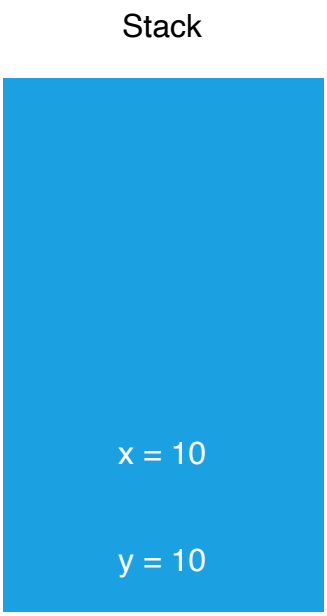
Let's take a look at the following example.

```
function square(x) {
    x = x * x;
    return x;
}

let y = 10;
let result = square(y);

console.log(result); // 100
console.log(y); // 10 -- no change
```

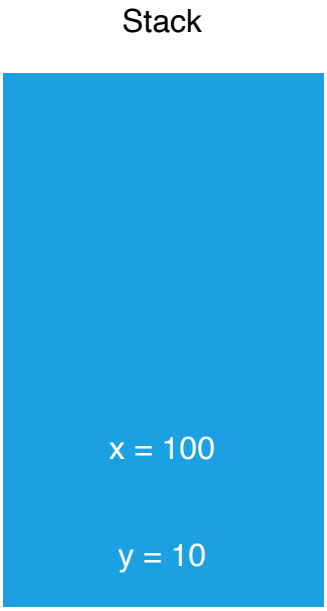
First, define a square() function that accepts an argument x. The function assigns the square of x to the x argument.



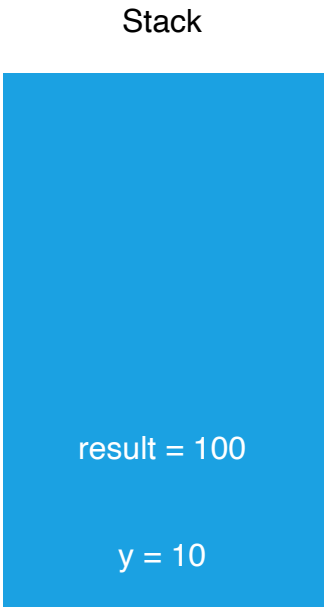
Then, pass the y variable into the square() function. When passing the variable y to the square() function, JavaScript copies y value to the x variable.



After that, the `square()` function changes the `x` variable. However, it does not impact the value of the `y` variable because `x` and `y` are separate variables.



Finally, the value of the `y` variable does not change after the `square()` function completes.



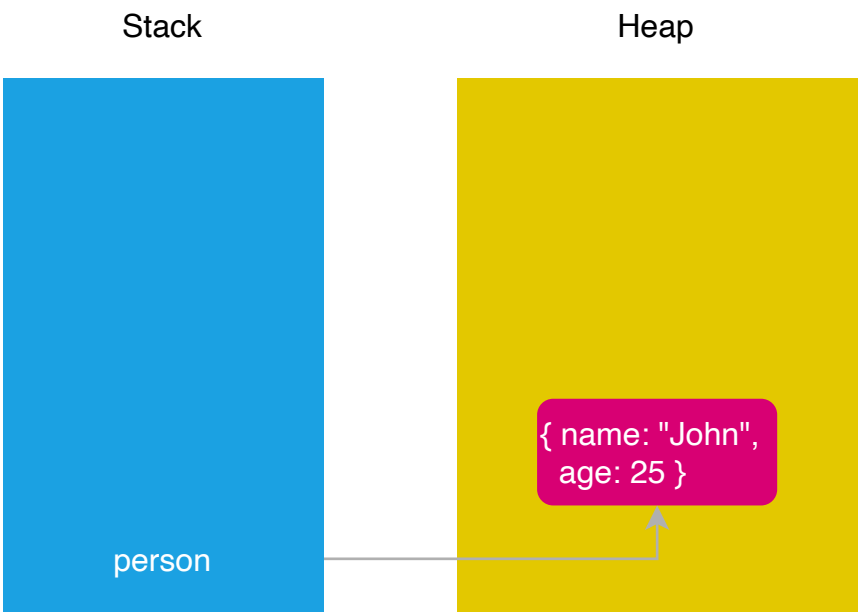
If JavaScript used the pass-by-reference, the variable `y` would change to `100` after calling the function.

