



# var vs let



```
function displayVar(){
```

```
    if(true){  
        var m = 20;  
    }  
    console.log(m);  
}
```

```
displayVar();
```

```
function displayLet(){
```

```
    if(true){  
        let m = 20;  
    }  
    console.log(m);  
}
```

```
displayLet();
```



# What is the output

```
var a = 20;  
function printA(){  
    console.log(a);  
    var a = 5;  
}  
printA();
```



# What is the output

```
var a = 20;  
    function printA(){  
        console.log(a);  
        a = 5;  
    }  
    printA();
```



# What is the output

```
function printA(){  
    a = 5;  
}  
printA();  
console.log(a);
```



# What is the output

```
function printA(){  
    var a = 5;  
}  
printA();  
console.log(a);
```



# const


- `const` is used to declare constants which cannot be overridden once it is declared .
- Constants defined with `const` follow the same scope rules as variables.
- Constants are block-scoped, much like variables defined using the `let` statement.
- The value of a constant cannot change through re-assignment, and it can't be redeclared.
- This declaration creates a constant whose scope can be either global or local to the block in which it is declared.
- Global constants do not become properties of the window object, unlike `var` variables.



# Softwares to be installed

- Chrome browser
- Visual studio code
- Nodejs
-





```
const tax_percentage = 13;  
    tax_percentage = 21;  
    console.log(tax_percentage);
```

Note: const are not hoisted



# Destructure

- It's a JavaScript expression that allows us to extract data from arrays, objects, maps and sets multiple at a time.
- ECMAScript 6 (ES2015) destructuring assignment allows you to extract individual items from arrays or objects and place them into variables using a shorthand syntax.
- It is a convenient way to extract values from data stored in (possibly nested) objects and arrays.
- Destructuring simply implies breaking down a complex structure into simpler parts.
- In JavaScript, this complex structure is usually an object or an array.



## Example

```
var employee = {  
  name : 'john',  
  age : 24,  
  salary : 100000,  
  married : true  
}  
var { name, age } = employee; // destructre  
  console.log(name); // john  
  console.log(age); // age  
var { name : empName, age : empAge} = employee; //destructure
```



## Array Destructure

```
[a, b, ...rest] = [10, 20, 30, 40, 50];
```

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

```
console.log(b);
```

```
console.log(rest);
```



```
const colorArray= ['red','blue','green'];
```

```
const [red,blue,green] = colorArray;
```

```
console.log(red)
```

```
console.log(blue)
```

```
console.log(green)
```



```
const colorArray= ['red','blue','green'];
```

```
const [red,blue] = colorArray;
```

```
console.log(red);
```

```
console.log(blue);
```



# Swap the variables

Let x = 10;

Let y = 20;



```
var x = 10;
```

```
var y = 20;
```

```
[y,x] = [x,y]; // es6 way destructure
```

```
console.log(x);
```

```
console.log(y);
```





# Default Parameters

- This is the ability to have your functions initialize parameters with default values even if the function call doesn't include them.
- Default function parameters allow formal parameters to be initialized with default values if no value or undefined is passed.

## Example:

```
function multiply(a,b=2){  
  return a * b;  
}  
multiply(10);
```



# Rest Parameters

- The rest operator is used to pass a variable number of arguments to a function, and it has to be the last one in the arguments list.
- If the name of the function argument starts with the three dots, the function will get the rest of the arguments in an array.
- The ES6 rest operator is represented by three dots (...).

## Example:

```
function total(name,...marks){  
  console.log(name);  
  for(let i =0; i<marks.length;i ++){  
    console.log(marks[i]);  
  }  
}  
var johnTotal = total("john",10,20,30,40);
```



# Spread Parameters

- The spread is closely related to rest parameters, because of ... (three dots) notation.
- It allows to split an array to single arguments which are passed to the function as separate arguments.
- The rest operator can turn a variable number of parameters into an array, the spread operator can do the opposite: turn an array into a list of values or function parameters.

## Example:

```
function total(x, y, z,p) {  
    return x + y + z +p;  
}
```

```
var args = [0, 1, 2,4];  
let result = total(...args);
```



## Spread arrays



### Example:

```
const originalArray = [1, 2, 3];
```

```
const cloneArray = [...originalArray];
```

# Spread objects



## Example:

```
const person = { firstName: "Alice", age: 30 };
const details = { lastName: "Smith", city: "New York" };

const mergedPerson = { ...person, ...details };
console.log(mergedPerson);
// Output: { firstName: "Alice", age: 30, lastName: "Smith", city: "New York"
}
```











# Template Literals

- Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.
- They were called "template strings" in prior editions of the ES2015 specification.
- Template literals are enclosed by the back-tick (``) character instead of double or single quotes.


