

ECMAScript 2015 (ES6)

Eman Fathi

The background features a large yellow circle on the right containing the letters 'JS' in bold black font. To its left are two smaller orange circles, each containing a white icon: the top one shows a code editor window with a '</>' symbol, and the bottom one shows a play button. The entire scene is set against a dark blue background with faint white lines.



Javascript Classes

Introducing JavaScript Classes

Unlike most formal **object-oriented** programming languages, JavaScript didn't support classes and classical inheritance as the primary way of defining similar and related objects when it was created.

When you're exploring **ECMAScript 6 classes**, it's helpful to understand the underlying mechanisms that classes use. ECMAScript 6 classes aren't the same as classes in other languages

Class-Like Structures in ECMAScript 5

ECMAScript 5 and earlier, JavaScript had no classes. The closest equivalent to a class was creating a constructor

```
function Person (name,age)
{   //This is constructor
    this.name=name;
    this.age=age;
    this.sayName=()=>{console.log(this.name)};
}
let person=new Person("Mona",30);
person.sayName(); //Mona
```

Class Declarations

Class declarations begin with the **class** keyword followed by the name of the class. The rest of the syntax looks similar to **concise methods** in object literals but doesn't require **commas** between the elements of the class.

```
class Person
{
  constructor(name, age)
  {
    this.name=name;
    this.age=age;
  }
  sayName()
  {console.log(this.name); }
};
let person=new Person("Mona",30);
```

Why Use the Class Syntax?

Class declarations, unlike function declarations, are not hoisted.

```
let person=new Person("Mona",30); //error
```

```
class Person
{
  constructor(name,age)
  {
    this.name=name;
    this.age=age;
  }
  sayName()
  {
    console.log(this.name);
  }
};
```

Calling the class constructor without new throws an error.

```
let person= Person("Mona",30); //Class constructor Person cannot be invoked  
without 'new'
```

Attempting to overwrite the class name within a class method throws an error.

```
class Foo  
{  
    constructor() {  
        Foo = "bar"; // throws an error when executed...  
    }  
}  
  
// but this is okay after the class declaration  
Foo = "baz";
```

Class Expressions

Classes and functions are similar in that they have two forms: declarations and expressions. Function and class declarations begin with an appropriate keyword (function or class, respectively) followed by an identifier. Functions have an expression form that doesn't require an identifier after function; similarly, classes have an expression form that doesn't require an identifier after class. These class expressions are designed to be used in variable declarations or passed into functions as arguments.

Basic Class Expression

```
let Person=Class
{
  constructor(name,age)
  {
    this.name=name;
    this.age=age;
  }
  sayName()
  { console.log(this.name); }
};

let person=new Person("Mona",30);
console.log(person instanceof Person);//true
console.log(typeof Person); //function
```

Accessor Properties

Although you should create own properties inside class constructors, classes allow you to define accessor properties on the prototype. To create a **getter**, use the keyword `get` followed by a space, followed by an identifier; to create a **setter**, do the same using the keyword `set`

```
class CustomHTMLElement {  
    constructor(element) { this.element = element; }  
    get html() { return this.element.innerHTML; }  
    set html(value) { this.element.innerHTML = value; }  
}  
  
let element=new CustomHTMLElement(document.getElementsByTagName("div")[0]);  
element.html; //will return innerHTML of the element  
element.html="test"; //will change innerHTML of the element
```

Static Members

ECMAScript 6 classes simplify the creation of **static** members by using the formal static annotation before the method or accessor property name. You can use the static keyword on any method or accessor property definition within a class. The only restriction is that you can't use static with the constructor method definition.

```
class Person
{
  constructor(name,age)
  {
    this.name=name;
    this.age=age;
  }
  static personInfo()
  {
    console.log(`this is person Class`);
  }
};
Person.personInfo();//this is person Class
```

Inheritance

Prior to ECMAScript 6, implementing inheritance with custom types was an extensive process. Classes make inheritance easier to implement by using the familiar **extends** keyword to specify the function from which the class should inherit.

The prototypes are automatically adjusted, and you can access the base class constructor by calling the **super()** method.

