طرق تعريف المتغيرات في JavaScript

في JavaScript يوجد 3 كلمات أساسية لتعريف المتغير:
var
let
const
خلينا نشرحهم.

1. var كانت الطريقة القديمة لتعريف المتغيرات.

```
var age = 20;
```

لكن فيها مشاكل، لذلك نادرًا ما تُستخدم الآن.

أهم مشكلة: Scope (نطاق المتغير) غير واضح أحيانًا.

مثال:

```
var x = 10;

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

2. let هي الطريقة الحديثة لتعريف متغير يمكن تغيير قيمته.

```
let age = 20;
age = 25;
```

هنا غيرنا القيمة.

أهم ميزة: Block Scope يعني المتغير يعمل فقط داخل { }

مثال:

```
let x = 10;
if (true) {
  let x = 20;
  console.log(x); // 20
}
console.log(x); // 10
```

3. const (قيمة ثابتة) يعني لا يمكن تغيير القيمة بعد تعريفها.

لكن انتبه لنقطة مهمة:

إذا كان المتغير Object أو Array يمكن تعديل المحتوى لكن ليس إعادة تعريفه.

مثال:

```
const user = {
  name: "Hossam"
};

user.name = "Ali"; // مسموح

console.log(user); // {name: 'Ali'}
console.log(user.name); // Ali
```

لكن هذا غير مسموح:

```
user = {};
```

الفرق بين var و let و const

Feature	var	let	const
Scope	Function Scope	Block Scope	Block Scope
يمكن تغيير القيمة	نعم	نعم	لا
إعادة تعريف المتغير	نعم	لا	لا
يستخدم في المشاريع الحديثة	نادرًا	نعم	نعم

أنواع ال Data Types

```
let name = "Hossam";  // String
let age = 22;               // Number
let isStudent = true;   // Boolean
let colors = ["red", "blue", "green"];   // Array
let user = {
  name: "Hossam",
  age: 22
};   // Object
```

متى نستخدم let ومتى const؟

هذه قاعدة يستخدمها أغلب المطورين: استخدم const أولًا دائمًا وإذا احتجت تغيير القيمة استخدم let

مثال:

```
const apiUrl = "https://api.site.com";
 let counter = 0;
```

مفهوم مهم جدًا في JavaScript

اسمه: Hoisting وهو أن JavaScript يرفع تعريف المتغير قبل تنفيذ الكود.

مثال:

```
console.log(y);
var y = 5;
```

الجافا سكريبت تقرأه هكذا:

```
var x;
console.log(x);
x = 5;
```

مثال ثاني:

```
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element

var x; // Declare x
```

الجافا سكريبت تقرأه هكذا:

```
var x; // Declare x
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element
```

ما هو Scope أصلًا؟

Scope يعني: المكان الذي يمكن الوصول فيه إلى المتغير (where a variable is accessible) بمعنى أين يمكن استخدام المتغير داخل الكود يوجد نوعان مهمان هنا:

1️⃣ **Function Scope** يعني أن المتغير يمكن استخدامه فقط داخل function التي تم تعريفه فيها **وهذا يحدث مع var**

```
function test() {
  var x = 10;
  console.log(x); // يعمل
}

test();
console.log(x); // Error
```

لماذا حدث الخطأ؟
لأن المتغير x موجود فقط داخل function scope.
يعني: x lives only inside the function

مثال يوضح المشكلة في var

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

هون عادي رح يطبع 20

لماذا؟
لأن var لا يحترم block
بل يحترم فقط function.

وهذه نقطة مهمة جدًا.

2️⃣ **Block Scope** يعني انو ال variable بشتغل داخل {}

الخلاصة انو ال var هو فقط function scope بينما ال let هو فنكشن scope and block scope يعني بحترمهم الاثنين

Feature	Function Scope	Block Scope
يعمل داخل	function فقط	أي block { }
يستخدم مع	var	let / const
يظهر خارج if	نعم	لا
يظهر خارج for	نعم	لا

Arithmetic Operators = العمليات الحسابية التي نستخدمها مع الأرقام (numbers) مثل الجمع والطرح.

أمثله:

```
let a = 10;
let b = 5;
console.log(a + b); // 15
```

المثال يلي تحت رح يعمل **Concatenation فقط في حالة +**

```
/*
let a = "10";
let b = 5;

console.log(a + b); // 105
*/
```

اما في حالة الضرب * **والقسمة / والطرح - والاس ** وباقي القسمة % رح يشتغل زي كانو number**
وهذا المصطلح اسمو Type Coercion

name	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Exponentiation	**
Increment	++
Decrement	--

Example:

```
let number1 = 10;
let number2 = 5;

let sum = number1 + number2;
let sub = number1 - number2;
let multi = number1 * number2;
let div = number1 / number2;
let mod = number1 % number2;

let message = `First Number is: ${sum}\nSecond Number is: ${number2}\nSum is:
${sum}\nSub is: ${sub}\nMulti is: ${multi}\nDiv is: ${div}\nMod is: ${mod}`;

let res = document.getElementById("result").innerHTML = message;
```

```
<h1>Hello World!</h1>
 <h3 id="result"></h3>
```

في JavaScript يوجد 3 طرق لكتابة النص:

Double quotes **" "**
Single quotes **' '**
string interpolation **` `**