Pass-by-value of reference values

It's not obvious to see that reference values are also passed by values. For example:

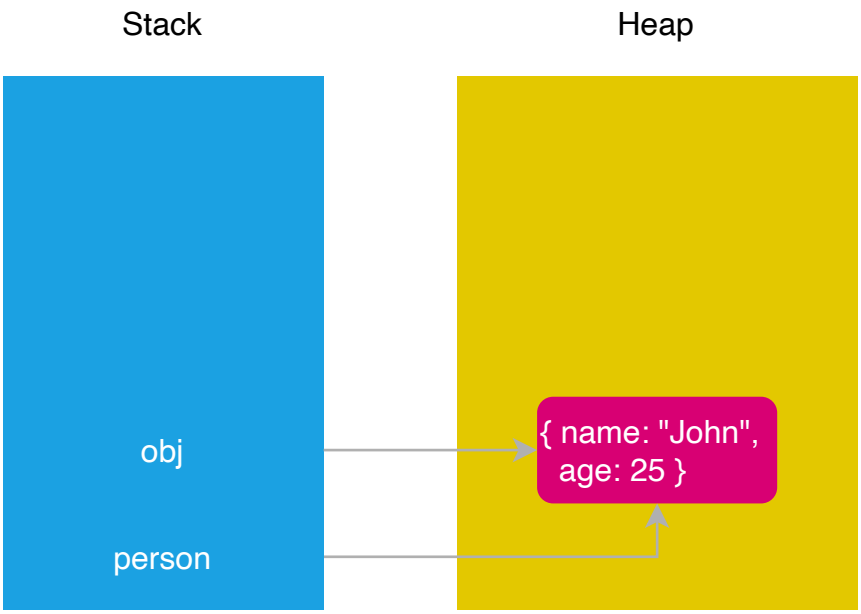
```
let person = {
  name: 'John',
  age: 25,
};

function increaseAge(obj) {
  obj.age += 1;
}

increaseAge(person);

console.log(person);
```

First, define the `person` variable that references an object with two properties `name` and `age`:



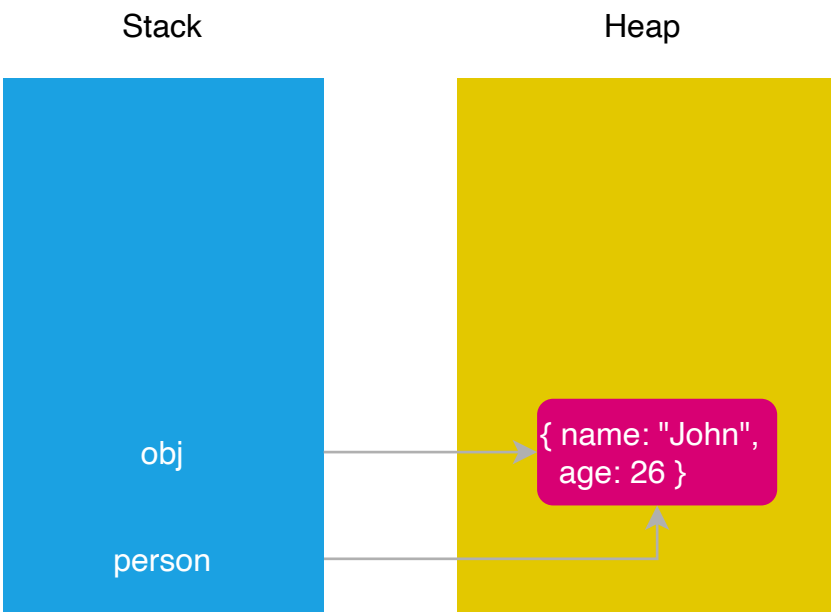
Next, define the `increaseAge()` function that accepts an object `obj` and increases the `age` property of the `obj` argument by one.

Then, pass the `person` object to the `increaseAge()` function:

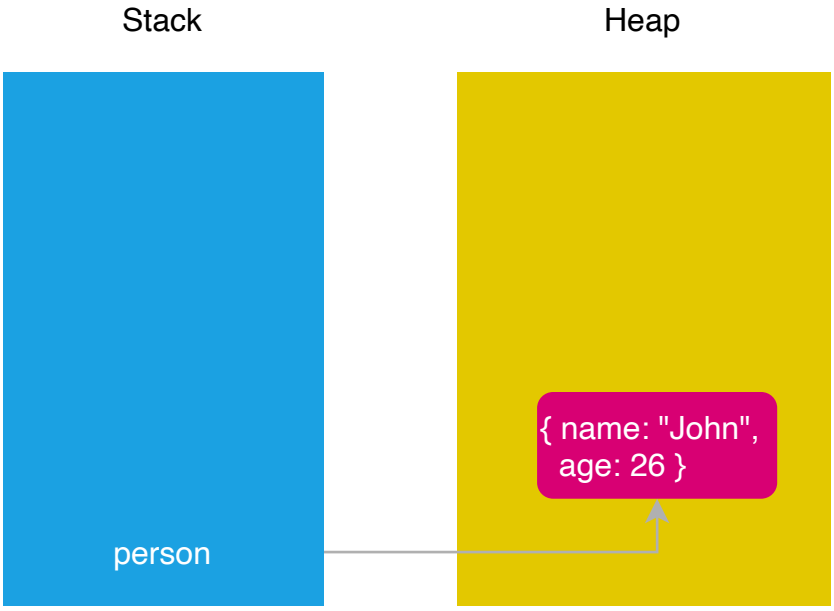


Internally, the JavaScript engine creates the `obj` reference and make this variable reference the same object that the `person` variable references.