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
    getArea() {
        return this.length * this.width;
    }
}

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}

var square = new Square(3);
console.log(square.getArea()); // 9
console.log(square instanceof Square); // true
console.log(square instanceof Rectangle); // true
```

Notes on Using `super()`

Keep the following key points in mind when you're using `super()`:

- ✓ You can only use `super()` in a derived class constructor. If you try to use it in a non derived class (a class that doesn't use `extends`) or a function, it will throw an error.
- ✓ You must call `super()` before accessing `this` in the constructor. Because `super()` is responsible for initializing `this`, attempting to access `this` before calling `super()` results in an error.

Shadowing Class Methods

The methods on derived classes always shadow methods of the same name on the base class.

```
class Square extends Rectangle {  
    constructor(length) {  
        super(length, length);  
    }  
  
    getArea() {  
        return this.length * this.length;  
    }  
}
```

Of course, you can always decide to call the base class version of the method by using the **super.getArea()** method

```
class Square extends Rectangle
{
    constructor(length) {
        super(length, length);
    }

    getArea() {
        return super.getArea();
    }
}
```


Using new.target in Class Constructors

You can also use `new.target` in class constructors to determine how the class is being invoked.

```
class Rectangle {  
  constructor(length, width) {  
    console.log(new.target.name === 'Rectangle');  
    this.length = length;  
    this.width = width;  
  }  
}  
  
// new.target is Rectangle  
var obj = new Rectangle(3, 4); // outputs true
```

This code shows that `new.target` is equivalent to `Rectangle` when `new Rectangle(3, 4)` is called. Class constructors can't be called without `new`, so the `new.target` property is always defined inside class constructors. But the value may not always be the same.

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target.name === 'Rectangle');
    this.length = length;
    this.width = width;
  }
}

class Square extends Rectangle {
  constructor(length) {
    super(length, length)
  }
}

// new.target is Square
var obj = new Square(3); // outputs false
```

Square is calling the Rectangle constructor, so **new.target** is equal to Square when the Rectangle constructor is called. This is important because it gives each constructor the ability to alter its behaviour based on how it's being called. For instance, you can create an **abstract base** class (one that can't be instantiated directly) by using new.target

```
class Shape { // abstract base class
  constructor() {
    if (new.target.name === 'Shape') {
      throw new Error("This class cannot be instantiated directly.") }
  }
}

class Rectangle extends Shape {
  constructor(length, width) {
    super();
    this.length = length;
    this.width = width;
  }
}

var x = new Shape(); // throws an error
var y = new Rectangle(3, 4); // no error
console.log(y instanceof Shape); // true
```



New Array Features

New Array features

- ✓ **New static Array methods**
 - ✓ **Array.of()**
 - ✓ **Array.from().**
- ✓ **New Array.prototype methods**
 - ✓ **Array.prototype.fill**
 - ✓ **Array.prototype.find**
 - ✓ **Array.prototype.entries**
 - ✓ **Array.prototype.keys**
 - ✓ **Array.prototype.values**

New Static Array Methods

Array.of() Method:

One reason ECMAScript 6 added new creation methods to JavaScript was to help developers avoid a quirk of creating arrays with the Array constructor. The Array constructor actually behaves differently based on the type and number of arguments passed to it

```
let items = new Array(2);
console.log(items.length); // 2
console.log(items[0]); // undefined
console.log(items[1]); // undefined
items = new Array("2");
console.log(items.length); // 1
console.log(items[0]); // "2"
items = new Array(1, 2);
console.log(items.length); // 2
console.log(items[0]); // 1
console.log(items[1]); // 2
```

New Static Array Methods

ECMAScript 6 introduces `Array.of()` to solve this problem. The `Array.of()` method works similarly to the `Array` constructor but has no special case regarding a single numeric value. The `Array.of()` method always creates an array containing its arguments regardless of the number of arguments or the argument types.

```
let items = Array.of(1, 2);
console.log(items.length); // 2
console.log(items[0]); // 1
console.log(items[1]); // 2
items = Array.of(2);
console.log(items.length); // 1
console.log(items[0]); // 2
items = Array.of("2");
console.log(items.length); // 1
console.log(items[0]); // "2"
```

New Static Array Methods

Array.from() Method:

Converting nonarray objects into actual arrays has always been problem in JavaScript. For instance, if you have an arguments object (**which is array-like**) and want to use it like an array, you'd need to convert it

```
function summAll()  
{  
    let result=0;  
    //use arguments object as an array  
    for(let i=0;i<arguments.length;i++)  
    {result+=arguments[i];}  
    return result;  
}
```