```
let name = "Hossam";
let age = 23;

console.log(`My name is ${name} and I am ${age} years old`);
```

String Method	Explanation (English)	Example
`length`	Returns the number of characters in a string.	`js let text = "JavaScript"; console.log(text.length); // 10`
`toUpperCase()`	Converts all characters in a string to uppercase letters.	`js let text = "hello"; console.log(text.toUpperCase()); // HELLO`
`toLowerCase()`	Converts all characters in a string to lowercase letters.	`js let text = "HELLO"; console.log(text.toLowerCase()); // hello`
`includes()`	Checks if a string contains a specific substring. Returns true or false.	`js let text = "I love JavaScript"; console.log(text.includes("love")); // true`
`startsWith()`	Checks if a string starts with a specific substring.	`js let text = "JavaScript"; console.log(text.startsWith("Java")); // true`
`endsWith()`	Checks if a string ends with a specific substring.	`js let text = "JavaScript"; console.log(text.endsWith("Script")); // true`
`indexOf()`	Returns the first position of a substring in a string. Returns -1 if not found.	`js let text = "Hello JavaScript"; console.log(text.indexOf("JavaScript"));`
`slice()`	Extracts a part of a string and returns it as a new string.	`js let text = "JavaScript"; console.log(text.slice(0, 4)); // Java`

`substring()`	Extracts characters between two indexes in a string.	`js let text = "JavaScript"; console.log(text.substring(0, 4)); // Java`
`replace()`	Replaces the first occurrence of a substring with another value.	`js let text = "I love Java"; console.log(text.replace("Java", "JavaScript"));`
`replaceAll()`	Replaces all occurrences of a substring.	`js let text = "JS JS JS"; console.log(text.replaceAll("JS", "JavaScript"));`
`trim()`	Removes whitespace from both ends of a string.	`js let text = " hello "; console.log(text.trim()); // hello`
`split()`	Splits a string into an array based on a separator.	`js let text = "red,blue,green"; console.log(text.split(","));`
`charAt()`	Returns the character at a specific index.	`js let text = "JavaScript"; console.log(text.charAt(0)); // J`
`concat()`	Joins two or more strings together.	`js let a = "Hello"; let b = "World"; console.log(a.concat(" ", b));`

الفرق بين slice و substring (سؤال مقابلات)

الفرق المهم: slice يقبل أرقام سالبة اما substring لا يقبل

** الكود يلي تحت اي رقم بعتبر true باستثناء ال 0 ف هو false

```
var x=4  or -4;
console.log(Boolean(x));  // true
```

** الكود يلي تحت الفنكشن **isNaN** كملة NaN اختصار ل Not a Number

```
var x = "hi";
var y = 5;
console.log(isNaN(x + y)); //true

var x = "5";
var y = 5;
console.log(isNaN(x + y)); //false
```

JavaScript Logical Operators

Oper	Name	Example
&&	AND	(x < 10 && y > 1) is true
||	OR	(x === 5 || y === 5) is false
!	NOT	!(x === y) is true

JavaScript Comparison Operators

Operator	Description	Comparing	Returns
==	equal to	x == 8	false
		x == 5	true
		x == "5"	true
===	equal value and equal type	x === 5	true
		x === "5"	false
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== 5	false
		x !== "5"	true
		x !== 8	true
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	true

ما هو Casting في JavaScript

يعني: Converting a value from one data type to another data type

1. Implicit Casting (تلقائي)

JavaScript تقوم بالتحويل تلقائيًا.

مثال:

```
console.log("5" - 2);
```

2. Explicit Casting (تحويل يدوي)

هنا أنت تتحكم بالتحويل.

مثال:

```
let value = "123";
let num = Number(value);

console.log(num);
……
let value = "50px";
let num = parseInt(value);

console.log(num); // 50
……..
let value = "10.5";
let num = parseFloat(value);

console.log(num);
……..
let num = 100;
let text = String(num);

console.log(text);
```

شيء مهم جدًا في المقابلات

يسمى:

Truthy and Falsy Values

Falsy values في JavaScript:

هذه تعتبر false عند التحويل:

1. false 2. 0 3. ""
4. Null 5. undefined 6. NaN

ما هي Array في JavaScript؟

Array هي بنية بيانات (Data Structure) تخزن عدة قيم داخل متغير واحد.

```
let numbers = [10, 20, 30];

numbers[1] = 50;
console.log(numbers);
```

Method	Explanation (English)	Example
`push()`	Adds an element to the end of the array.	`arr.push(5)`
`pop()`	Removes the last element from the array.	`arr.pop()`
`shift()`	Removes the first element from the array.	`arr.shift()`
`unshift()`	Adds an element to the beginning of the array.	`arr.unshift(1)`
`includes()`	Checks if the array contains a value.	`arr.includes(10)`
`indexOf()`	Returns the index of a value.	`arr.indexOf(20)`
`join()`	Converts an array to a string.	`arr.join(",")`
`slice()`	Returns part of an array without modifying the original array.	`arr.slice(1,3)`
`splice()`	Adds or removes elements from the array.	`arr.splice(1,1)`
`concat()`	Merges arrays together.	`arr1.concat(arr2)`
`reverse()`	Reverses the array.	`arr.reverse()`
`sort()`	Sorts elements in the strings.	`arr.sort()`

** isArray(array)

slice vs splice (مهم جدًا)

slice لا يغير المصفوفة الأصلية