After that, increase the `age` property by one inside the `increaseAge()` function via the `obj` variable



Finally, accessing the object via the person reference:



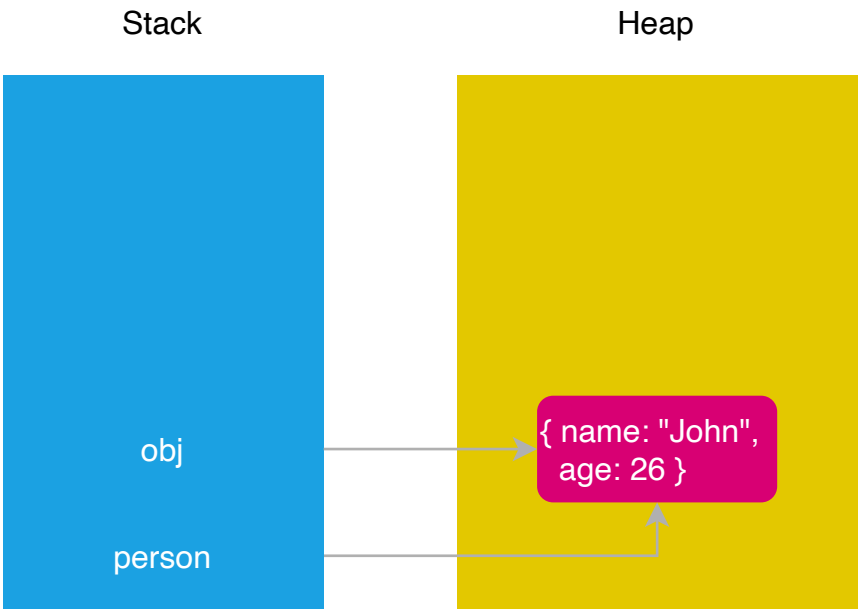
It seems that JavaScript passes an object by reference because the change to the object is reflected outside the function. However, this is not the case.

```
let person = {
  name: 'John',
  age: 25,
};

function increaseAge(obj) {
  obj.age += 1;
```

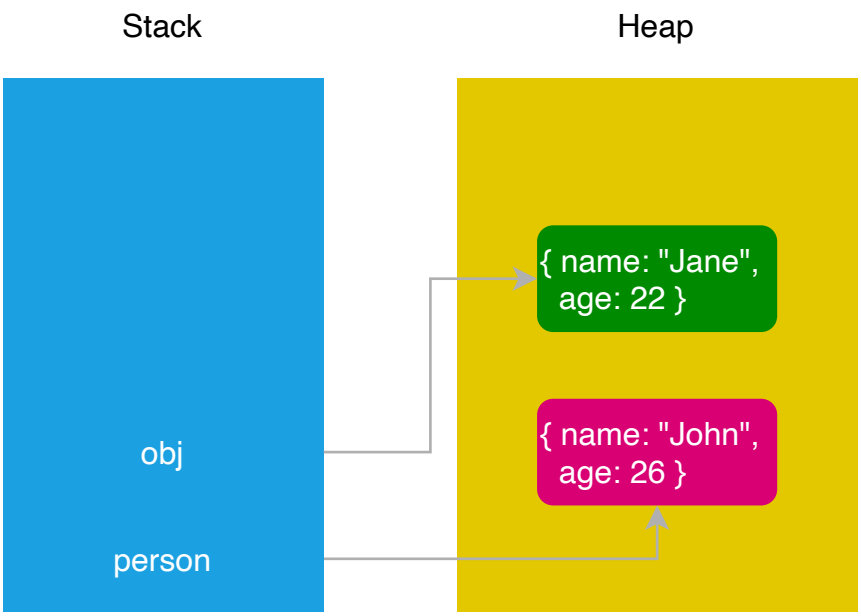
```
  // reference another object
  obj = { name: 'Jane', age: 22 };
}

increaseAge(person);

console.log(person);
```

Output:
{ name: 'John', age: 26 }

In this example, the `increaseAage()` function changes the `age` property via the `obj` argument:



and makes the `obj` reference another object:



However, the person reference still refers to the original object whose the age property changes to 26. In other words, the increaseAge() function doesn't change the person reference.

Introduction to the JavaScript recursive functions

A recursive function is a function that calls itself until it doesn't. And this technique is called recursion.

Suppose that you have a function called recurse(). The recurse() is a recursive function if it calls itself inside its body, like this:

```
function recurse() {
    // ...
    recurse();
    // ...
}
```

A recursive function always has a condition to stop calling itself. Otherwise, it will call itself indefinitely. So a recursive function typically looks like the following:

```
function recurse() {
```

```
    if(condition) {
        // stop calling itself
        //...
    } else {
        recurse();
    }
}
```

JavaScript recursive function examples

Let's take some examples of using recursive functions.