New Static Array Methods

Array.from() Method:

```
function summAll()  
{  
    //convert arguments to an array and use it  
    let newArray=Array.from(arguments);  
    return eval(newArray.join("+"));  
}
```

Array.from has ability to take array conversion a step further

```
let numbers=[2,4,1,5];  
multiplyArray=Array.from(numbers, function(number)  
{  
    return number*number  
});  
//return [4, 16, 1, 25]
```

New Array.prototype Methods

fill() Method:

The fill method fills all the elements of an array from a start index to an end index with a static value(end index is excluded). The default values for start and end are respectively 0 and the length of the array.

```
[1, 2, 3].fill(4); // [4, 4, 4]
[1, 2, 3].fill(4, 1); // [1, 4, 4]
[1, 2, 3].fill(4, 1, 2); // [1, 4, 3]
[1, 2, 3].fill(4, 1, 1); // [1, 2, 3]
[1, 2, 3].fill(4, 3, 3); // [1, 2, 3]
[1, 2, 3].fill(4, -3, -2); // [4, 2, 3]
[1, 2, 3].fill(4, NaN, NaN); // [1, 2, 3]
[1, 2, 3].fill(4, 3, 5); // [1, 2, 3]
(new Array(3)).fill(4); // [4, 4, 4]
```

New Array.prototype Methods

find() and findIndex() Methods:

Searching for an array elements and return the first array element that satisfy the call back condition (and the same for findIndex)

```
let numbers = [25, 30, 35, 40, 45];  
console.log(numbers.find(n => n > 33)); // 35  
console.log(numbers.findIndex(n => n > 33)); // 2
```

```
[6, -5, 8].find(x => x < 0)//-5  
[6, 5, 8].find(x => x < 0)//undefined  
[6, -5, 8].findIndex(x => x < 0)//1  
[6, 5, 8].findIndex(x => x < 0)//-1
```

New Array.prototype Methods

entries(), keys() and values() Methods:

The result of each of the aforementioned methods is a sequence of values, but they are not returned as an array; they are revealed one by one, via an iterator.

```
Array.from([ 'a', 'b' ].keys())
//[ 0, 1 ]
Array.from([ 'a', 'b' ].values())
//[ 'a', 'b' ]
Array.from([ 'a', 'b' ].entries())
//[ [ 0, 'a' ], [ 1, 'b' ] ]
for(let [index,values] of numbers.entries())
{
    console.log(index,values);
}
```



Destructuring

Object Destructuring

```
let book={  
  Title:"ES",  
  Version:"6",  
  borrow:function(){console.log(this.Title,this.Version)}  
}  
let {Title,Version,borrow}=book;  
console.log(Version);// 6
```

the value of book.Title is stored in a variable called Title, and the value of book.Version is stored in a variable called Version. This syntax is the same as the **object literal initializer** shorthand.

Destructuring Assignment

A destructuring assignment expression evaluates to the right side of the expression **(after the =)**. That means you can use a destructuring assignment expression anywhere a value is expected.

```
var Title,Version,borrow;  
  {Title,Version,borrow}=book  //-->still Error  
//solution  
({Title,Version,borrow}=book)
```

Default Values

When you use a destructuring assignment statement and you specify a local variable with a property name that doesn't exist on the object, that local variable is assigned a value of **undefined**.

```
let book={  Title:"ES",
            Version:"6",
            borrow:function(){console.log(this.Title,this.Version)}
          }

let {Title,Version,borrow,printData}=book;
console.log(printData);// undefined
```

You can optionally define a default value to use when a specified property doesn't exist. To do so, insert an equal sign (=) after the property name and specify the **default** value.

```
let {Title,Version,borrow,printData=null}=book;
console.log(print);// null
```


Assigning to Different Local Variable Names

Up to this point, each **destructuring** assignment example has used the object property name as the local variable name; for example, the value of **book.Title** was stored in a **Title variable**. That works well when you want to use the same name, but what if you don't? ECMAScript 6 has an extended syntax that allows you to assign to a local variable with a different name, and that syntax looks like the object literal non-shorthand property initializer syntax.

```
let {X,Y,Z}=book; //not working because we name variables with different names
```

```
let {Title:X,Version:Y,borrow:Z,printData:W="BOOK"}=book;  
console.log(X,Y,Z); //ES 6 f (){console.log(this.Title,this.Version)}
```

