

ECMAScript 2015 (ES6)

Eman Fathi

The image features a large, bright yellow circle on the right side containing the letters 'JS' in a bold, black, sans-serif font. The background is a dark blue-grey color with several overlapping circles in shades of red, orange, and yellow. Some of these circles contain icons: a code editor window with a '</>' symbol, and a play button icon. The overall design is modern and tech-oriented.

ECMAScript Releases



JavaScript is born
as LiveScript

1997

ES3 comes out and
IE5 is all the rage

2000

ES5 comes out and
standard JSON

2015

ES7/ECMAScript2016
comes out

2017

1995 **ECMAScript** standard
is established

1999

XMLHttpRequest,
a.k.a. AJAX,
gains popularity

2009

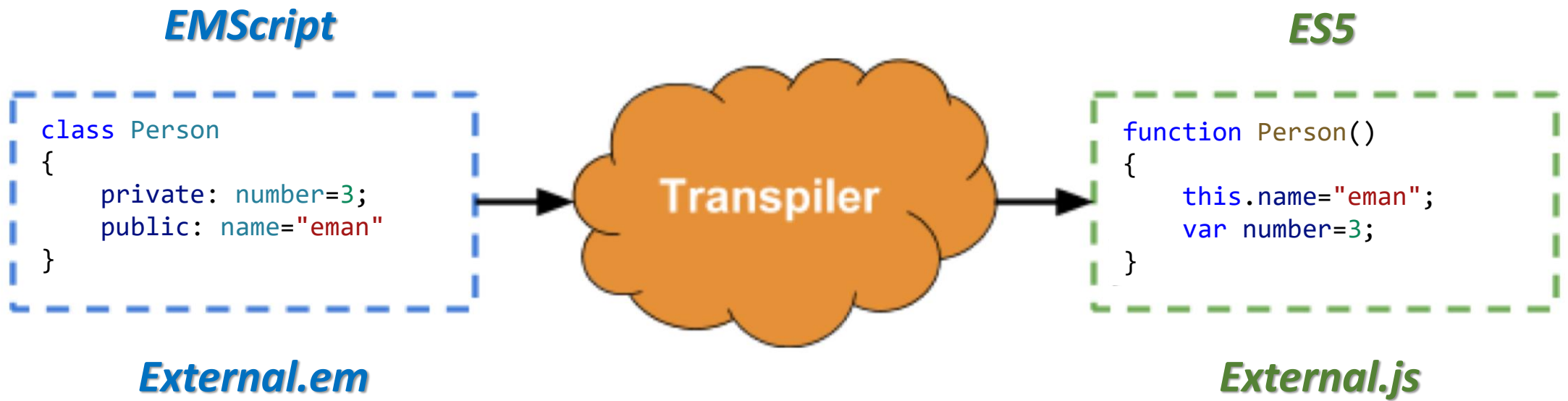
ES6/ECMAScript2015
comes out

2016

ES.Next



Transpilers



Transpilers

BABEL



TypeScript



Dart



ADVANTAGES OF JAVASCRIPT

ES6

MODULES

ARROWS

ENHANCED
OBJECT LITERALS

CLASSES

BLOCK
SCOPING

PROMISES



Sort by **Engine types**Show obsolete platforms ☐Show unstable platforms ☒

V8



SpiderMonkey



JavaScriptCore



Chakra



Carakan



KJS



Other

Minor difference (1 point)

Small feature (2 points)

Medium feature (4 points)

Large feature (8 points)

Compilers/polyfills

Desktop browsers

Feature name

Current
browser

Traceur

Babel 6
+
core-js^[2]Babel 7
+
core-js^[2]

Closure

Type-
Script
+
core-jses6-
shimKong
4.14^[3]

IE 11

Edge
15Edge
16Edge 17
PreviewFF 52
ESR

FF 57

FF 58

FF 59
BetaFF 60
NightlyCH 64,
OP 51^[1]CH 65,
OP 52^[1]CH 66,
OP 53^[1]

Optimisation

proper tail calls (tail call optimisation)

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

Syntax

default function parameters

7/7

4/7

4/7

4/7

5/7

5/7

0/7

0/7

0/7

7/7

7/7

7/7

6/7

7/7

7/7

7/7

7/7

7/7

7/7

7/7

7/7

rest parameters

5/5

4/5

3/5

3/5

2/5

4/5

0/5

0/5

0/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

spread (...) operator

15/15

15/15

13/15

13/15

12/15

4/15

0/15

0/15

0/15

15/15

15/15

15/15

15/15

15/15

15/15

15/15

15/15

15/15

15/15

15/15

15/15

object literal extensions

6/6

6/6

6/6

6/6

5/6

6/6

0/6

0/6

0/6

6/6

6/6

6/6

6/6

6/6

6/6

6/6

6/6

6/6

6/6

6/6

6/6

for..of loops

9/9

9/9

9/9

9/9

6/9

3/9

0/9

0/9

0/9

9/9

9/9

9/9

7/9

9/9

9/9

9/9

9/9

9/9

9/9

9/9

9/9

octal and binary literals

4/4

2/4

4/4

4/4

4/4

4/4

2/4

0/4

0/4

4/4

4/4

4/4

4/4

4/4

4/4

4/4

4/4

4/4

4/4

4/4

4/4

template literals

5/5

4/5

4/5

4/5

3/5

3/5

0/5

0/5

0/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

RegExp "y" and "u" flags

5/5

3/5

3/5

3/5

0/5

0/5

0/5

0/5

0/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

5/5

destructuring declarations

22/22

20/22

21/22

21/22

20/22

15/22

0/22

0/22

0/22

22/22

22/22

22/22

21/22

22/22

22/22

22/22

22/22

22/22

22/22

22/22

22/22

destructuring assignment

24/24

23/24

24/24

24/24

21/24

19/24

0/24

0/24

0/24

24/24

24/24

24/24

23/24

24/24

24/24

24/24

24/24

24/24

24/24

24/24

24/24

destructuring parameters

24/24

19/24

21/24

21/24

19/24

16/24

0/24

0/24

0/24

23/24

23/24

23/24

21/24

24/24

24/24

24/24

24/24

24/24

24/24

24/24

24/24

Unicode code point escapes

2/2

1/2

1/2

1/2

1/2

1/2

0/2

0/2

0/2

2/2

2/2

2/2

1/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

new.target

2/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

0/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

2/2

Bindings

const

16/16

14/16

14/16

14/16

14/16

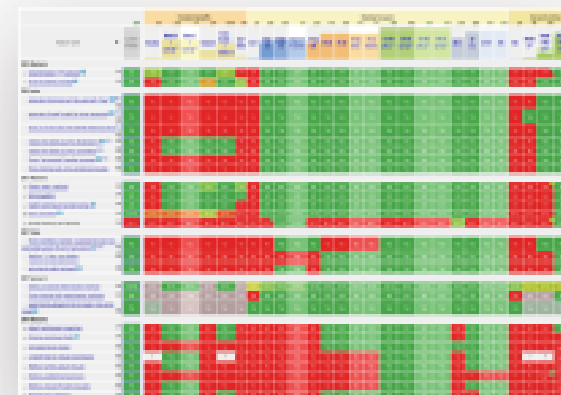
14/16

0/16

2/16

👤 **During development:** Simply use the latest Google Chrome!