```
let arr = [1,2,3,4];

let result = arr.slice(1,3);
console.log(result);
```

splice يغير المصفوفة الأصلية

```
let arr = [1,2,3,4];

arr.splice(1,2);
console.log(arr);
```

Array Methods مهمة جدًا

Method	ماذا تفعل
`map()`	تعديل عناصر المصفوفة
`filter()`	تصفية عناصر معينة
`find()`	إيجاد عنصر
`forEach()`	تنفيذ كود على كل عنصر
`some()`	يتحقق إذا كان عنصر واحد يحقق شرط
`every()`	يتحقق إذا كل العناصر تحقق شرط
`reduce()`	يجمع القيم

ال map

map تمر على كل عنصر في products (وهو object)، ثم ترجع خاصية name من كل object، وتجمع هذه القيم داخل مصفوفة جديدة من نوع string لو كان string بتصير array of string لو كان numbers بتصير array of number

والكود يلي تحت صارت ترجع Array of Object

```
let result = products.map(product => {
   return {
      productName: product.name,
      productPrice: product.price
   };
});
```

كيف تعمل filter

الدالة filter() تمر على كل عنصر في المصفوفة، ويجب أن يرجع callback قيمة:

true ← يتم الاحتفاظ بالعنصر في المصفوفة الجديدة
false ← يتم حذف العنصر من النتيجة
الصيغة العامة:

```
array.filter(element => condition)
```

JavaScript ستقيّم هذه القيمة كـ truthy أو falsy.
لذلك سيتم الاحتفاظ فقط بالعناصر التي:
product.name ليست null
ليست undefined
ليست "" (string فارغ)
يعني بالنهاية بترجع List of Object

ماذا تفعل some

الدالة some تمر على عناصر المصفوفة (objects)، وتطبق الشرط على كل عنصر، فإذا وجدت عنصر واحد يحقق الشرط ترجع true، وإذا لم تجد أي عنصر ترجع false.

الدالة every() تتحقق من: هل كل عناصر المصفوفة تحقق الشرط؟ ودائما بترجع اما true or false

method	ماذا تتحقق
`some()`	هل يوجد **عنصر واحد** يحقق الشرط
`every()`	هل **كل العناصر** تحقق الشرط

الدالة reduce() تأخذ كل عناصر المصفوفة وتدمجها تدريجياً في نتيجة واحدة.
الصيغة:

```
array.reduce((accumulator, element) => {
  return newValue;
}, initialValue);
```

الجزء	معناه
accumulator	القيمة المتجمعة
element	العنصر الحالي
initialValue	القيمة الابتدائية

example:

```
let numbers = [10, 20, 32, 43, 4]
let totalPrice = numbers.reduce((sum, n) => {
   return sum+=n;
}, 0);

console.log(totalPrice);
```

Example:

```
let products = [
  { id: 1, name: "Laptop", price: 1200, inStock: true },
  { id: 2, name: "Mouse", price: 25, inStock: true },
  { id: 3, name: "Keyboard", price: 80, inStock: false },
  { id: 4, name: "Monitor", price: 300, inStock: true },
  { id: 5, name: "USB Cable", price: 10, inStock: true }
];

products.forEach(product => {
  console.log(product.name);
});
```

الكود يلي تحت تصفية المنتجات المتوفرة فقط.

```
let availableProducts = products.filter(product => product.inStock);

console.log(availableProducts);
```

الكود يلي تحت تعديل البيانات (مثلاً أخذ أسماء المنتجات فقط).

```
let productNames = products.map(product => product.name);

console.log(productNames);
["Laptop", "Mouse", "Keyboard", "Monitor", "USB Cable"]
```

الكود يلي تحت إيجاد منتج معين.

```
let product = products.find(p => p.id === 3);

console.log(product);
```

الكود يلي تحت هل يوجد منتج سعره أكبر من 1000؟

```
let expensiveProduct = products.some(p => p.price > 1000);

console.log(expensiveProduct); //true
```

الكود يلي تحت هل كل المنتجات متوفرة؟

```
let allAvailable = products.every(p => p.inStock);

console.log(allAvailable);
```

الكود يلي تحت حساب مجموع أسعار المنتجات.

```
let totalPrice = products.reduce((sum, product) => {
  return sum + product.price;
}, 0);

console.log(totalPrice);
```

مثلاً:
نريد مجموع أسعار المنتجات المتوفرة فقط.

```
let totalAvailablePrice = products
  .filter(p => p.inStock)
  .map(p => p.price)
  .reduce((sum, price) => sum + price, 0);

console.log(totalAvailablePrice);
```

هذا يسمى:

Method Chaining

وهو مهم جدًا في JavaScript الحديثة.

1 filter → نأخذ المنتجات المتوفرة فقط
2 map → نأخذ الأسعار فقط
3 reduce → نجمع الأسعار

3 مواضيع مهمة جدًا في JavaScript بالترتيب لأنهم مرتبطين ببعض:

1️⃣ Anonymous Functions
2️⃣ Callback Functions
3️⃣ Arrow Functions

1️⃣ **Anonymous Function** هي function بدون اسم.
An anonymous function is a function without a name.
عادةً تُستخدم عندما لا نحتاج إعادة استخدام الدالة.
الكود يلي تحت هي named function لانو الها اسم