1) A simple JavaScript recursive function example

Suppose that you need to develop a function that counts down from a specified number to 1. For example, to count down from 3 to 1:

The following shows the `countDown()` function:

```
function countDown(fromNumber) {
    console.log(fromNumber);
}

countDown(3);
```

This `countDown(3)` shows only the number 3.

To count down from the number 3 to 1, you can:

1. show the number 3.
2. and call the `countDown(2)` that shows the number 2.
3. and call the `countDown(1)` that shows the number 1.

The following changes the `countDown()` to a recursive function:

```
function countDown(fromNumber) {
    console.log(fromNumber);
```

```
    countDown(fromNumber-1);
}

countDown(3);
```

Output:

```
Uncaught RangeError: Maximum call stack size ex
```

The count down will stop when the next number is zero. Therefore, you add an if condition as follows:

```
function countDown(fromNumber) {
    console.log(fromNumber);

    let nextNumber = fromNumber - 1;

    if (nextNumber > 0) {
        countDown(nextNumber);
    }
}
countDown(3);
```

Output:

```
3
2
1
```

For example, the following code will result in an error:

```
let newYearCountDown = countDown;
// somewhere in the code
countDown = null;
// the following function call will cause an error
newYearCountDown(10);
```

Error

```
Uncaught TypeError: countDown is not a function
```

How the script works:

- First, assign the `countDown` function name to the variable `newYearCountDown`.
- Second, set the `countDown` function reference to `null`.
- Third, call the `newYearCountDown` function.

The code causes an error because the body of the `countDown()` function references the `countDown` function name, which was set to `null` at the time of calling the function.

To fix it, you can use a named function expression as follows:

```
let countDown = function f(fromNumber) {
    console.log(fromNumber);

    let nextNumber = fromNumber - 1;

    if (nextNumber > 0) {
        f(nextNumber);
    }
}

let newYearCountDown = countDown;
countDown = null;
newYearCountDown(10);
```

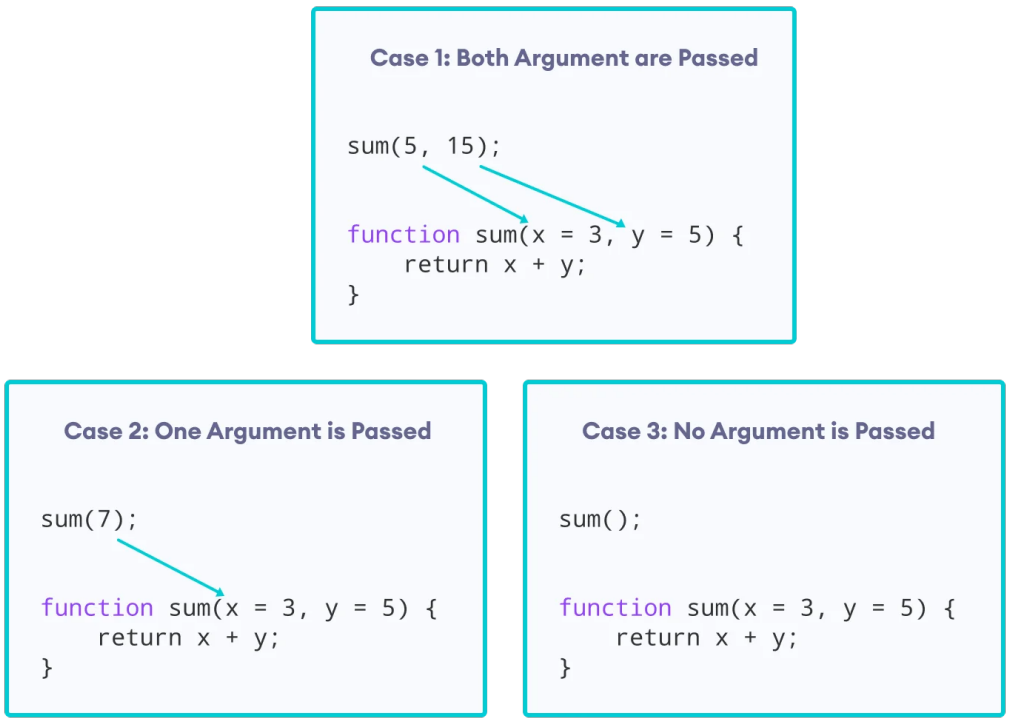
2) Calculate the sum of n natural numbers example

```
function sum(n) {
  if (n <= 1) {
    return n;
  }
  return n + sum(n - 1);
}
```