🚀 **During production:** Use Babel to transpile and polyfill your code (converting back to ES5 to ensure browser compatibility for all users).



The image shows a screenshot of the Babel compatibility table, which is a grid where rows represent different ES6+ features and columns represent various web browsers. Green cells indicate that a feature is supported by a browser, while red cells indicate it is not. The table is used to check the compatibility of modern JavaScript code with different browsers before deployment.

<http://kangax.github.io/compat-table>

ES5

- 👉 Fully supported in all browsers (down to IE 9 from 2011);
- 👉 Ready to be used today 👍

ES6/ES2015

↓

ES2020

- 👉 ES6+: Well supported in all **modern** browsers;
- 👉 No support in **older** browsers;
- 👉 Can use **most** features in production with transpiling and polyfilling 🤖

ES2021 – ∞

- 👉 **ESNext**: Future versions of the language (new feature proposals that reach Stage 4);
- 👉 Can already use **some** features in production with transpiling and polyfilling.

BABEL

ECMAScript 2015 (ES6)

- Since ECMAScript 2015 (also known as ES6) was released, it has introduced a huge set of new features. They include arrow functions, sets, maps, classes and destructuring, and much more. In many ways, ES2015 is almost like learning a new version of JavaScript.
- Ecma Technical Committee governs the ECMA specification. They decided to release a new version of ECMAScript every year starting in 2015. A yearly update means no more big releases like ES6.



Top 10 best ES6 Features

- ✓ Block scope constructs with let and const
- ✓ Template Literals
- ✓ Multi-Line strings
- ✓ Array New Features
- ✓ Arrow functions
- ✓ Default parameters
- ✓ Enhanced object literals
- ✓ Classes
- ✓ Destructuring Assignment
- ✓ Modules
- ✓ Promises





*Let **and** Const Block Scoping*

Block Level Declaration with let and const

Variable declarations using **var** are treated as if they're at the top of the function (or in the global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called **hoisting**.

Misunderstanding hoisting unique behaviour can end up causing bugs. For this reason, ECMAScript 6 introduces **block-level** scoping options to give developers more control over a variable's life cycle.

Block Scopes are created in the following places:

- 1- Inside a function
- 2- Inside a block (indicated by the { and } characters)

Let Declaration

The **let** declaration syntax is the same as the syntax for **var**. You can basically replace var with let to declare a variable but limit the variable's scope to only the current code block

```
let studentId = 2;  
console.log(studentId); // output 2
```

let declarations are **not hoisted** to the top of the enclosing block, it's best to place let declarations first in the block so they're available to the entire block.

```
console.log(studentId); //error: studentId is not defined  
let studentId = 2;
```

By using let you **can not** define parameter twice

```
let count = 2;
```

```
var count=2; //error : Identifier 'count' has already been declared  
//or
```

```
var count=2;
```

```
let count = 2; //error : Identifier 'count' has already been declared
```

Because let will not redefine an identifier that already exists in the same scope, the let declaration will throw an error.

```
var count = 30;
```

```
if (true) {
```

```
    let count = 40; // doesn't throw an error
```

```
}
```

```
function countWords(myName)
```

```
{
```

```
    let count = 0;
```

```
    //more code
```

```
}
```

Const Declaration

constants, meaning their values **cannot be** changed once set. For this reason, every const variable must be initialized on declaration

```
// valid constant
```

```
const maxItems = 30;
```

```
maxItems=40; //error: Assignment to constant variable.
```

```
const name; // syntax error: Missing initializer in const declaration
```

Const variables are not hoisted

```
console.log(maxItems); //error : maxItems is not defined
```

```
const maxItems = 30;
```

- ✓ Constants, like let declarations, are block-level declarations.

```
const maxItems = 30;
if (true) {
    const maxItems = 5;
    // more code
}
```

- ✓ In another similarity to let, a const declaration throws an error when made with an identifier for an already defined variable in the same scope.

```
var message = "Hello!";
let age = 25;
// each of these throws an error
const message = "Goodbye!";
const age = 30;
```

- ✓ Even if we start defining variable with const

```
const age = 30;
let age=30; // error : Identifier 'age' has already been declared
```


Block Bindings in Loops

Perhaps one area where developers most want block-level scoping of variables is within `for` loops, where the throwaway counter variable is meant to be used only inside the loop.

```
items=[1,2,3,4,5,6,7,8,9,10]
```

```
for (var i = 0; i < 10; i++) {  
    console.log (items[i]);  
}
```

```
console.log(i); // 10 -> i is still accessible here
```

The variable `i` is still accessible after the loop is completed because the `var` declaration is hoisted.

```
for (let i = 0; i < 10; i++) {  
    console.log (items[i]);  
}
```

```
console.log(i); // i is not accessible here - throws an error
```

In this example, the variable `i` exists only within the `for` loop. When the loop is complete, the variable is no longer accessible elsewhere.

Functions in loops

```
<ul>
  <li>water</li>
  <li>milk</li>
</ul>
<script>
  var liElm = document.getElementsByTagName("li");
  for (var i = 0; i < liElm.length; i++) {
    liElm[i].onclick = function () {
      alert(i); //alert always show 2
    };
  }
</script>
```

The characteristics of var have long made creating functions inside loops problematic, because the loop variables are accessible from outside the scope of the loop.

The `let` declaration creates a **new** variable `i` each time through the loop, so each function created inside the loop gets its own copy of `i`. Each copy of `i` has the value it was assigned at the beginning of the loop iteration in which it was created.

```
for (let i = 0; i < liElm.length; i++) {  
    liElm[i].onclick = function () {  
        alert(i);  
    };  
}
```

The same is true for `for-in` and `for-of` loops.

Global Blocks Bindings

Another way in which `let` and `const` are different from `var` is in their **global scope** behaviour. When `var` is used in the global scope, it creates a new global variable, which is a property on the global object (window in browsers). That means you can accidentally overwrite an existing global using `var`

```
var name="Eman";  
console.log(name===window.name); // true
```

If you instead use `let` or `const` in the global scope, a new binding is created in the global scope but no property is added to the global object.

```
let name="Eman";  
console.log(name===window.name); // false
```



Template Literals & multiline String

Template Literals

Template literals are ECMAScript 6's answer to the following features that JavaScript lacked in ECMAScript 5 and in earlier versions:

- ✓ Multiline strings A formal concept of multiline strings
- ✓ Basic string formatting The ability to substitute parts of the string for values contained in variables.