```
function sayHello() {
  console.log("Hello");
}
```

هنا الدالة لا يوجد لها اسم anonymous function

```
let greet = function () {
  console.log("Hello");
};

greet();
```

مثال ثاني

```
setTimeout(function () {
  console.log("Hello after 2 seconds");
}, 2000);
```

2️⃣ **Callback Function** دالة يتم إرسالها كـ parameter لدالة أخرى.

A callback function is a function passed as an argument to another function and executed later.

مثال:

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}

function sayBye() {
  console.log("Goodbye");
}

greet("Ali", sayBye);
```

مثال ثاني:

```
function processUser(name, callback) {
  console.log("Processing " + name);
  callback();
}

processUser("Hossam", function () {
  console.log("Done");
});
```

كل هذه methods تستخدم **callback**:

1. map
2. filter
3. forEach
4. reduce

```
let numbers = [1,2,3];
numbers.forEach(function(n){
  console.log(n);
});
```

Arrow Functions 3

Arrow functions are a shorter syntax for writing functions in JavaScript.

الشكل التقليدي

```
function add(a,b){
  return a + b;
}
```

الكود يلي تحت باستخدام arrow function

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

والكود يلي تحت هو الشكل المختصر

```
let numbers = [1,2,3];
numbers.forEach(n => console.log(n));
```

هذه الأسئلة تأتي كثيرًا في JavaScript interviews؟

1 الفرق في this

- In a regular function, this depends on how the function is called.

example:

```
const person = {
  name: "Ali",
  sayHello: function () {
    console.log(this.name);
  }
};
person.sayHello(); //Ali
```

this → person

- Arrow functions do not create their own this. They use this from the surrounding scope (lexical this).

example:

```
const person = {
  name: "Ali",
  sayHello: () => {
    console.log(this.name);
  }
};
person.sayHello(); // Undefined
```

this ≠ person

2️⃣ **الفرق في arguments** في ال regular function بقدر استخدم ال argument بينما بال arrow ما بقدر استخدم

```
function sum() {
  console.log(arguments);
}

sum(10, 20, 30);
```

الناتج: [10 ,20 ,30]

ما هو arguments؟ هو object يحتوي جميع القيم الممررة للدالة. حتى لو لم نعرف parameters.

```
function sum() {
  return arguments[0] + arguments[1];
}

console.log(sum(5, 7));
```

في Arrow Function لا يوجد arguments

```
const sum = () => {
  console.log(arguments);
};

sum(10, 20);
ReferenceError: arguments is not defined
```

كيف نحل الخطأ عن طريق اضافة ...nums ك parameter الها

3️⃣ **الفرق في constructor**

Regular Function يمكن استخدامها كـ constructor مع new

```
example:
function Person(name) {
  this.name = name;
}

let p = new Person("Ali");

console.log(p.name);
```

بينما بال Arrow Function لا يمكن استخدام new

Feature	Regular Function	Arrow Function
`this`	depends on how it is called	lexical this
`arguments`	available	not available
`constructor`	can use `new`	cannot use `new`
syntax	longer	shorter

example:

```
const products = [
  { id: 1, name: "Laptop", price: 1200, category: "electronics", inStock: true },
  { id: 2, name: "Phone", price: 800, category: "electronics", inStock: true },
  { id: 3, name: "Shoes", price: 150, category: "fashion", inStock: false },
  { id: 4, name: "Watch", price: 200, category: "fashion", inStock: true },
  { id: 5, name: "Keyboard", price: 100, category: "electronics", inStock: true }
];
```

1 filter + Arrow Function نريد المنتجات المتوفرة .

```
const availableProducts = products.filter(product => product.inStock);

console.log(availableProducts); هون فلتر بحاجه الى دالة
```

2 map + Arrow Function نريد تخفيض السعر 10%

```
const discountedProducts = availableProducts.map(product => {
  return {
    ...product,
    price: product.price * 0.9
  };
});

console.log(discountedProducts);
```

3 find نريد إيجاد منتج معين.

```
const laptop = products.find(product => product.name === "Laptop");

console.log(laptop);
```

4 some هل يوجد منتج سعره أكثر من 1000؟

```
const expensiveProduct = products.some(product => product.price > 1000);

console.log(expensiveProduct);
```

5 **every هل كل المنتجات أقل من 2000؟**

```
const allAffordable = products.every(product => product.price < 2000);

console.log(allAffordable);
```

6 **reduce نريد مجموع أسعار المنتجات.**

```
const totalPrice = products.reduce((sum, product) => {
  return sum + product.price;
}, 0);

console.log(totalPrice);
```

7 **forEach نريد طباعة أسماء المنتجات.**

```
products.forEach(product => {
  console.log(product.name);
});
```

8 **Anonymous Function الدالة بدون اسم:**

```
setTimeout(function () {
  console.log("Order processed");
}, 2000);
```

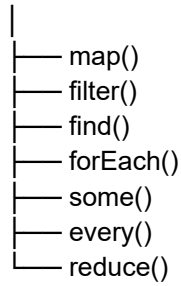
9 **Callback Function مثال واضح.**

```
function processOrder(order, callback) {
  console.log("Processing order:", order);

  callback();
}

processOrder("Laptop", function () {
  console.log("Order completed!");
});
```

```
Functions
│
├── Normal Function
│
├── Anonymous Function
│      function(){ }
│
├── Arrow Function
│      () => { }
│
└── Callback Function
       function passed as argument
```

Array Methods


في **JavaScript** الـ **Object** هو واحد من أهم أنواع البيانات، ويستخدم كثيراً في المشاريع لأنه يسمح لك بتخزين مجموعة من القيم المرتبطة ببعضها.