Array Destructuring

Array destructuring syntax is very similar to object destructuring: it just uses array literal syntax instead of object literal syntax. The destructuring operates on **positions** within an array rather than the named **properties** that are available in objects.

```
var salaries = [32000, 5000, 75000];  
var [low,medium,high]=salaries;  
console.log(medium); //5000
```

```
//default values  
var salaries = [32000, 5000];  
var [low,medium,high=9000]=salaries;  
console.log(high); //9000
```

Nested Array and rest item

//Nesting Arrays

```
var salaries=[1000,2000,[3000,4000]];
[x,y,[a,b,c=6000]]=salaries;
console.log(x,y,a,c); //1000,2000,3000,6000
```

//Rest items

```
var salaries = [32000, 5000, 75000,9000];
var [low,...remaining]=salaries;
console.log(remaining);//[5000, 75000, 9000]
```

```
function setCookie(name,value,options)
{
    options=(options===undefined?{}:options);
    var secure=(options.secure===undefined?"true":options.secure)
    var path=(options.path===undefined?"true":options.path)
    var expires=(options.expires===undefined?"true":options.expires)
    //rest of code
}

//calling
setCookie("itiEs",'version6',{secure:false,path:'d://iti',expires:60000});

//now using destructuring
function setCookie(name,value,{secure=false,path='/',expires=new Date(Date.now() +
360000000)}={})
{ //function code body }

//calling
setCookie("itiEs",'version6',{secure:false,path:'d://iti',expires:60000});
```



Modules

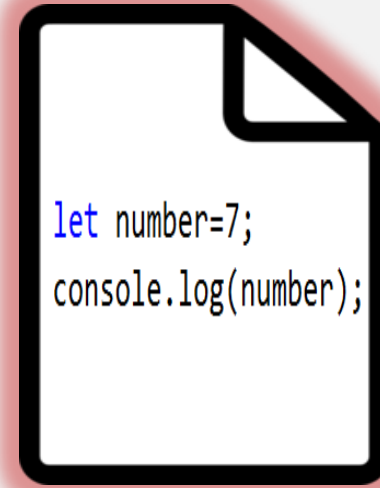
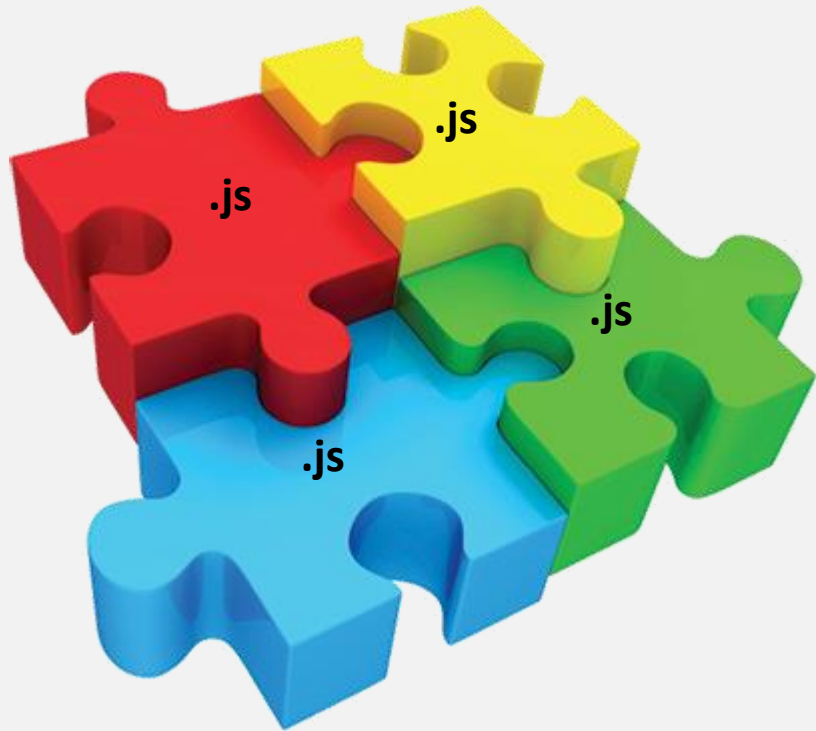
Encapsulating Code with Modules

JavaScript's

JavaScript's “shared everything” approach to loading code is one of the most error prone and confusing aspects of the language.