At their simplest, template literals act like regular strings delimited by **backticks (`)** instead of double or single quotes.

```
let message = `Hello world!`;
console.log(message); // "Hello world!"
console.log(typeof message); // "string"
console.log(message.length); // 12
```

There's no need to escape either double or single quotes inside template literals.

Multiline Strings

JavaScript developers have wanted a way to create multiline strings since the first version of the language. But when you're using double or single quotes, strings must be completely contained on a single line.

Pre-ECMAScript 6 Workarounds Thanks to a long-standing syntax bug, JavaScript does have a workaround for creating multiline strings. You can create multiline strings by using a **backslash (\)** before a newline.

```
var message = "Multiline \nstring";  
console.log(message); // "Multiline string"
```

ECMAScript 6's template literals make multiline strings easy because there's no special syntax. Just include a newline where you want, and it appears in the result


```
let message = `Multiline
string`;
console.log(message); // "Multiline
                      // string"
console.log(message.length); // 16
```

All whitespace inside the **backticks** is part of the string, so be careful with indentation. For example:

```
let message = `Multiline
                string`;
console.log(message); // "Multiline
                      // string"
console.log(message.length); // 31
```

Making Substitutions

Substitutions are delimited by an opening `${` and a closing `}` that can have any JavaScript expression inside. The simplest substitutions let you embed local variables directly into a resulting string.

```
let name = "Nicholas",  
message = `Hello, ${name}.`;   
console.log(message); // "Hello, Nicholas."
```

Because all substitutions are JavaScript expressions, you can substitute more than just simple variable names. You can easily embed calculations, function calls, and more.

```
let count = 10,  
price = 0.25,  
message = `${count} items cost $$${(count * price).toFixed(2)}.`;  
console.log(message); // "10 items cost $2.50."
```



Functions Blocks

Functions Blocks

- ✓ **Functions with Default Parameter Values**
- ✓ **Spread and rest Operators**
- ✓ **Arrow Functions**



Function Blocks: Default Parameters

Functions with Default Parameter Values

Functions in JavaScript are **unique** in that they allow any number of parameters to be passed regardless of the number of parameters declared in the function definition. This allows you to define functions that can handle different numbers of parameters, often by just filling in **default values** when parameters aren't provided.

```
function getProduct(price, type="HardWare")
{
    //do something
}

//uses default type parameter
getProduct(1000);
//overwrite Default type value
getProduct(1000, "software");
```

```
function makeRequest(url, timeout = 2000, callback = function() {}) {  
    // the rest of the function };  
  
    // uses default timeout and callback  
    makeRequest("/foo");  
    // uses default callback  
    makeRequest("/foo", 500);  
    // doesn't use defaults  
    makeRequest("/foo", 500, function() {  
        //doSomething; });  
    function callBackRequest(){/*body*/}  
    makeRquest("/foo",500,callBackRequest);
```


How Default Parameter Values Affect the arguments Object

```
function makeRequest(url, timeout = 2000, callback = function() {}) {  
    console.log(arguments.length);  
};
```

```
// uses default timeout and callback
```

```
makeRequest("/foo"); //→ output 1
```

```
// uses default callback
```

```
makeRequest("/foo", 500); //→ output 2
```

```
// doesn't use defaults
```

```
makeRequest("/foo", 500, function() {
```

```
//doSomething; }); //→ output 3
```

Default Parameter Expressions

The most interesting feature of default parameter values is that the default value need not be a primitive value.

```
let baseDiscount=0.5;
function getProduct(price, type="HardWare",Discount=baseDiscount)
{ //do something ; }
getProduct(1000); //Discount→ 0.5
function getBaseDiscount(){ return 0.2 }
function getProduct(price, type="HardWare",Discount=getBaseDiscount())
{ //do something }
getProduct(1000); //Discount→ 0.2
```

Keep in mind that **getBaseDiscount()** is called only when **getProduct()** is called without a second parameter

You can use a previous parameter as the default for a later parameter.

```
function getProduct(price, type="HardWare",Discount=price*0.2)
{ //do something }
getProduct(1000); // Discount = 200
```

you can pass **price** into a function to get the value for **Discount**

```
function getBaseDiscount(value){ return value*0.2 }
function getProduct(price, type="HardWare",Discount=getBaseDiscount(price))
{ //do something }
getProduct(1000); //Discount = 200
```

The ability to reference parameters from default parameter assignments works only for **previous** arguments, so earlier arguments don't have access to later arguments.

```
function getProduct(price=Discount, type="HardWare",Discount=0.3)
{ //do something }
getProduct(); // ERROR
```



Function Blocks: Rest and Spread

Rest Parameters

A **rest** parameter is indicated by **three dots (...)** preceding a named parameter. That named parameter becomes an **Array** containing the rest of the parameters passed to the function, which is where the name rest parameters originates.

```
function addOrder(orderId, ...products)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
addOrder(2, "item1", "item2");
```

Rest Parameter Restrictions

The first restriction is that there **can be only one rest parameter**, and the rest parameter **must be last parameter**.

```
function addOrder(orderId, ...products, category)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
```

Or

```
function addOrder(orderId, ...products, ...category)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
```

Syntax Error → Rest parameter must be last formal parameter.

The Spread Operator

The **spread** operator allows you to specify an **array** that should be **split** and passed in as separate arguments to a function.

Consider the built-in `Math.max()` method, which accepts any number of arguments and returns the one with the highest value.

```
var numbers = [3, 1, 7, 4, 9];  
console.log(Math.max(3,1,7,4,9)); //9  
//or  
console.log(Math.max.apply(null,numbers)); //9
```

you can pass the array to `Math.max()` directly and prefix it with the same **... pattern** you use with rest parameters. The JavaScript engine then splits the array into individual arguments and passes them in.

```
console.log(Math.max(...numbers)); //9
```

We can use spread inside another array as follow

```
var AllNumbers = [33, 55, 11, ...numbers, 90]; // [33,55,11,3,1,7,4,9,90]
```




Function Blocks: Arrow Functions

Arrow Functions

One of the **most** interesting new parts of **ECMAScript 6** is the arrow function.