ببساطة: Object = مجموعة من الخصائص (properties) والقيم (values)

```
const person = {
  name: "Hossam",
  age: 25,
  job: "Developer"
};
```

الصيغة العامة :

```
const objectName = {
  key: value,
  key: value
};
```

كيف نصل إلى قيم الـ Object

يوجد طريقتان:

1 Dot Notation

```
console.log(person.name);
console.log(person.age);
```

2 Bracket Notation

```
console.log(person["name"]);
console.log(person["age"]);
```

** **إضافة Property جديدة** يعني بفترض انو ال property or key موجود هاي طريقة اضافة property جديد.

```
person.country = "Jordan";

console.log(person);
```

** **حذف Property**

```
delete person.job;
```

**** Object يحتوي Array**

```
const student = {
  name: "Ali",
  age: 20,
  skills: ["HTML", "CSS", "JavaScript"]
};

console.log(student.skills[1]);
```

**** Object يحتوي Object**

```
const user = {
  name: "Ahmad",
  address: {
    city: "Amman",
    street: "Main Street"
  }
};

console.log(user.address.city);
```

**** Object Method**

```
const person = {
  name: "Hossam",
  age: 25,
  greet: function () {
    console.log("Hello " + this.name);
  }
};

person.greet();
```

**** Loop على Object**

```
const car = {
  brand: "Toyota",
  model: "Camry",
  year: 2022
};

for (let key in car) {
  console.log(key, car[key]);
}
```

** تحويل Object إلى Array مفيد جداً في المشاريع.**

```
console.log(Object.keys(car));
```

** Object.values()**

```
console.log(Object.values(car));
```

** Object.entries()**

```
console.log(Object.entries(car));
```

** مثال واقعي من مشروع**

```
const product = {
  id: 1,
  name: "Laptop",
  price: 1200,
  category: "Electronics",
  inStock: true
};

console.log(product.name);
console.log(product.price);
```

الفرق بين Array و Object

Array	Object
يعتمد على index	يعتمد على key
[ ]	{ }
ترتيب مهم	ترتيب أقل أهمية

موضوعين مهمين في JavaScript ويستخدمان كثيراً في المشاريع:

1️⃣ **Nested Objects** كلمة Nested تعني Object داخل Object آخر أي أن الـ property value يمكن أن تكون Object آخر.

```
const user = {
  name: "Hossam",
  age: 25,
  address: {
    city: "Amman",
    country: "Jordan"
  }
};
console.log(user.address.city);
console.log(user.address.country);
```

* Nested أكثر من مستوى.

```
const company = {
  name: "Tech Corp",
  location: {
    country: "Jordan",
    city: "Amman",
    office: {
      floor: 5,
      room: 501
    }
  }
};
console.log(company.location.office.room);
```

2️⃣ **Destructuring Assignment** هو طريقة لاستخراج القيم من Object أو Array بسهولة.

بدل كتابة:

```
const user = {
  name: "Hossam",
  age: 25,
  country: "Jordan"
};

let name = user.name;
let age = user.age;
let country = user.country;
```

نستخدم Destructuring Assignment.

```
const user = {
  name: "Hossam",
  age: 25,
  country: "Jordan"
};

const { name, age, country } = user;

console.log(name);
console.log(age);
console.log(country);
```

** **تغيير اسم المتغير (Renaming)** أحياناً نريد اسم متغير مختلف.

```
const user = {
  name: "Hossam",
  age: 25
};

const { name: userName } = user;

console.log(userName);
```

** Default Value إذا لم تكن القيمة موجودة.

```
const user = {
  name: "Hossam"
};
const { name, age = 30 } = user;

console.log(age);
```

** **Destructuring Assignment مع Functions (مهم جداً)** يستخدم كثيراً في React و APIs.

```
function printUser({ name, age }) {
  console.log(name);
  console.log(age);
}

const user = {
  name: "Hossam",
  age: 25
};

printUser(user);
```

موضوع مهم جداً في JavaScript لأنه يفسر لماذا أحياناً تتغير القيم داخل الـ functions وأحياناً لا.

1️⃣ **Pass By Value** يعني تمرير نسخة (copy) من القيمة إلى الـ function أي أن القيمة الأصلية لا تتغير.

يحدث مع
number
string
boolean
null
undefined
symbol
bigint
هذه تسمى Primitive Data Types.

مثال:

```
let x = 10;

function changeValue(a) {
  a = 20;
}

changeValue(x);
console.log(x);
……
let name = "Hossam";

function changeName(n) {
  n = "Ali";
}
changeName(name);
console.log(name); // Hossam
```

2️⃣ **Pass By Reference** يعني تمرير مرجع (reference) للقيمة وليس نسخة منهاأي أن التعديل يؤثر على القيمة الأصلية.

يحدث مع

Object
Array
Function
هذه تسمى Reference Data Types.
مثال:

```
let user = {
  name: "Hossam"
};

function changeUser(u) {
  u.name = "Ali";
}

changeUser(user);

console.log(user.name);
```

ماذا حدث؟ بدلاً من نسخ القيمة يتم نسخ العنوان في الذاكرة (memory address).