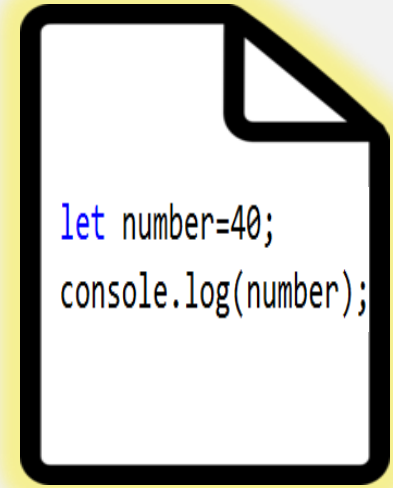
Other languages use concepts such as packages to define code scope, but before ECMAScript 6, everything defined in every JavaScript file of an application shared one global scope. As web applications became more complex and started using even more JavaScript code, that approach caused problems, such as naming collisions and security concerns. One goal of ECMAScript 6 was to solve the scope problem and bring some order to JavaScript applications. That's where modules come in.

What Are Modules?



```
let number=7;  
console.log(number);
```

Red.js



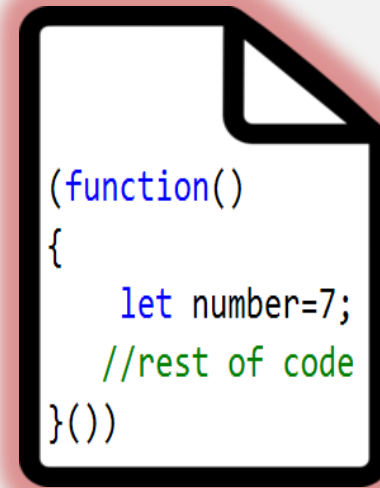
```
let number=40;  
console.log(number);
```

Yellow.js

```
<script src="yellow.js"></script>  
<script src="red.js"></script>  
<script>  
    console.log(number); // 7  
</script>
```

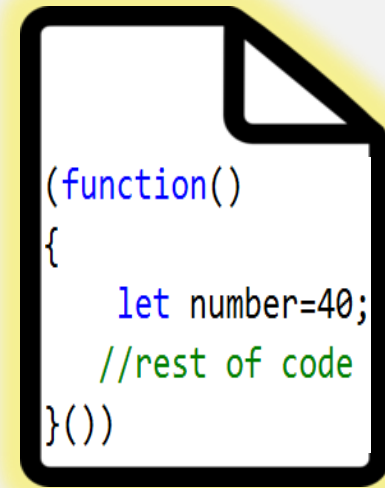
IIFE (Immediately Invoked Function Expression)

IIFE (Immediately Invoked Function Expression) is a JavaScript **function** that runs as soon as it is defined.

A document icon with a red glow, representing a JavaScript file named Red.js. It contains the following code:

```
(function()  
{  
    let number=7;  
    //rest of code  
})();
```

Red.js

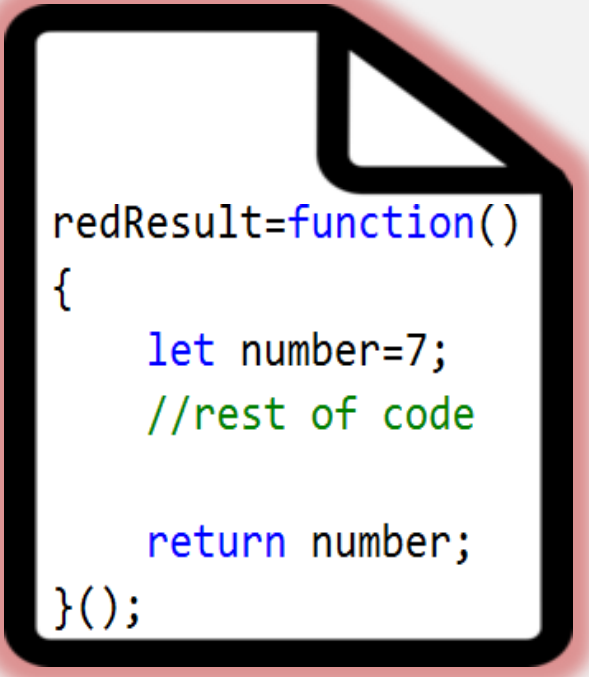
A document icon with a yellow glow, representing a JavaScript file named Yellow.js. It contains the following code:

```
(function()  
{  
    let number=40;  
    //rest of code  
})();
```