## Example:

```
let category = "music";  
let id = 2112;  
let url = `http://apiserver/${category}/${id}`;
```




# Classes

- Classes are the feature that are newly introduced in the ES6.
- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance.
- The class syntax does not introduce a new object-oriented inheritance model to JavaScript.
- A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
- Classes can be included in the code either by declaring them or by using class expressions.



```
class BankAccount{
    constructor(accountName,accountNumber){
        this.accountName = accountName;
        this.accountNumber=accountNumber;
        this.balance = 2000;
    }
    withdraw(amount){
        this.balance = this.balance - amount;
    }
    deposit(amount){
        this.balance = this.balance + amount;
    }
}
```



```
var johnAccount = new BankAccount('john',200);  
console.log(johnAccount.balance);  
  
console.log(johnAccount.balance);
```



## Method overloading not possible

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

```
function add(x,y){  
  return x + y  
}  
console.log(add(10,20,30));
```



# Inheritance

```
class SavingsAccount extends Bankaccount{
  constructor(accpountno,accountname){
    super(accpountno,accountname);
    this.accountType = "savings";
  }
  withDraw(amount){
    if(amount>10000){
      console.log("sorry your daily withdrawl limit is over");
    }
    else{
      super.withDraw(amount);
    }
  }
}
```

```
class CurrentAccount extends Bankaccount{
  constructor(accpountno,accountname){
    super(accpountno,accountname);
    this.accountType = "current";
  }
  withdraw(amount){
    if(amount>40000){
      console.log("sorry your daily withdrawl limit is over");
    }
    else{
      super.withDraw(amount);
    }
  }
  drawDD(amount){
    console.log(`DD has been drawn for amount ${amount} for account ${this.accountname} `);
  }
}
```

# Class Hoisting



Classes are not hoisted.

```
var nimitAccount = new BankAccount('nimit',105);  
console.log(nimitAccount.balance);
```

```
class BankAccount{  
  
    constructor(accountName,accountNumber){  
        this.accountName = accountName;  
        this.accountNumber = accountNumber;  
        this.balance = 2000;  
    }  
}
```



# Static Methods



# Class Properties or getters and setters

- The get syntax binds an object property to a function that will be called when that property is looked up.
- The set syntax binds an object property to a function to be called when there is an attempt to set that property.

```
get fullName() {  
    return this.accountfName + " " + this.accountlName;  
}  
  
set fullName(value) {  
    var nameParts = value.split(' ');  
    this.accountfName = nameParts[0];  
    this.accountlName = nameParts[1];  
}
```

# Arrays New Methods

---



## Assume an Array with follow data

```
<script>
  let mobiles = [
    {
      "name": "Motorola",
      "model": "accessories phone",
      "price": 10000
    },
    {
      "name": "Lenovo",
      "model": "yoga tab",
      "price": 20000
    }
  ]
</script>
```



## forEach

- ES6 introduced the `Array.forEach()` method for looping through arrays.
- The `forEach()` method executes a provided function once for each array element.
- `forEach()` executes the provided callback once for each element present in the array in ascending order. It is not invoked for index properties that have been deleted or are uninitialized.

### Example:

```
mobiles.forEach(function(mobile){  
  console.log(mobile.name);  
  console.log(mobile.price);  
});
```



```
mobiles.forEach(loopArray)
```

```
function loopArray(mobile,mobileIndex,mobileArray){  
  console.log(mobile);  
  console.log(mobileArray);  
  console.log(mobileIndex);  
  // console.log(mobile.name);  
  // console.log(mobile.model);  
}
```



## filter

- The filter() method creates a new array with all elements that pass the test implemented by the provided function.
- filter() calls a provided callback function once for each element in an array, and constructs a new array of all the values for which callback returns a value that coerces to true.

### Example:

```
var filteredArray = mobiles.filter(function(mobile){  
  return mobile.price > 15000;  
});  
console.log(filteredArray);
```



## find

- The `Array.find()` method returns the value of the first element in an array that passes a given test.