JavaScript Default Parameters

The concept of default parameters is a new feature introduced in the **ES6** version of JavaScript. This allows us to give default values to function parameters.

```
function sum(x = 3, y = 5) {

    // return sum
    return x + y;
}

console.log(sum(5, 15));  // 20
console.log(sum(7));      // 12
console.log(sum());       // 8
```



Example 1: Passing Parameter as Default Values

```
function sum(x = 1, y = x,  z = x + y) {
    console.log( x + y + z );
}

sum(); // 4
```

In the above program,

- The default value of x is **1**
- The default value of y is set to x parameter
- The default value of z is the sum of x and y

If you reference the parameter that has not been initialized yet, you will get an error. For example,

```
function sum( x = y, y = 1 ) {
    console.log( x + y);
}

sum();
```

Output:

```
ReferenceError: Cannot access 'y' before initialization
```

Example 2: Passing Function Value as Default Value

```
// using a function in default value expression

const sum = () => 15;

const calculate = function( x, y = x * sum() ) {
    return x + y;
}

const result = calculate(10);
console.log(result);          // 160
```

In the above program,

- **10** is passed to the calculate() function.

- `x` becomes `10`, and `y` becomes `150` (the sum function returns `15`).
- The result will be `160`.

Introduction to the JavaScript object methods

An object is a collection of key/value pairs or properties. When the value is a function, the property becomes a method. Typically, you use methods to describe the object behaviors.

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

person.greet = function () {
    console.log('Hello!');
}

person.greet();
console.log(person);
```

Output:
Hello!

In this example:

- First, use a function expression to define a function and assign it to the `greet` property of the `person` object.
- Then, call the method `greet()` method.

ES6 provides you with the concise method syntax that allows you to define a method for an object:

```
let person = {
    firstName: 'John',
    lastName: 'Doe',
```

```
    greet() {
        console.log('Hello, World!');
    }
};

person.greet();
```

The following example uses the `this` value in the `getFullName()` method:

```
let person = {
    firstName: 'John',
    lastName: 'Doe',
    greet: function () {
        console.log('Hello, World!');
    },
    getFullName: function () {
        return this.firstName + ' ' + this.lastName;
    }
};


console.log(person.getFullName());
```

Output:
'John Doe'

Introduction to JavaScript constructor functions

Technically speaking, a constructor function is a regular function with the following convention:

- The name of a constructor function starts with a capital letter like `Person`, `Document`, etc.
- A constructor function should be called only with the `new` operator.

The following example defines a constructor function called `Person`:

```
function Person(firstName, lastName) { //functional components
```

```
    this.firstName = firstName;
    this.lastName = lastName;
}
```

To create a new instance of the `Person`, you use the `new` operator:

```
let person = new Person('John','Doe');
```

Basically, the `new` operator does the following:

- Create a new empty object and assign it to the `this` variable.
- Assign the arguments `'John'` and `'Doe'` to the `firstName` and `lastName` properties of the object.
- Return the `this` value.

```
function Person(firstName, lastName) {
    // this = {};

    // add properties to this
    this.firstName = firstName;
    this.lastName = lastName;

    // return this;
}
```

However, the constructor function `Person` allows you to create multiple similar objects. For example:

```
let person1 = new Person('Jane','Doe')
let person2 = new Person('James','Smith')
```

Adding methods to JavaScript constructor functions

An object may have methods that manipulate its data. To add a method to an object created via the constructor function, you can use the `this` keyword. For example:

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;

    this.getFullName = function () {
        return this.firstName + " " + this.lastName;
    };
}
```

Now, you can create a new `Person` object and invoke the `getFullName()` method:

```
let person = new Person("John", "Doe");
console.log(person.getFullName());
```

Output:
John Doe

The following returns `undefined` because the `Person` constructor function is called like a regular function:

```
let person = Person("John", “Doe");
```

Output:

undefined

However, the following returns a reference to the `Person` function because it’s called with the `new` keyword:

```
let person = new Person("John", “Doe");
```

Output:

[Function: Person]

By using the `new.target`, you can force the callers of the constructor function to use the `new` keyword. Otherwise, you can throw an error like this:

```
function Person(firstName, lastName) {
```

```
    if (!new.target) {
        throw Error("Cannot be called without the new keyword");
    }

    this.firstName = firstName;
    this.lastName = lastName;
}
```

Alternatively, you can make the syntax more flexible by creating a new Person object if the users of the constructor function don't use the new keyword:

```
function Person(firstName, lastName) {
    if (!new.target) {
        return new Person(firstName, lastName);
    }

    this.firstName = firstName;
    this.lastName = lastName;
}

let person = Person("John", "Doe");

console.log(person.firstName);
```

This pattern is often used in JavaScript libraries and frameworks to make the syntax more flexible.