Yellow.js

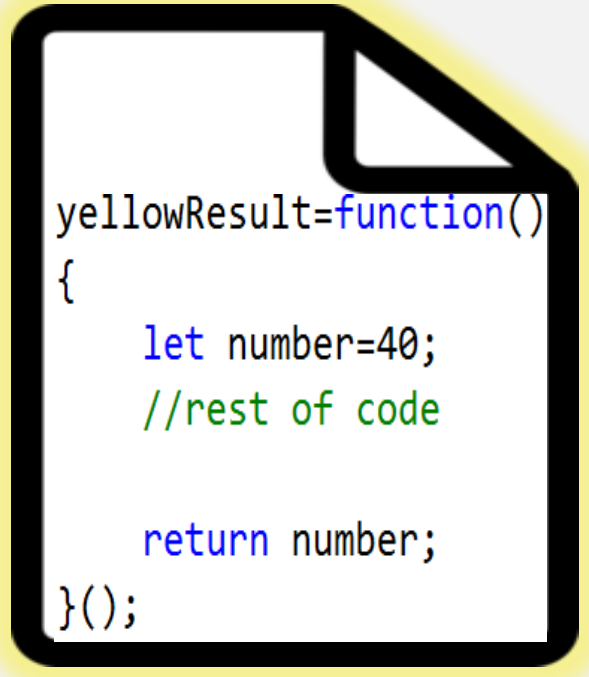
```
console.log(number); //throw Error
```


IIFE (Immediately Invoked Function Expression)

A document icon with a red glow and a black border, representing a file named Red.js.

```
redResult=function()  
{  
    let number=7;  
    //rest of code  
  
    return number;  
}();
```

Red.js

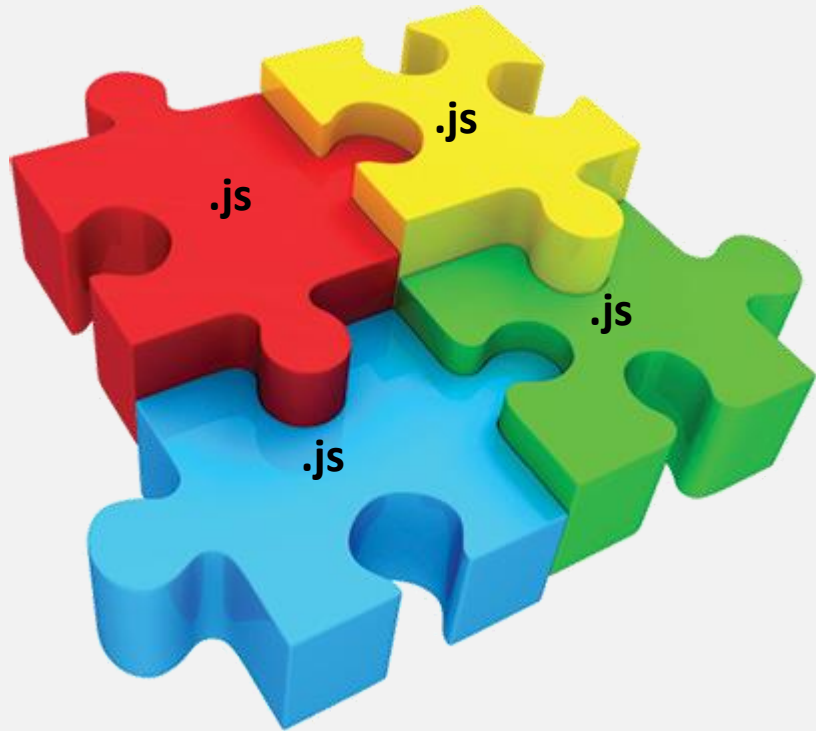
A document icon with a yellow glow and a black border, representing a file named Yellow.js.

```
yellowResult=function()  
{  
    let number=40;  
    //rest of code  
  
    return number;  
}();
```