### Example:

```
var findItem = mobiles.find(function(mobile){  
  return mobile.price > 15000;  
});  
console.log(findItem);
```






# Map

```
var array1 = [2,4,6,8]; // => [4,16,36,64];
```

```
var array2 = array1.map(function (item){  
    return item * item;  
});
```



```
var mobilePriceWithGST = mobiles.map(function(mobile){  
  let newMobile = {};  
  newMobile = mobile;  
  newMobile.price = mobile.price + mobile.price * 18/100;  
  return newMobile;  
})
```

# map

- Here, we transform our array from one array to another array.
- Map is a data collection type in which, data is stored in a form of pairs, which contains a unique key and value mapped to that key.
- 

```
var mobilePriceWithGST = mobiles.map(function(mobile){  
  let newMobile = {};  
  newMobile = Object.assign({},mobile);  
  newMobile.price = mobile.price + mobile.price * 18/100;  
  return newMobile;  
})
```



## reduce

- The `reduce()` method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.
- Just like `.map()`, `.reduce()` also runs a callback for each element of an array. What's different here is that `reduce` passes the result of this callback (the accumulator) from one array element to the other.
- The accumulator can be pretty much anything and must be instantiated or passed when calling `.reduce()`.

```
let numbers = [2,4,6,8];  
let finalTotal= numbers.reduce(function(total,number){  
  return total + number;  
})  
console.log(finalTotal);
```



## findIndex

```
let itemIndex = mobiles.findIndex(function(mobile,index,array){  
  return mobile.price > this;  
},16000);  
console.log(itemIndex);
```



# Arrow Functions

- Arrow functions were introduced with ES6 as a new syntax for writing JavaScript functions. They save developers time and simplify function scope.
- Arrow functions also called “fat arrow” functions, from CoffeeScript (a transcompiled language) are a more concise syntax for writing function expressions.
- They utilize a new token `=>`, that looks like a fat arrow.
- By using arrow functions, we avoid having to type the function keyword, return keyword (it’s implicit in arrow functions), and curly brackets.

## Example:

```
var numbers = [2,4,6,8];  
var mappedArray = numbers.reduce(  
  (total,number) => {return total + number;}  
)
```



## Variation of arrow functions

```
(total,number) => total + number;
```

```
num => num * num ;
```

```
(total,number) => {let x = 10; x = total+number};
```



## Function context


Every function has function context. When ever we refer **this** inside a function , to which it points to is called **function context**. By default function context points to object which invoked the function.






# Who is calling the call back is Window


```
let itemIndex = mobiles.findIndex(function(mobile,index,array){  
    return mobile.price > this;  
});  
console.log(itemIndex);
```



```
function add(x,y){  
  console.log(this); // window  
  return x + y;  
}  
add(10,20);
```




```
function add(x,y){  
    console.log(this); //{name:'kumar'}  
    return x + y;  
}  
boundAdd = add.bind({name:'kumar'});  
boundAdd(10,20);  
add(40,50); // check what is the context
```




```
let itemIndex = mobiles.findIndex(function(mobile,index,array){  
  return mobile.price > this;  
},16000);  
console.log(itemIndex);
```

# What is happening in background


```
var array = {  
  — findIndex: function(callback,abc){  
    },  
    fireFindIndex:function(){  
      if(abc == null){  
        findIndex();  
      }else  
      {  
        // lopp through all array items  
        boundCallBack = callback.bind(abc);  
        boundCallBack();  
      }  
    }  
  }
```



```
var simpleAdd = function (x,y){  
  console.log(this); //{name:'kumar'}  
  return x + y;  
}  
boundAdd = simpleAdd.bind({name:'kumar'});  
boundAdd(10,20);
```




```
var simpleAdd = (x,y)=>{  
  console.log(this);  
  return x + y;  
}  
boundAdd = simpleAdd.bind({name: 'kumar'});  
boundAdd(10,20);
```



```
var x = {  
  name: 'kumar',  
  displayName: function(){  
    console.log(this.name);  
  }  
}
```

```
x.displayName();
```







```
var x = {  
  name: 'kumar',  
  displayName: function(){  
    console.log(this.name);  
  }  
}
```

```
x.displayName(); // object  
yfn = x.displayName;
```

```
yfn();// window
```




```
var x = {
  name: 'kumar',
  displayName: function(){
    console.log(this.name);
    console.log(this);
  }
}
x.displayName(); // object
yfn = x.displayName;
// yfnBind = yfn.bind({name: 'abhishek'});
yfnBind();
```



```
var x = {
  name: 'kumar',
  displayName: ()=>{
    console.log(this.name);
    console.log(this);
  }
}
//x.displayName(); // object
yfn = x.displayName;
yfnBind = yfn.bind({name: 'abhishek'});
yfnBind();
```