Arrow functions are, as the name suggests, functions defined with a new syntax that uses an **arrow (=>)**. But arrow functions behave differently than traditional JavaScript functions in a number of important ways:

- **Cannot be called with new** Arrow functions do not have a `[[Construct]]` method and therefore cannot be used as constructors. Arrow functions throw an error when used with `new`.
- **Can't change this** The value of `this` inside the function can't be changed. It remains the same throughout the entire life cycle of the function.
- **No arguments object** Because arrow functions have no arguments binding, you must rely on named and rest parameters to access function arguments.

Arrow Function Syntax

All variations begin with function arguments, followed by the arrow, followed by the body of the function. The arguments and the body can take different forms depending on usage.

Function have no input or return

```
let getPrice = () => console.log("testing");  
getPrice(); //-->testing
```

This is equivalent to

```
let getPrice =function ()  
{  
    console.log("testing");  
}
```

✓ Function take one argument and return one value

```
let getBaseDiscount = (price) => price*0.2;  
console.log(getBaseDiscount (1000)); //-->200
```

This is equivalent to

```
let getBaseDiscount=function (price) {  
    return price * 0.2  
}
```

✓ Function with more than one input and return one value

```
let getPrice = (product, price) =>[product, price];
```

✓ Function with more than one output statement

```
let getPrice = (product, price) => {  
    let result;  
    if(price>50)  
        result = product + " : " + (price * 2)  
    else  
        result = product + " : " + (price)  
    return result;  
}
```

Curly braces denote the function's body, which works just fine in the cases you've seen so far. But an arrow function that wants to return an object literal outside a function body must wrap the literal in parentheses.

```
let getTempItem = id => ({ id: id, name: "Temp" });
```

```
// effectively equivalent to:
```

```
let getTempItem = function(id) {  
    return {  
        id: id,  
        name: "Temp"  
    };  
};
```

No this Binding

Because the value of this can change inside a single function depending on the context in which the function is called, it's possible to mistakenly affect one object when you meant to affect another.

```
function Person()  
{  
    this.count=20;  
    setTimeout(function(){  
        this.count++; //this here does not refer to Person Object (Window Object)!!!  
    },2000);  
}
```

This is equivalent to

```
function timeoutFun()  
{ this.count++; //here this refers to window object }  
function Person()  
{    this.count=20;    setTimeout("timeoutFun()",2000);}
```

Arrow functions have no this binding, which means the value of this inside an arrow function can only be determined by looking up the scope chain.

```
var video={
  title:"ES6",
  tags:["js","jquery","es5"],
  showTags:function(){
    this.tags.forEach(function(tag){
      console.log(this.title,tag)//this refers to window object
    })//end of forEach;
  }//end of showTags
}

video.showTags();
```

Now using arrow function

```
var video={
  title:"ES6",
  tags:["js","jquery","es5"],
  showTags:function(){
    this.tags.forEach((tag)=>{
      console.log(this.title,tag) //this now referes to video object
    })//end of forEach;
  } //end of showtags
}

video.showTags();
```


Warnings

```
var Book = {Title: "ES",  
            Borrow: function (name) {console.log(name+ "borrowing"+ this.Title);}  
            };  
console.log(Book.Borrow("Eman"));    // Eman borrowing ES
```

But

```
var Book = {Title: "ES",  
            Borrow:(name) =>{ console.log(name + " borrowing " + this.Title); }  
            };  
Console.log(Book.Borrow("Eman"));    // Eman borrowing undefined
```

And

If you try to use the new operator with an arrow function, you'll get an error

```
var MyType = () => {},  
object = new MyType(); // error - MyType is not a constructor
```

Arrow Functions and Arrays

The concise syntax for arrow functions makes them ideal for use with array processing, too.

```
var result = values.sort(function(a, b) {  
    return a - b;  
});
```

```
var result = values.sort((a, b) => a - b);
```

The array methods that accept callback functions, such as `sort()`, can all benefit from simpler arrow function syntax, which changes seemingly complex processes into simpler code.

No arguments Binding

Arrow function does not contains arguments object as normal Functions

```
let myFunction=(x)=>console.log(arguments.length)  
myFunction(3)  //error -> arguments is not defined
```



Expanded Object Functionality

Object Literal Syntax Extensions

The object literal is one of the most popular patterns in JavaScript. **JSON** is built on its syntax, and it's in nearly every JavaScript file on the Internet. The object literal's popularity is due to its succinct syntax for creating objects that would otherwise take several lines of code to create. Fortunately for developers, ECMAScript 6 makes object literals more powerful and even more succinct by extending the syntax in several ways.

Property Initializer Shorthand

```
let Title = "EcmaScript";
```

```
let version = "2015";
```

In ECMAScript 5 :

```
var Book = {Title:Title, version:version}
```

In ECMAScript 6:

```
var Book = {Title, version};
```

```
console.log(Book); //output --> {Title: "EcmaScript", version: "2015"}
```

In ECMAScript 6, you can eliminate the duplication that exists around property names and local variables by using the property initializer shorthand syntax. When an object property name is the same as the local variable name, you can simply include the name without a colon and value.

When a property in an object literal only has a name, the JavaScript engine looks in the surrounding scope for a variable of the same name. If it finds one, that variable's value is assigned to the same name on the object literal

In ECMAScript 5 :

```
function createPerson(name, age) {  
    return {  
        name: name,  
        age: age  
    };  
}
```

In ECMAScript 6 :

```
function createPerson(name, age) {  
    return { name, age};  
}  
  
createPerson("eman", 20)
```

Concise Methods

ECMAScript 6 also improves the syntax for assigning methods to object literals.

In **ECMAScript 5** and earlier, you must specify a name and then the full function definition to add a method to an object.

```
var person = {  
  name: "Nicholas",  
  sayName: function () {console.log(this.name);}  
};
```

In **ECMAScript 6**, the syntax is made more concise by eliminating the colon and the function keyword.

```
var person = {  
  name: "Nicholas",  
  sayName() {console.log(this.name);}  
};
```


Computed Property Names

ECMAScript 5 and earlier could compute property names on object instances when those properties were set with square brackets instead of dot notation. The square brackets allow you to specify property names using variables and string literals that might contain characters that would cause a syntax error if they were used in an identifier.

```
var person = {},
    lastName = "last name";
person["first name"] = "Eman";
person[lastName] = "Fathi";
console.log(person["first name"]); // "Eman"
console.log(person[lastName]); // "Fathi"
```

Additionally, you can use string literals directly as property names in object literals

```
var person = {"first name": "Eman"};
console.log(person["first name"]); // "Eman"
```

In **ECMAScript 6**, computed property names are part of the object literal syntax, and they use the same square bracket notation that has been used to reference computed property names in object instances.

```
var titleParameter = "Title";  
var titleValue = "ES6";
```

```
var Book = {  
  [titleParameter]: titleValue,  
  ["Book"+titleParameter]: titleValue,  
  [titleValue+" printing"]: function () {  
    return this[titleParameter] + " : " + this["Book" + titleParameter];  
  }  
};  
console.log(Book)//{Title: "ES6", BookTitle: "ES6", ES6 printing: f}
```

ECMAScript 2015 (ES6)

Eman Fathi

The image features a large, bright yellow circle on the right side containing the letters 'JS' in a bold, black, sans-serif font. The background is a dark blue-grey color with several overlapping circles in shades of red, orange, and yellow. Some of these circles contain faint icons: a code editor window with a '</>' symbol and a play button. The overall design is modern and tech-oriented.