JavaScript Getter and Setter

In JavaScript, there are two kinds of object properties:

- Data properties
- Accessor properties

Data Property

```
const student = {

    // data property
```

```
    firstName: 'Monica';
};
```

Accessor Property

In JavaScript, accessor properties are methods that get or set the value of an object. For that, we use these two keywords:

- `get` - to define a getter method to get the property value
- `set` - to define a setter method to set the property value

JavaScript Getter

In JavaScript, getter methods are used to access the properties of an object. For example,

```
const student = {

    // data property
    firstName: 'Monica',

    // accessor property(getter)
    get getName() {
        return this.firstName;
    }
};

// accessing data property
console.log(student.firstName); // Monica

// accessing getter methods
console.log(student.getName); // Monica
```

```
// trying to access as a method
console.log(student.getName()); // error
```

In the above program, a getter method `getName()` is created to access the property of an object.

And also when accessing the value, we access the value as a property.

```
student.getName;
```

When you try to access the value as a method, an error occurs.

```
console.log(student.getName()); // error
```

JavaScript Setter

In JavaScript, setter methods are used to change the values of an object. For example,

```
const student = {
    firstName: 'Monica',

    //accessor property(setter)
    set changeName(newName) {
        this.firstName = newName;
    }
};

console.log(student.firstName); // Monica

// change(set) object property using a setter
student.changeName = 'Sarah';

console.log(student.firstName); // Sarah
```

In the above example, the setter method is used to change the value of an object.

```
set changeName(newName) {
    this.firstName = newName;
}
```

As shown in the above program, the value of firstName is Monica.

Then the value is changed to Sarah.

```
student.changeName = 'Sarah';
```

Introduction to JavaScript prototype

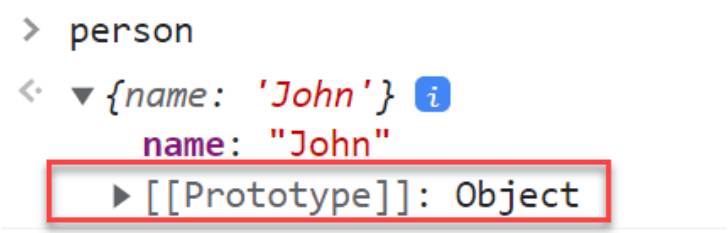
In JavaScript, objects can inherit features from one another via **prototypes**. Every object has its own property called prototype.

Because a prototype itself is also another object, the prototype has its own prototype. This creates a something called **prototype chain**. The prototype chain ends when a prototype has `null` for its own prototype.

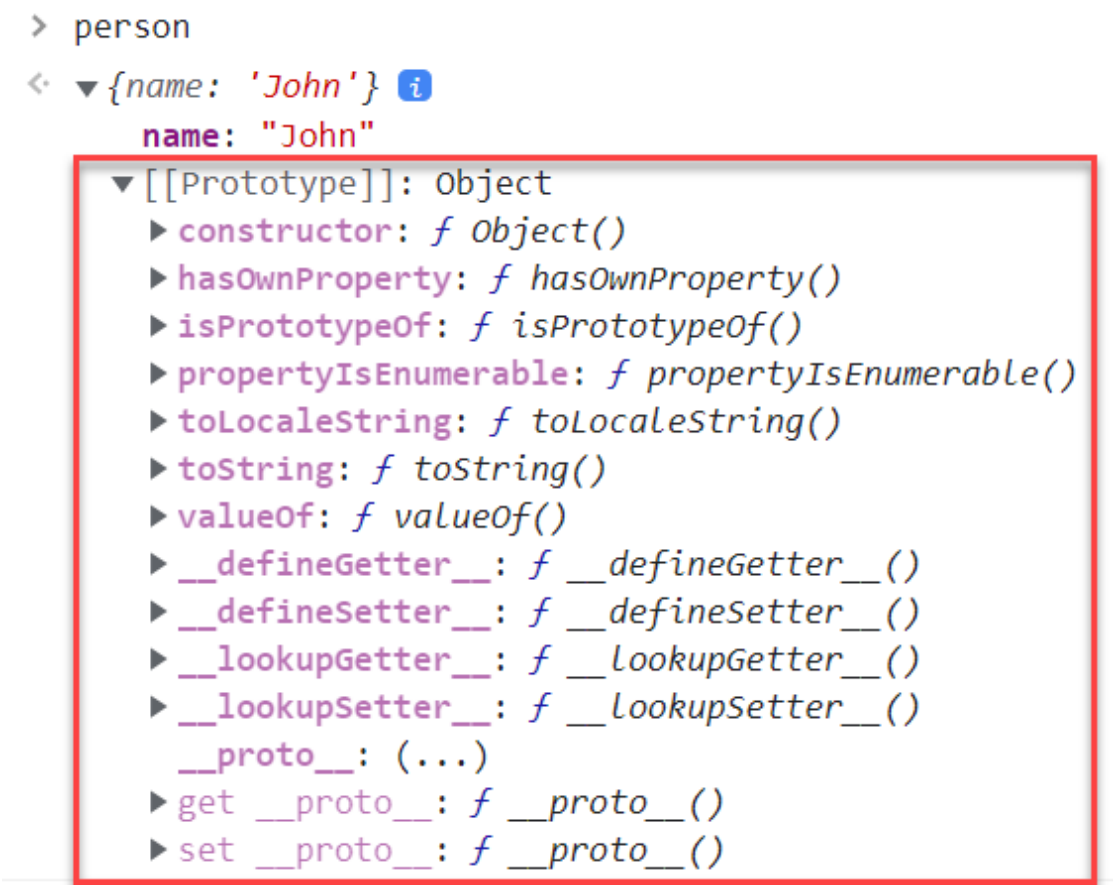
Suppose you have an object `person` with a property called `name`:

let person = {'name' : 'John'}

When examining the `person` object in the console, you'll find that the `person` object has a property called `prototype` denoted by the `[[Prototype]]`:



The prototype itself is an object with its own properties:



When you access a property of an object, if the object has that property, it'll return the property value. The following example accesses the `name` property of the `person` object:

```
> person.name
'John'
```

It returns the value of the `name` property as expected.

However, if you access a property that doesn't exist in an object, the JavaScript engine will search in the prototype of the object.

For example, you can call the `toString()` method of the `person` object like this:

```
> person.toString()
'[object Object]'
```

The `toString()` method returns the string representation of the `person` object. By default, it's `[object Object]` which is not obvious.

Since the person's prototype has the `toString()` method, JavaScript calls the `toString()` of the person's prototype object.


```
> Object.prototype
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __lookupGetter__: ƒ, …}
  constructor: ƒ Object()
  hasOwnProperty: ƒ hasOwnProperty()
  isPrototypeOf: ƒ isPrototypeOf()
  propertyIsEnumerable: ƒ propertyIsEnumerable()
  toLocaleString: ƒ toLocaleString()
  toString: ƒ toString()
  valueOf: ƒ valueOf()
  __defineGetter__: ƒ __defineGetter__()
  __defineSetter__: ƒ __defineSetter__()
  __lookupGetter__: ƒ __lookupGetter__()
  __lookupSetter__: ƒ __lookupSetter__()
  get __proto__: ƒ __proto__()
  set __proto__: ƒ __proto__()
```

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

Prototype Inheritance

In JavaScript, a prototype can be used to add properties and methods to a constructor function. And objects inherit properties and methods from a prototype. For example,

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// creating objects
const person1 = new Person();
const person2 = new Person();

// adding property to constructor function
Person.prototype.gender = 'male';

// prototype value of Person
console.log(Person.prototype);

// inheriting the property from prototype
console.log(person1.gender);
console.log(person2.gender);
```

Output:

```
{ gender: "male" }
male
male
```

Changing Prototype

If a prototype value is changed, then all the new objects will have the changed property value. All the previously created objects will have the previous value. For example,

```
// constructor function
function Person() {
    this.name = 'John'
}

// add a property
Person.prototype.age = 20;

// creating an object
const person1 = new Person();

console.log(person1.age); // 20

// changing the property value of prototype
Person.prototype = { age: 50 }

// creating new object
const person3 = new Person();

console.log(person3.age); // 50
console.log(person1.age); // 20
```

JavaScript Prototype Chaining

If an object tries to access the same property that is in the constructor function and the prototype object, the object takes the property from the constructor function. For example,

```
function Person() {
    this.name = 'John'
```

```
}

// adding property
Person.prototype.name = 'Peter';
Person.prototype.age = 23

const person1 = new Person();

console.log(person1.name); // John
console.log(person1.age); // 23
```

Another Example:

```
function Person () {
    this.name = 'John'
}

// adding a prototype
Person.prototype.age = 24;

// creating object
const person = new Person();

// accessing prototype property
console.log(person.__proto__);   // { age: 24 }
```

In the above example, a `person` object is used to access the prototype property using `__proto__`. However, `__proto__` has been deprecated and you should avoid using it.

JavaScript Form Validation Example

In this example, we are going to validate the name and password. The name can’t be empty and password can’t be less than 6 characters long.

```
<html>

<body>

<script>

function validateform(){

var name=document.myform.name.value;

var password=document.myform.password.value;


if (name==null || name==""){

  alert("Name can't be blank");

  return false;

}else if(password.length<6){

  alert("Password must be at least 6 characters long.");

  return false;

  }

}

</script>

<body>

<form name="myform" method="post" action="valid.html" onsubmit="return validateform()" >

Name: <input type="text" name="name"><br/>
```

Password: <input type="password" name="password">

<input type="submit" value="register">

</form>

</body>

</html>

JavaScript Retype Password Validation

```
<script type="text/javascript">
function matchpass(){
var firstpassword=document.f1.password.value;
var secondpassword=document.f1.password2.value;

if(firstpassword==secondpassword){
return true;
}
else{
alert("password must be same!");
return false;
}
}
</script>

<form name="f1" action="register.jsp" onsubmit="return matchpass()">
Password:<input type="password" name="password" /><br/>
Re-enter  Password:<input  type="password"  name="password2"/><br/>
<input type="submit">
</form>
```

JavaScript Number Validation

Let's validate the textfield for numeric value only. Here, we are using isNaN() function.