Arrow functions do not bind their own `this`, instead, they inherit the one from the parent scope, which is called "lexical scoping". This makes arrow functions to be a great choice in some scenarios but a very bad one in others



```
var x = {  
  name: 'kumar',  
  displayName: ()=>{  
    console.log(this.name);  
  }  
}
```

```
x.displayName();// window object
```

```
var yFunction = x.displayName;  
var yBoundFunction = yFunction.bind({name: 'sarvanedhra'})  
yBoundFunction();// window object
```



```
myObject.myMethod() // this === myObject
```

```
myObject.myArrowFunction() // this === myObject
```

```
const myArrowFunction = myObject.myArrowFunction;  
myArrowFunction() // this === myObject
```



# Api calls

```
myObject = {  
  myMethod: function () {  
    helperObject.doSomethingAsync('superCool', () => {  
      console.log(this); // this === myObject  
    });  
  },  
};
```

# Modules

---





# Modules

- Module a concept to include functionality declared in one file within another.
- Typically, a developer creates an encapsulated library of code responsible for handling related tasks.
- That library can be referenced by applications or other modules.
- The ES6 Module Transpiler is a tool that takes your ES6 module and compiles it into ES5 compatible code in the CommonJS or AMD style.



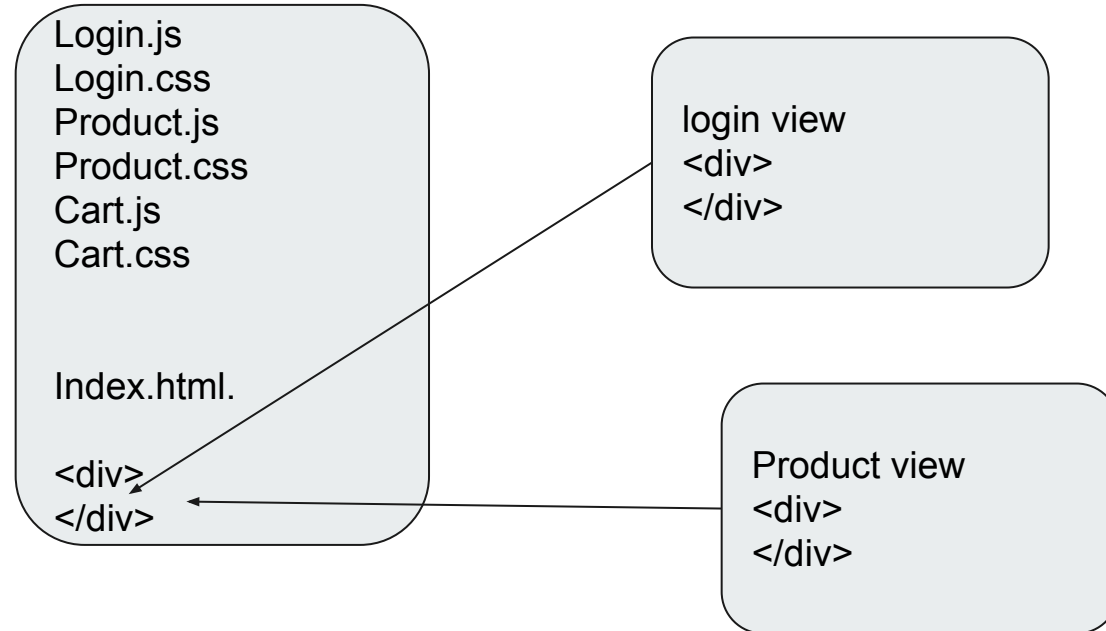
## Benefits

- Code can be split into smaller files of self-contained functionality.
- The same modules can be shared across any number of applications
- Code referencing a module understands it's a dependency
- Ideally, modules need never be examined by another developer, because they've has been proven to work.



# SPA - Single Page Application

- In complete application we will have only one html. Such applications are called as SPA.
- Angular follows SPA. mean in any application developed with Angular is going to have only one html.
- In typical asp.net applications we will have multiple html or aspx page where we redirect.
- In Non SPA each time we will move from page to another page complete page life cycle is going to start again






# Multi page applications

login.html  
login.js  
Login.css

product.html  
product.js  
product.css

cart.html  
cart.js  
cart.css



Before ES6 All the JS Files Required by application are loaded into the HTML page on the page load it self.

If an app needs only login.js for the app to start , it still loads all other 10 js files like util,log,register,productlist,card etc js files which are not required at start.