JS



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
```

Basic Class Expression

ECMAScript 10

```
class Person
{
    name="eman";    //now we can define public here
    age=20;
    #ID=1;          //private member
    constructor(name,age,ID)
    {
        this.name=name;
        this.age=age;
        this.#ID=ID;
    }
};
```

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
{  static count=0;          //new Feature
    constructor(name,age)
    {  Person.count++;
        this.name=name;
        this.age=age;
    }
    static personInfo()      //basic ES6 Fearture
    {
        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
```



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

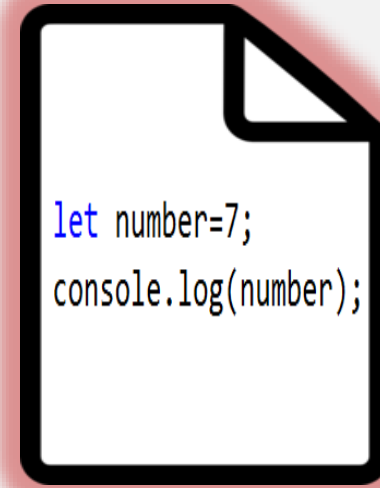
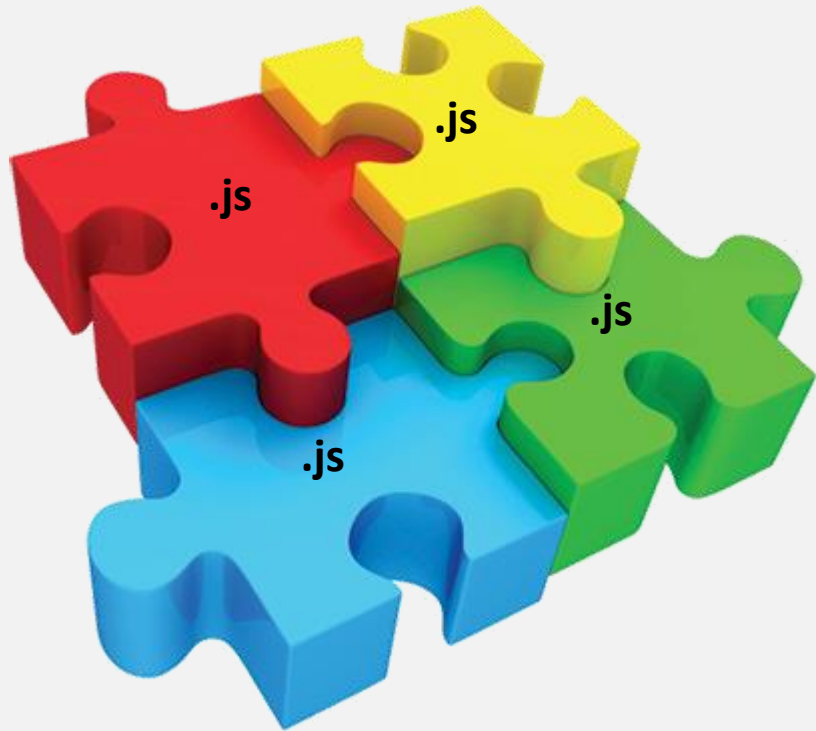
Encapsulating Code with Modules

JavaScript's

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

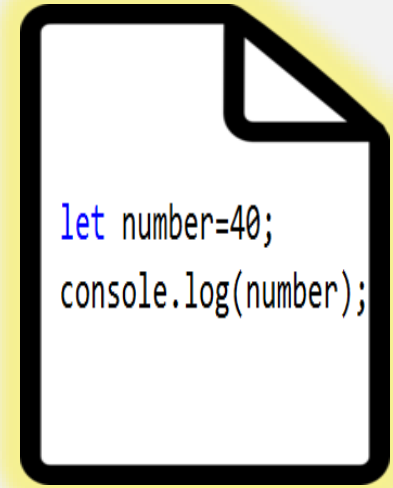
Other languages use concepts such as packages to define code scope, but before ECMAScript 6, everything defined in every JavaScript file of an application shared one global scope. As web applications became more complex and started using even more JavaScript code, that approach caused problems, such 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



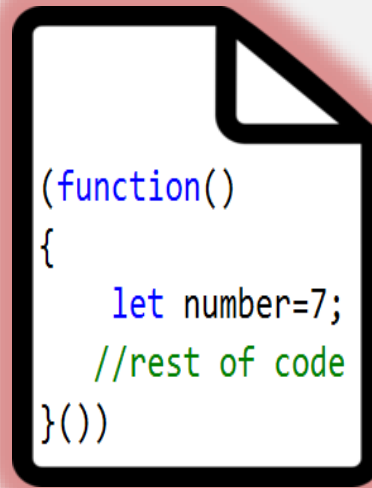
```
let number=40;  
console.log(number);
```

Yellow.js

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

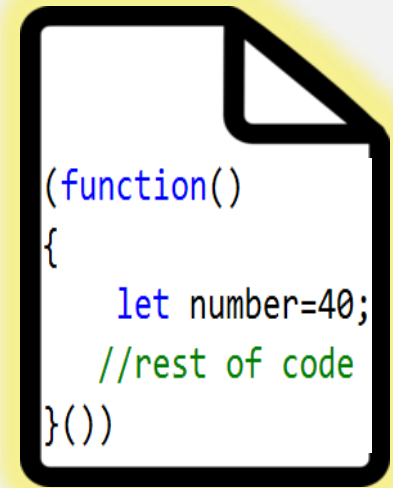
IIFE (Immediately Invoked Function Expression)