التفسير:

user ⟶ { name: "Hossam" }

u ⟶ نفس ال object

مثال مهم (يُسأل كثيراً في المقابلات)

```
let obj = { value: 10 };

function test(o) {
  o.value = 20;
}
test(obj);
console.log(obj.value);
```

Primitive Types
→ Pass By Value
→ يتم إرسال نسخة

Reference Types
→ Pass By Reference
→ يتم إرسال المرجع

ما هو Closure ؟

هو: قدرة الـ function على تذكّر المتغيرات الموجودة في النطاق (scope) الخارجي حتى بعد انتهاء تنفيذ هذا النطاق.

ما هو Hoisting ؟

مصطلح Hoisting في JavaScript هو سلوك في اللغة يجعل الإعلانات (declarations) عن المتغيرات أو الدوال يتم رفعها (move conceptually) إلى أعلى الـ scope قبل تنفيذ الكود.

لكن مهم جدًا: الذي يتم رفعه هو التعريف (declaration) وليس القيمة (initialization).

1. ال Hoisting مع var

```
console.log(x);
var x = 5;
// undefined
```

طيب ليش لأن JavaScript تفسر الكود هكذا:

```
var x;      // تم رفع التعريف للأعلى (Hoisting)
console.log(x);
x = 5;
```

يعني: تم إنشاء المتغير x وقيمته الابتدائية undefined ثم بعد ذلك تم إعطاؤه القيمة 5

2. ال Hoisting مع const and let

```
console.log(a);
let a = 10;
// ReferenceError
```

طيب ليش؟ لأن let و const **يتم رفع تعريفهم ولكن لا يمكن استخدامهم قبل تعريفهم.**

هذه الفترة تسمى: TDZ يعني Temporal Dead Zone

يعني الكود يُفسَّر كأنه:

```
let a; // محجوز لكن غير مسموح استخدامه
console.log(a); // خطأ
a = 10;
```

3. ال Hoisting مع Function Declaration

```
sayHello();

function sayHello() {
  console.log("Hello");
}
//Hello
```

لماذا؟ لأن الدوال المعرفة بالطريقة التقليدية يتم رفعها بالكامل.

JavaScript تفسرها هكذا:

```
function sayHello() {
  console.log("Hello");
}
sayHello();
```

4. ال Hoisting مع Function Expression

```
sayHi();

var sayHi = function() {
  console.log("Hi");
};
// TypeError: sayHi is not a function
```

لماذا؟ لأن الذي تم رفعه فقط هو:

```
var sayHi;
```

فتصبح هكذا:

```
var sayHi;

sayHi();   // undefined()

sayHi = function() {
  console.log("Hi");
};
```

5. ال Hoisting مع Arrow Function

```
sayHello();

const sayHello = () => {
  console.log("Hello");
};
// ReferenceError
```

لماذا؟ لأن const داخل Temporal Dead Zone.

خلاصة ال Hoisting

النوع	هل يحدث Hoisting	هل يمكن استخدامه قبل التعريف
var	نعم	نعم لكن القيمة undefined
let	نعم	لا
const	نعم	لا
function declaration	نعم	نعم
function expression	فقط المتغير	لا
arrow function	فقط المتغير	لا

مثال يجمع كل شي

```
console.log(a);
var a = 5;

console.log(b);
let b = 10;

hello();

function hello() {
  console.log("Hello");
}
```

مصطلح Factory Functions

هو طريقة لإنشاء Objects باستخدام دالة (function) ترجع Object جديد كل مرة يتم استدعاؤها.

فكرتها بسيطة جدًا: الدالة تعمل مثل مصنع (Factory) يصنع Objects.

```
function createUser(name, age) {
  return {
    name: name,
    age: age,
    greet() {
      console.log("Hello " + name);
    }
  };
}
```

الاستخدام:

```
let user1 = createUser("Ali", 25);
let user2 = createUser("Ahmad", 30);

user1.greet();
user2.greet();
```

طريقة أقصر (Object Shorthand)

في JavaScript الحديثة:

```
function createUser(name, age) {
  return {
    name,
    age,
    greet() {
      console.log("Hello " + name);
    }
  };
}
```

الفرق بين Factory Function و Constructor Function

Factory Function

```
 function createUser(name) {
  return {
    name
  };
}
let u = createUser("Ali");
```

لا نستخدم new

Constructor Function

```
function User(name) {
  this.name = name;
}
let u = new User("Ali");
```

ال Factory Function مع Arrow Function

يمكن كتابتها هكذا:

```
const createUser = (name, age) => ({ name, age });
let user1 = createUser('Ali', 22);
console.log(user1.name);
console.log(user1.age);
```

مثال مهم يوضح فكرة ال Factory

```
function createCar(brand, model) {
  return {
    brand,
    model,
    info() {
      console.log(brand + " " + model);
    }
  };
}
let car1 = createCar("BMW", "X5");
let car2 = createCar("Toyota", "Corolla");

car1.info();
car2.info();
……
const people = {
  person1: {
    firstName: "John",
    lastName: "Doe"
  },
  person2: {
    firstName: "Mark",
    lastName: "Smith"
  }
};

const lookFor = function (first, last) {
  for (const key in people) {

    const person = people[key];

    if (person.firstName === first && person.lastName === last) {
      return first + " " + last + " is found";
    }
```

```
  }

  return first + " " + last + " is not found";
};

console.log(lookFor("John", "Doe"));
// John Doe is found

console.log(lookFor("Jane", "Doe"));
// Jane Doe is not found
```

……

من أهم الركائز اللي بتخليك تفهم كيف البيانات بتتصرف في الذاكرة، وبتوفر عليك "صداع" برمجي كبير في المستقبل إذا استوعبتها صح.