Disadvantages of above approach:

- Makes the page load slow
- Occupies browser memory



# Module in JS [import & export]

- Physical JS file with functionality with Import and export statements
- It should be loaded on demand.

## Export:

The export statement is used when creating JavaScript modules to export functions, objects, or primitive values from the module so they can be used by other programs with the import statement.

## Import:

The import statement is used to import bindings which are exported by another module.



# Export variable

```
export let variable_name;
```





# Export function

```
export function function_name() {  
  
    // Statements  
  
}
```



# Export Class

```
export class Class_Name {  
  
    constructor() {  
  
        // Statements  
  
    }  
  
}
```

# example

```
let num_set = [1, 2, 3, 4, 5];  
export default function hello() {  
  console.log("Hello World!");  
}  
  
class Greeting {  
  constructor(name) {  
    this.greeting = "Hello, " + name;  
  }  
}  
  
export { num_set, Greeting as Greet };
```



```
import Greeting as Greet from "./export.js";
```



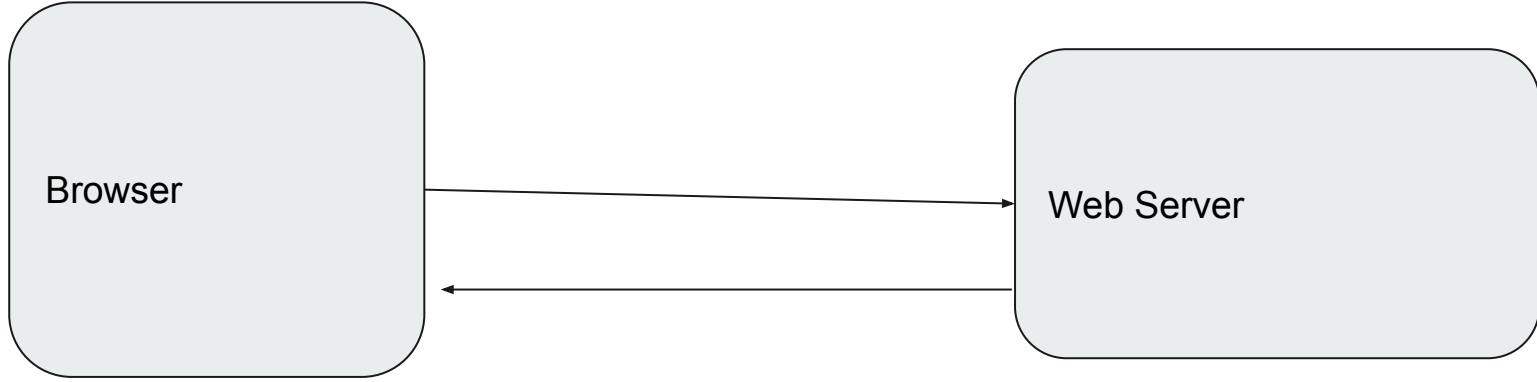
# Export Class

---

# Demo

# Web Server

---







# Web Servers

- 1) IIS
- 2) TomCat
- 3) Apache
- 4) XAMP
- 5) Http-server (from nodejs)



# Web server

Http-server : we will use node js npm package http-server to run the web server

- Install node.js and install the http-server
- `npm install http-server -g`
- Point to folder where we want to run the web server and execute the command
- `http-server` => it will list where the web server is running and open the browser from that path
-



# Origin is

- The Origin request header indicates where a fetch originates from. It doesn't include any path information, but only the server name.
- It is sent with CORS requests, as well as with POST requests. It is similar to the Referer header, but, unlike this header, it doesn't disclose the whole path.

<http://192.168.0.20:8080>

- Http
- Ip address
- Port number



# CORS

- Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.
- A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos.

## Example:

- Cross Origin Resource Sharing
- Origin : <http://ici:8765>
- <http://citi:8564>



# Bundling

- Combining multiple JS files into single JS file.
- A web essentials bundle file is a recipe for grouping, and usually compressing, a set of files of the same type to limit the number and the amount the data to be downloaded by the browser.
- **Web Essentials offers two bundling types:**
  - **.bundle :** for CSS and JS files.
  - **.sprite :** for images (PNG, JPG and GIF).



# Minification

- It is like compressing JS file into smaller memory.
- It will remove white spaces
- Replaces functions with dynamic single letter variables
- The main intent is to reduce the size of the file
- The time taken to load the 5 mb file is less than 10 mb file and your application loads faster