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

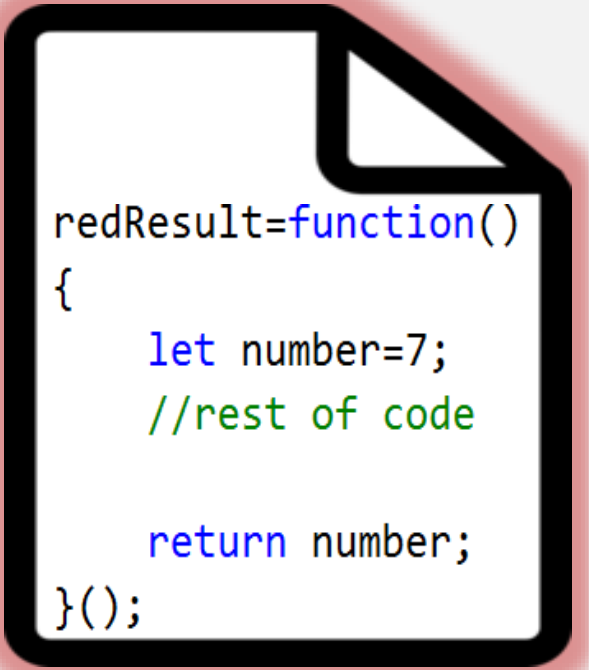
A document icon with a yellow glow, representing a JavaScript file named Yellow.js. It contains the following code:

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

Yellow.js

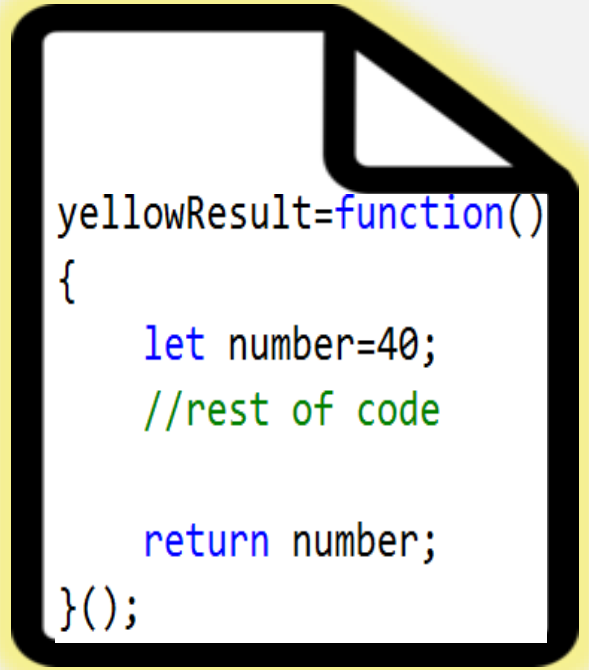
```
console.log(number); //throw Error
```

IIFE (Immediately Invoked Function Expression)

A document icon with a red border and a folded top-right corner, representing a file named Red.js.

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

Red.js

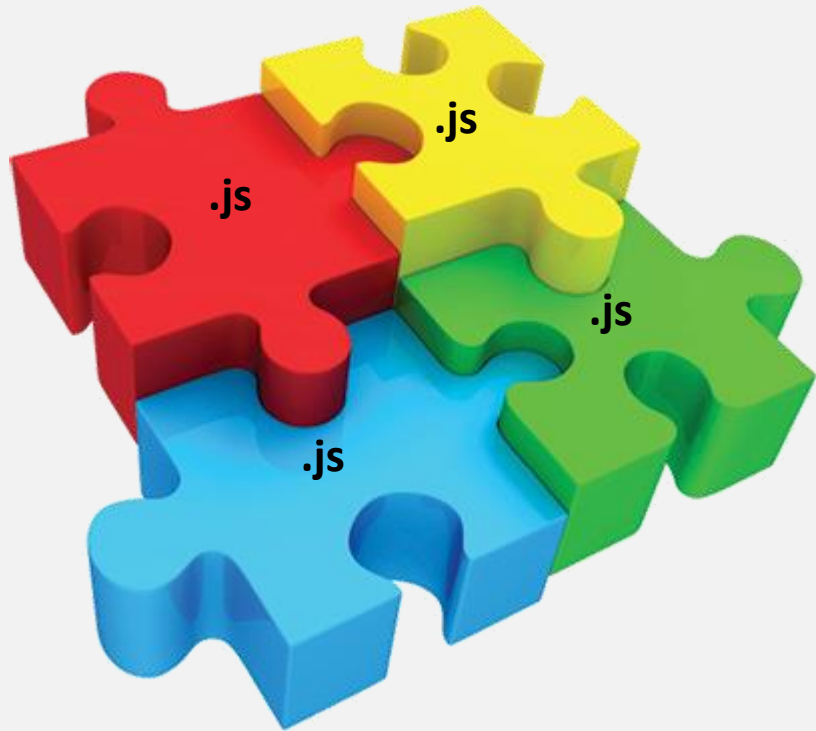
A document icon with a yellow border and a folded top-right corner, representing a file named Yellow.js.

```
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**

MODULE

- 👉 Reusable piece of code that **encapsulates** implementation details;
- 👉 Usually a **standalone file**, but it doesn't have to be.

WHY
MODULES?

- 👉 **Compose software:** Modules are small building blocks that we put together to build complex applications;
- 👉 **Isolate components:** Modules can be developed in isolation without thinking about the entire codebase;
- 👉 **Abstract code:** Implement low-level code in modules and import these abstractions into other modules;
- 👉 **Organized code:** Modules naturally lead to a more organized codebase;
- 👉 **Reuse code:** Modules allow us to easily reuse the same code, even across multiple projects.

IMPORT
(DEPENDENCY)



MODULE

```
import { rand } from './math.js';  
const diceP1 = rand(1, 6, 2);  
const diceP2 = rand(1, 6, 2);  
const scores = { diceP1, diceP2 };  
export { scores };
```

Module code



EXPORT
(PUBLIC API)

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";
```



Promise

Asynchronous Programming Background

JavaScript engines are built on the concept of a **single-threaded event loop**. Single-threaded means that only one piece of code is executed at a time. Contrast this with languages like Java or C++, where threads can allow multiple different pieces of code to execute at the same time.

JavaScript engines can execute only one piece of code at a time, so they need to keep track of code that is meant to run. That code is kept in a **job queue**. Whenever a piece of code is ready to be executed, it is added to the job queue. When the JavaScript engine is finished executing code, the event loop executes the next job in the queue. The event loop is a process inside the JavaScript engine that monitors code execution and manages the job queue. Keep in mind that as a queue, job execution runs from the first job in the queue to the last.

PROMISE

👉 **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.

↓ Less formal

👉 **Promise:** A container for an asynchronously delivered value.