ببساطة، الكلمتين معناهم القابلية للتغيير:

Mutable: يعني "كائن قابل للتغيير". تقدر تعدل على محتواه بعد ما تنشئه بدون ما تغير مكانه في الذاكرة.

Immutable: يعني "كائن غير قابل للتغيير". بمجرد ما تنشئه، مستحيل تعدل عليه. إذا حاولت تغيره، الكمبيوتر فعلياً بيعمل "نسخة جديدة" بالقيمة الجديدة.

1. الفرق الجوهري (كيف يشوفها الكمبيوتر)
تخيل الـ Immutable مثل "الرقم"؛ لو عندك رقم 5 وبدك تخليه 6، أنت ما بتعدل على الرقم 5 نفسه، أنت بتبدله برقم جديد تماماً.
أما الـ Mutable مثل "صندوق"؛ الصندوق بضل هو نفسه، بس بتقدر تفتح الغطاية وتغير الأشياء اللي جواته.

2. في لغة جافاسكريبت (JavaScript)
جافاسكريبت بتقسم أنواع البيانات لقسمين واضحين جداً من هاد المنطلق:

الأنواع البدائية Immutable= Primitives
كل الأنواع الأساسية في JS هي Immutable. تشمل:

String, Number, Boolean, null, undefined, Symbol.

مثال:

```
let name = "Ahmad";
name[0] = "O"; // حاولت أغير أول حرف
console.log(name); // لا يتغير String لأن الـ "Ahmad" رح تضل
```

لما تعمل name = "Omar"، أنت ما غيرت الكلمة القديمة، أنت خليت المتغير name يأشر على مكان جديد في الذاكرة فيه كلمة "Omar".
الأجسام والمصفوفات (Objects & Arrays) = Mutable
هون "اللعب" بصير. الأوبجكت والمصفوفة هم Mutable بطبيعتهم.

مثال:

```
let user = { name: "Ahmad" };
user.name = "Omar"; // هون فعلياً غيرنا المحتوى داخل نفس المكان في الذاكرة
console.log(user.name); // "Omar"
```

3. "الفخ" المشهور في جافاسكريبت (Reference)

لأن الأوبجكت Mutable، لما تنسخه أنت فعلياً بتنسخ "العنوان" تاعه في الذاكرة مش القيمة نفسها:

```
let list1 = [1, 2, 3];
let list2 = list1; // هون list2 صارت تأشر على نفس المكان

list2.push(4);

console.log(list1); // [1, 2, 3, 4] !!
// لاحظ إن list1 تغيرت مع إننا عدلنا على list2
```

4. كيف تتعامل مع الـ Immutability في JS؟

في البرمجة الحديثة (خاصة مع React)، بنفضل الـ Immutability لأنها بتقلل الأخطاء. عشان تعدل على "أوبجكت" بدون ما تخرب الأصلي، بنستخدم الـ Spread Operator:

للمصفوفات: const newList = [...oldList, 4];

للأجسام: const newUser = { ...oldUser, name: "Omar" };

ملاحظة مهمة: استخدام const لا يعني أن القيمة Immutable. const بتمنعك تعيد تعريف المتغير (re-assignment) بس بتسمحلك تغير محتوى الـ Object إذا كان Mutable.

.....

1 المفهوم العام

Mutable (قابل للتغيير)

يعني يمكنك تعديل القيمة بعد إنشائها.

التغييرات تحصل على نفس المكان في الذاكرة.

غالبًا سريع، لكنه قد يسبب bugs لو مش مراقب.

Immutable (غير قابل للتغيير)

يعني لا يمكن تعديل القيمة بعد إنشائها.

أي تغيير يولّد نسخة جديدة.

أمان أكثر للـ state و multi-threading.

2 Mutable vs Immutable في JavaScript

◆ القيم الأساسية (Primitive types)

numbers, strings, booleans, null, undefined, symbol, bigint

كلها immutable

```
let x = 10;
let y = x;
```

```
y = y + 1;
console.log(x); // 10 -> ما تغيرت
```

هنا y تغيرت، لكن x نفسها ما تغيرت.

🔹 القيم المرجعية (Reference types)
objects, arrays, functions
كلها mutable عادة

```
let obj = {name: "Ali"};
let obj2 = obj;
obj2.name = "Omar";
console.log(obj.name); // "Omar" -> نفس المكان بالذاكرة
```

هنا التغيير على obj2 أثر على obj لأنهم يشيروا لنفس المكان بالذاكرة.

3️⃣ علاقة الـ Immutable بالـ React

في React:

state يجب أن يكون immutable
كل مرة تعمل React، update تنتظر نسخة جديدة من الـ state لتعرف أنه حصل تغيير.
لو عدّلت الـ state مباشرة (mutable) مثل player.score++، React قد لا تعمل re-render صح.