Yellow.js

```
redResult; // 7  
yellowResult; // 40
```

What Are Modules?



```
let number =7;  
export {number};
```

Red.js

```
let number =40;  
export {number};
```

Yellow.js

```
import {number} from './red.js'  
import {number as number1} from './yellow.js'
```

Differences between Modules and scripts

- 1- ES6 modules are running in **strict mode** by default
- 2- the value of **this** is undefined
- 3- variables are **local** to the module
- 4- ES6 modules are loaded and executed **asynchronously**

Basic Exporting

```
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;
// export function
export function sum(num1, num2) {return num1 + num1; }
// export class
export class Rectangle {constructor(length, width) {
    this.length = length;
    this.width = width;} }
function subtract(num1, num2) {return num1 - num2;} // this function is private to the
module
function multiply(num1, num2) { return num1 * num2;} // define a function...
export multiply; // ...and then export it later
```

Basic Importing

When you have a module with exports, you can access the functionality in another module by using the **import** keyword. The two parts of an import statement are the identifiers you're importing and the module from which those identifiers should be imported.

```
import { identifier1, identifier2 } from "./example.js";
```

The curly braces after import indicate the **bindings** to import from a given module. The keyword from indicates the module from which to import the given binding. The module is specified by a string representing the path to the module (called the module specifier). Browsers use the same path format you might pass to the <script> element, which means you must include a file extension.

Importing a Single Binding

Suppose that the example in “**Basic Exporting**” is in a module with the filename `example.js`. You can import and use bindings from that module in a number of ways.

```
// import just one
import { sum } from "./example.js";
console.log(sum(1, 2)); // 3
sum = 1; // throws an error
```

If you try to assign a new value to `sum`, the result is an error because you can't reassign imported bindings.

Importing Multiple Bindings

If you want to import multiple bindings from the example module

```
// import multiple
import { sum, multiply, magicNumber } from "./example.js";
console.log(sum(1, magicNumber)); // 8
console.log(multiply(1, 2)); // 2
```

Here, three bindings are imported from the example module: sum, multiply, and magic Number. They are then used as though they were locally defined.

Importing an Entire Module`

A special case allows you to import the entire module as a single object. All exports are then available on that object as properties.

```
// import everything
import * as example from "./example.js";
console.log(example.sum(1,example.magicNumber)); // 8
console.log(example.multiply(1, 2)); // 2
```

In this code, all exported bindings in example.js are loaded into an object called example. The named exports (the sum() function, the multiple() function, and magic Number) are then accessible as properties on example.

This import format is called a **namespace** import because the example object doesn't exist inside the example.js file and is instead created to be used as a namespace object for all the exported members of example.js. However, keep in mind that no matter how many times you use a module in import statements, the module will execute only once. After the code to import the module executes, the instantiated module is kept in memory and reused whenever another import statement references it.

```
import { sum } from "./example.js";  
import { multiply } from "./example.js";  
import { magicNumber } from "./example.js";
```

Module Syntax Limitations

An important limitation of both export and import is that they must be used outside other statements and functions. For instance, this code will give a syntax error:

```
if (flag) {  
    export flag; // syntax error  
}
```

Similarly, you can't use import inside a statement; you can only use it at the top-level. That means this code also gives a syntax error:

```
function tryImport()  
{  
    import flag from "./example.js"; // syntax error  
}
```

A Subtle Quirk of Imported Bindings

ECMAScript 6's import statements create read-only bindings to variables, functions, and classes rather than simply referencing the original bindings like normal variables. Even though the module that imports the binding can't change the binding's value, the module that exports that identifier can

```
export let name = "Nicholas";  
export function setName(newName) {  
    name = newName;  
}
```

When you import these two bindings, the setName() function can change the value of name:

```
import { name, setName } from "./example.js";  
console.log(name); // "Nicholas"  
setName("Greg");  
console.log(name); // "Greg"  
name = "Nicholas"; // throws an error
```

The call to setName("Greg") goes back into the module from which setName() was exported and executes there, setting name to "Greg" instead.

Note that this change is automatically reflected on the imported name binding.

The reason is that name is the local name for the exported name identifier. The name used in this code and the name used in the module being imported from aren't the same.