↓ Less formal

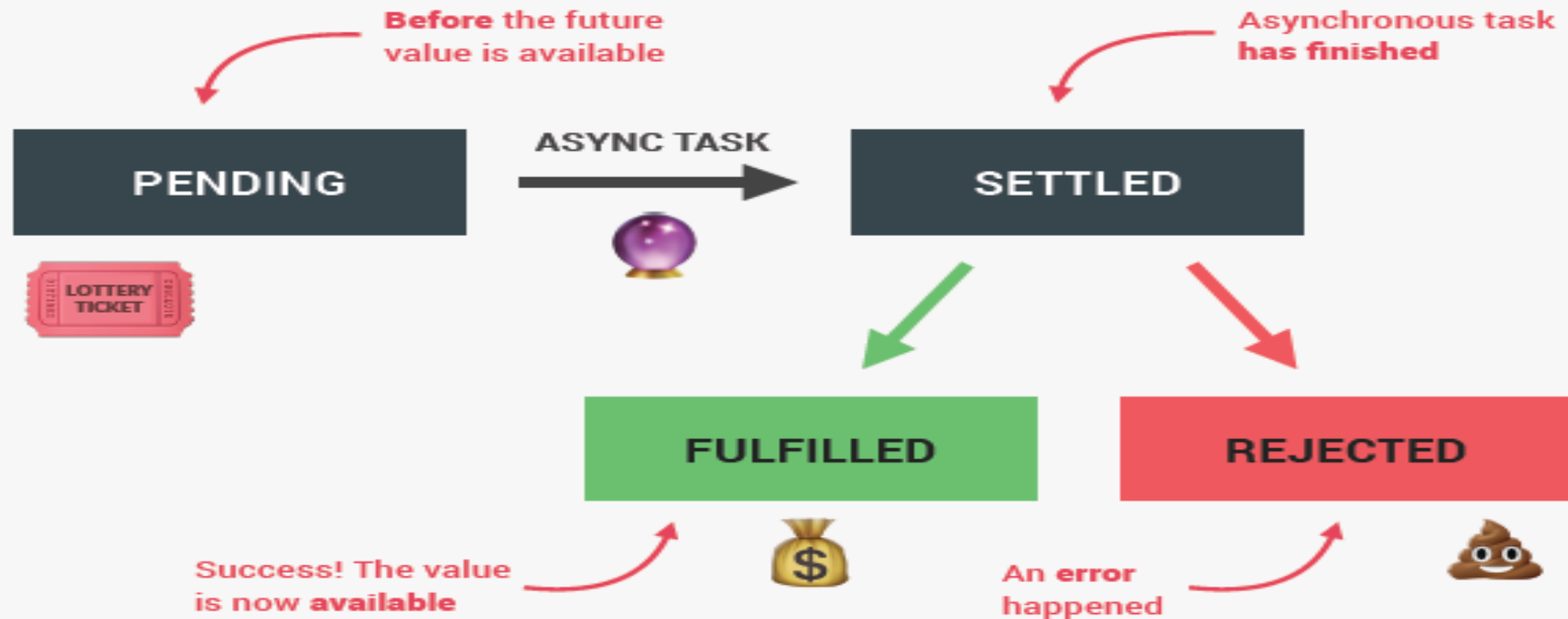
👉 **Promise:** A container for a future value.

Example: Response
from AJAX call

👉 We no longer need to rely on events and callbacks passed into asynchronous functions to handle asynchronous results;

👉 Instead of nesting callbacks, we can **chain promises** for a sequence of asynchronous operations: **escaping callback hell** 🎉

The Promise LifeCycle



👉 We are able **handle** these different states in our code!

Asynchronous Programming Background

Callback functions

```
let buttonObj = document.getElementById("my-btn");  
buttonObj.onclick = function(event) {  
    console.log("Clicked");  
};
```

In this code, `console.log("Clicked")` will not be executed until button is clicked. When button is clicked, the function assigned to `onclick` is added to the back of the job queue and will be executed when all other jobs ahead of it are complete.

Asynchronous Programming Background

Callback patterns

```
makeSandwich(function(){
    makeTea(function(){
        eatSandwich(function(){
            drinkTea(function(){
                cleanUp(function(){
                    //All work Done
                }); //cleanUp
            }); //drinkTea
        }); //eatSandwich
    }); //makeTea
}); //makeSandwich
```

***Callback Hell
In Javascript***

Asynchronous Programming Background

Callback patterns

Nodejs Example:

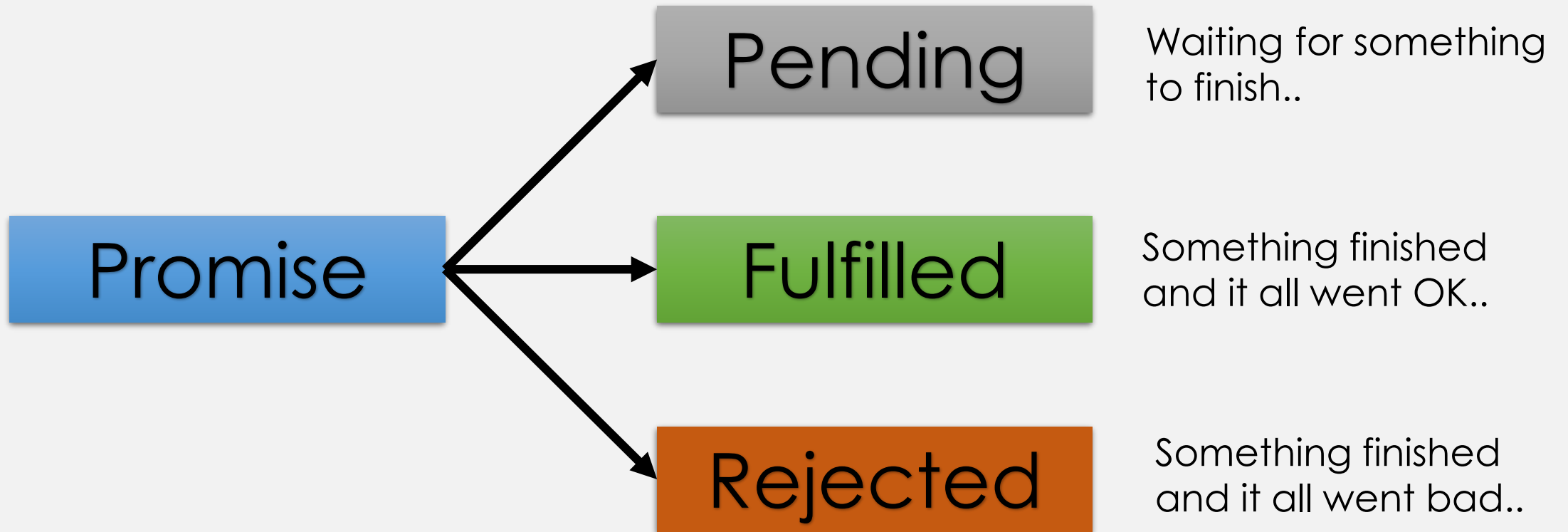
```
readFile("example.txt", function(err, contents) {  
    if (err) {throw err;}  
    writeFile("example.txt", function(err) {  
        if (err) {throw err;}  
        console.log("File was written!");  
    }); //write file  
}); //readFile
```