```
<script>
function validate(){
var num=document.myform.num.value;
if (isNaN(num)){
   document.getElementById("numloc").innerHTML="Enter Numeric value only";
  return false;
}else{
  return true;
  }
}
</script>
<form name="myform" onsubmit="return validate()" >
Number: <input type="text"  name="num"><span id="numloc"></span><br/>
<input type="submit" value="submit">
</form>
```

JavaScript email validation

We can validate the email by the help of JavaScript.

There are many criteria that need to be follow to validate the email id such as:

- email id must contain the @ and . character
- There must be at least one character before and after the @.
- There must be at least two characters after . (dot).

Let's see the simple example to validate the email field.

```
<script>
function validateemail()
{
```

```
var x=document.myform.email.value;
var atposition=x.indexOf("@");
var dotposition=x.lastIndexOf(".");
if (atposition<1 || dotposition<atposition+2 ||
 dotposition+2>=x.length){
            alert("Please enter a valid e-
mail address \n atpostion:"+atposition+"\n dotposition:"+dotpositio
n);
  return false;
  }
}
</script>
<body>
<form name="myform"  method="post" action="#" onsubmit="retu
rn validateemail();">
Email: <input type="text" name="email"><br/>

<input type="submit" value="register">
</form>
```

JavaScript - Form Validation

JavaScript provides a way to validate form's data on the client's computer before sending it to the web server. Form validation generally performs two functions.

- Basic Validation – First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.
- Data Format Validation – Secondly, the data that is entered must be checked for correct form and value. Your code must include appropriate logic to test correctness of data.

```
<html>
   <head>
      <title>Form Validation</title>
      <script type = "text/javascript">
         <!--
            // Form validation code will come here.
         //-->
```

```
      </script>
   </head>

   <body>
      <form action = "/cgi-bin/test.cgi" name =
"myForm" onsubmit = "return(validate());">
         <table cellspacing = "2" cellpadding = "2"
border = "1">

            <tr>
               <td align = "right">Name</td>
               <td><input type = "text" name = "Name"
/></td>
            </tr>

            <tr>
               <td align = "right">EMail</td>
               <td><input type = "text" name =
"EMail" /></td>
            </tr>

            <tr>
               <td align = "right">Zip Code</td>
               <td><input type = "text" name =
"Zip" /></td>
            </tr>

            <tr>
               <td align = "right">Country</td>
               <td>
                  <select name = "Country">
                     <option value = "-1"
selected>[choose yours]</option>
                     <option value = "1">USA</option>
                     <option value = "2">UK</option>
                     <option value = "3">INDIA</
option>
                  </select>
               </td>
            </tr>
```

```
            <tr>
               <td align = "right"></td>
               <td><input type = "submit" value =
"Submit" /></td>
            </tr>

         </table>
      </form>
   </body>
</html>
```

Basic Form Validation

First let us see how to do a basic form validation. In the above form, we are calling validate() to validate data when onsubmit event is occurring. The following code shows the implementation of this validate() function.

```
<script type = "text/javascript">
   <!--
      // Form validation code will come here.
      function validate() {

         if( document.myForm.Name.value == "" ) {
            alert( "Please provide your name!" );
            document.myForm.Name.focus() ;
            return false;
         }
         if( document.myForm.EMail.value == "" ) {
            alert( "Please provide your Email!" );
            document.myForm.EMail.focus() ;
            return false;
         }
         if( document.myForm.Zip.value == "" ||
isNaN( document.myForm.Zip.value ) ||
            document.myForm.Zip.value.length != 5 ) {
```

```
            alert( "Please provide a zip in the
format #####." );
            document.myForm.Zip.focus() ;
            return false;
         }
         if( document.myForm.Country.value == "-1" )
{
            alert( "Please provide your country!" );
            return false;
         }
         return( true );
      }
   //-->
</script>
```

Data Format Validation

Now we will see how we can validate our entered form data before submitting it to the web server.

The following example shows how to validate an entered email address. An email address must contain at least a '@' sign and a dot (.). Also, the '@' must not be the first character of the email address, and the last dot must at least be one character after the '@' sign.

```
<script type = "text/javascript">
   <!--
      function validateEmail() {
         var emailID = document.myForm.EMail.value;
         atpos = emailID.indexOf("@");
         dotpos = emailID.lastIndexOf(".");

         if (atpos < 1 || ( dotpos - atpos < 2 )) {
            alert("Please enter correct email ID")
            document.myForm.EMail.focus() ;
            return false;
         }
         return( true );
      }
   //-->
```

```
</script>
```