Renaming Exports and Imports

Sometimes, you may not want to use the original name of a variable, function, or class you've imported from a module. Fortunately, you can change the name of an export during the export and during the import.

```
function sum(num1, num2) {  
    return num1 + num2;  
}
```

```
export { sum as add };
```

That means when another module wants to import this

function, it will have to use the name add:

```
import { add } from "./example.js";
```

If the module importing the function wants to use a different name, it can also use as:

```
import { add as sum } from "./example.js";
```

```
console.log(typeof add); // "undefined"
```

```
console.log(sum(1, 2));
```

Default Values in Modules

Exporting Default Values

```
export default function(num1, num2) {  
  return num1 + num2;  
}
```

This module exports a function as its default value. The default keyword indicates that this is a default export. The function doesn't require a name because the module represents the function.

You can also specify an identifier as the default export by placing it after export default, like this:

```
function sum(num1, num2) {  
  return num1 + num2;  
}  
export default sum;
```

A third way to specify an identifier as the default export is by using the renaming syntax as follows:

```
function sum(num1, num2) {  
    return num1 + num2;  
}  
export { sum as default };
```

The identifier default has special meaning in a renaming export and indicates a value should be the default for the module.

Importing Default Values

```
// import the default
import sum from "./example.js";
console.log(sum(1, 2)); // 3
```

This import statement imports the default from the module `example.js`. Note that no curly braces are used, unlike what you'd see in a non-default import. The local name `sum` is used to represent whatever default function the module exports.

For modules that export a default and one or more non-default bindings, you can import all exported bindings using one statement.

```
export let color = "red";
export default function(num1, num2) {
    return num1 + num2;
}
```


You can import color and the default function using the following import statement:

```
import sum, { color } from "./example.js";  
console.log(sum(1, 2)); // 3  
console.log(color); // "red"
```

The comma separates the default local name from the non-defaults, which are also surrounded by curly braces. Keep in mind that the default must come before the non-defaults in the import statement.

```
import { default as sum, color } from "./example.js";  
console.log(sum(1, 2)); // 3  
console.log(color); // "red"
```

In this code, the default export (default) is renamed to sum and the additional color export is also imported. This example is otherwise equivalent to the preceding example.

Re-exporting a Binding

Eventually, you may want to re-export something that your module has imported. For instance, perhaps you're creating a library from several small modules.

```
import { sum } from "./example.js";  
export { sum }
```

Although that works, a single statement can also do the same task:

```
export { sum } from "./example.js";
```

Of course, you can also export a different name for the same value:

```
export { sum as add } from "./example.js";
```

Here, `sum` is imported from `example.js` and then exported as `add`.

If you want to export everything from another module, you can use the `*` pattern:

```
export * from "./example.js";
```

By exporting everything, you're including the default as well as any named exports, which may affect what you can export from your module.

Browser Module Specifier Resolution

All examples to this point in the chapter have used a relative path (as in the string `./example.js`) for the module specifier. Browsers require module specifiers to be in one of the following formats:

- ✓ Begin with `/` to resolve from the root directory
- ✓ Begin with `./` to resolve from the current directory
- ✓ Begin with `../` to resolve from the parent directory
- ✓ URL format

```
// imports from https://www.example.com/modules/example1.js
```

```
import { first } from "./example1.js";
```

```
// imports from https://www.example.com/example2.js
```

```
import { second } from "../example2.js";
```

```
// imports from https://www.example.com/example3.js
```

```
import { third } from "/example3.js";
```

```
// imports from https://www2.example.com/example4.js
```

```
import { fourth } from "https://www2.example.com/example4.js";
```

ECMAScript 2016 (ES7) new Features

- ✓ `Array.prototype.includes()`
- ✓ Exponentiation operator

ECMAScript 2017(ES8) new Features

- ✓ Object.values/Object.entries
- ✓ String padding
- ✓ Object.getOwnPropertyDescriptors
- ✓ Trailing commas in function parameter lists and calls
- ✓ Async Functions

ECMAScript 2018(ES9) new Features

- ✓ Lifting template literal restriction
- ✓ Asynchronous iterators
- ✓ Promise.prototype.finally library
- ✓ Object Rest/spread properties
- ✓ Unicode property escapes in regular expressions
- ✓ Async Functions