Promise Basics

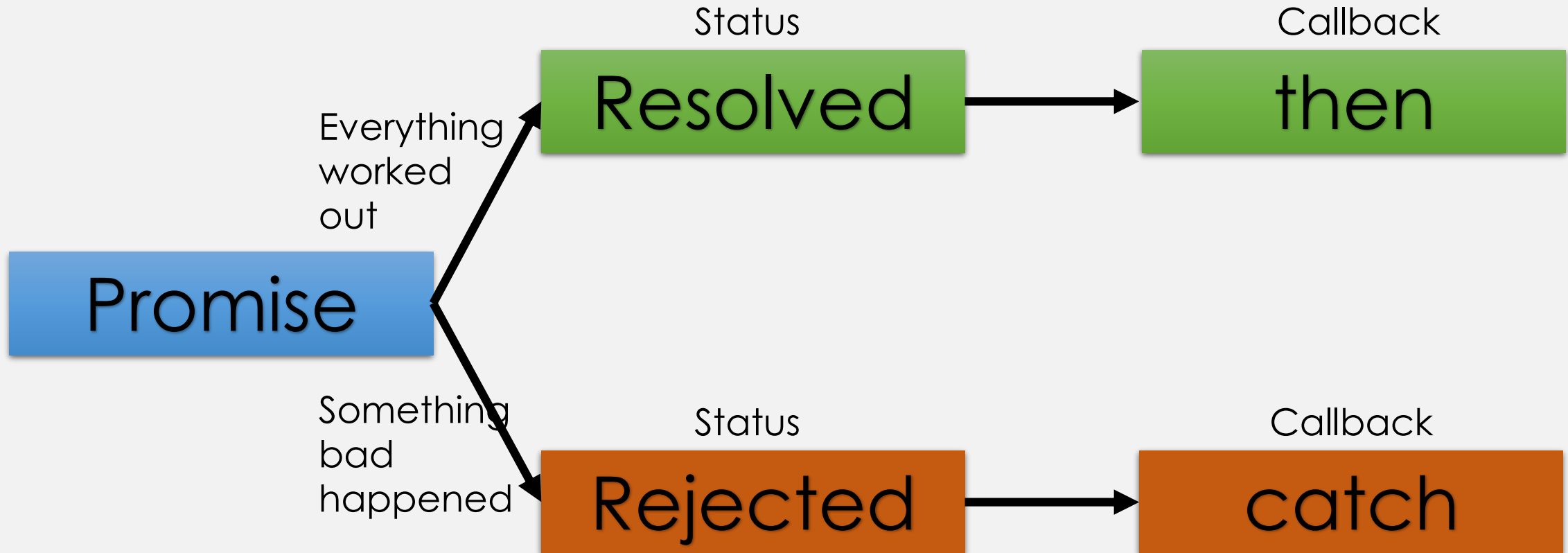
A promise is a placeholder for the result of an asynchronous operation.

Each promise goes through a short life cycle starting in the **pending state**, which indicates that the asynchronous operation hasn't completed yet. A pending promise is considered **unsettled**.

Promise States



Promise States



Promise Basics

```
let promise=new Promise((resolve,reject)=>{  
    resolve();  
});
```

```
promise.then(()=>{  
    console.log("resolved");  
});
```

```
promise.catch(()=>{  
    console.log("ERROR");  
});
```

```
//output -- > resolved
```

Promise Basics

```
let promise=new Promise((resolve,reject)=>{  
    reject();  
});
```

```
promise.then(()=>{  
    console.log("resolved");  
});
```

```
promise.catch(()=>{  
    console.log("ERROR");  
});
```

```
//output -- > ERROR
```

Promise Basics

```
let promise=new Promise((resolve,reject)=>{
  let number=2;
  if(number==2) resolve();
  else reject();
});

promise.then(()=>{
  console.log("resolved");
});
promise.catch(()=>{
  console.log("ERROR");
});
//output -- > ERROR
```

Promise Basics

```
let promise=new Promise((resolve,reject)=>{  
    let number=2;  
    if(number==2) resolve("Promise is OK");  
    else reject("Promise is BAD");  
});
```

```
promise.then((value)=>{  
    console.log(value);  
}).catch((value)=>{  
    console.log(value);  
});
```

```
//output -- > Promise is OK
```

Responding to Multiple Promises

Promise.all()

The `Promise.all()` method accepts a single argument, which is an iterable (such as an array) of promises to monitor, and returns a promise that is resolved only when every promise in the iterable is resolved.

```
let promise_1 = new Promise(function(resolve, reject) {resolve(11);});
let promise_2 = new Promise(function(resolve, reject) {resolve(22);});
let promise_3 = new Promise(function(resolve, reject) {resolve(33);});
let allPromises = Promise.all([promise_1, promise_2, promise_3]);
allPromises.then(function(value) {
    console.log(Array.isArray(value)); // true
    console.log(value[0]); // 11
    console.log(value[1]); // 22
    console.log(value[2]); // 33
});
```

Responding to Multiple Promises

Promise.race()

The `Promise.race()` method provides a slightly different take on monitoring multiple promises. This method also accepts an iterable of promises to monitor and returns a promise, but the returned promise is settled as soon as the first promise is settled. Instead of waiting for all promises to be fulfilled, like the `Promise.all()` method, the `Promise.race()` method returns an appropriate promise as soon as any promise in the array is fulfilled.

```
let promise_1 = new Promise(function(resolve, reject) {resolve(11);});
let promise_2 = new Promise(function(resolve, reject) {resolve(22);});
let promise_3 = new Promise(function(resolve, reject) {resolve(33);});
let allPromises = Promise.race([promise_1, promise_2, promise_3]);
allPromises.then(function(value) {
    console.log(value); // 11
});
```

Responding to Multiple Promises

Promise.race()

The `Promise.race()` method provides a slightly different take on monitoring multiple promises. This method also accepts an iterable of promises to monitor and returns a promise, but the returned promise is settled as soon as the first promise is settled. Instead of waiting for all promises to be fulfilled, like the `Promise.all()` method, the `Promise.race()` method returns an appropriate promise as soon as any promise in the array is fulfilled.

```
let promise_1 = new Promise(function(resolve, reject) {resolve(11);});
let promise_2 = new Promise(function(resolve, reject) {resolve(22);});
let promise_3 = new Promise(function(resolve, reject) {resolve(33);});
let allPromises = Promise.race([promise_1, promise_2, promise_3]);
allPromises.then(function(value) {
    console.log(value); // 11
});
```




async / await