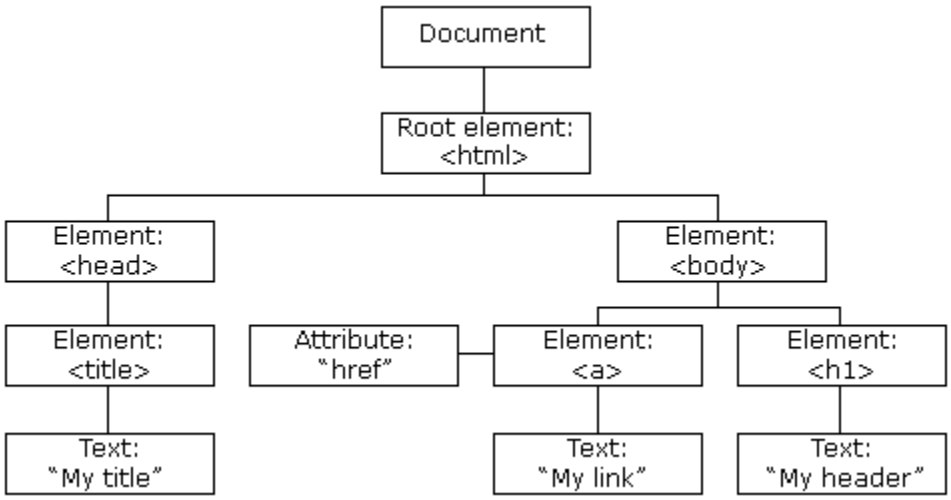
```
// الصح
setPlayer({...player, score: player.score + 1}); // إنشاء نسخة جديدة

// الغلط
player.score++; // ممكن <- عدلت القديم React ما تلاحظ التغيير
```

4️⃣ نصائح عملية في JavaScript

الأشياء الأساسية (numbers, strings, booleans) دائماً آمنة.
الأشياء المعقدة (objects, arrays) استخدم spread operator أو Object.assign لتكون immutable.
عند العمل مع React أو Redux: لا تعدّل state مباشرة أبدًا.

ال DOM هي اختصار ل Document Object Model



* 4 طرق رئيسية للوصول للعناصر؟

1. الوصول عبر الـ ID (الأكثر استخداماً)
هذه الطريقة هي الأسرع والأكثر دقة لأن الـ ID يجب أن يكون فريداً لو عندي اكثر من tag الهم نفس ال id هون رح يجب فقط اول واحد.

```
// الوصول لعنصر واحد فقط
let element = document.getElementById("header-title");
element.innerHTML = "أهلاً بك في أكاديمية ترميز";
```

2. الوصول عبر اسم الكلاس (Class Name)
تُرجع هذه الطريقة مصفوفة (أو ما يشبه المصفوفة) لأن الكلاس قد يتكرر.

```
// تُرجع قائمة بكل العناصر التي تحمل هذا الكلاس
let items = document.getElementsByClassName("list-item");

// لتغيير العنصر الأول في القائمة
items[0].style.color = "red";
```

3. الوصول عبر اسم التاج (Tag Name)
تُستخدم للوصول لجميع العناصر من نوع معين (مثل كل الـ h1 أو كل الـ p).

```
let paragraphs = document.getElementsByTagName("p");
paragraphs[1].innerHTML = "تم تغيير الفقرة الثانية";
```

4. الطريقة الحديثة والمجال الواسع (Query Selector)

تسمح لك باستخدام سينتاكس الـ CSS للوصول للعناصر، وهي مرنة جداً.

```
// الوصول لأول عنصر فقط يطابق الوصف
let firstBtn = document.querySelector(".main-container #submit-btn");
// الوصول لكل العناصر التي تطابق الوصف (ترجع مصفوفة)
let allButtons = document.querySelectorAll("button");
```

تغيير النصوص إلى أداة تتحكم في شكل العناصر (Style) و خصائصها (Attributes)؟

1. تعديل الـ Style (التنسيق)

يتم ذلك عبر خاصية .style. تذكر دائماً أن الأسماء التي تحتوي على شرطة في CSS (مثل background-color) تتحول في الجافا سكريبت إلى طريقة CamelCase (مثل backgroundColor).

```
let element = document.getElementById("my-div");

// تغيير لون الخلفية
element.style.backgroundColor = "blue";

// تغيير الحجم والخط
element.style.fontSize = "25px";
element.style.color = "white";

// إضافة حواف
element.style.border = "2px solid black";
```

2. تعديل الـ Attributes (الخصائص)

الـ Attributes هي الخصائص الموجودة داخل تاغ الـ HTML (مثل src للصور، href للروابط، أو value للمدخلات). هناك طريقتان للتعامل معها:

أ. الطريقة المباشرة (Direct Access):

```
let myImage = document.getElementById("main-img");
let myInput = document.getElementById("user-input");

// تغيير مصدر الصورة
myImage.src = "new-image.jpg";

// تغيير قيمة حقل نصي
myInput.value = "تم الكتابة بواسطة JS";
```

ب. استخدام setAttribute: وهي الطريقة الأكثر رسمية وتعمل مع أي خاصية:

```
let link = document.querySelector("a");

// تغيير الرابط وفتحه في صفحة جديدة
link.setAttribute("href", "https://google.com");
```

```
link.setAttribute("target", "_blank");

// حذف خاصية معينة
link.removeAttribute("class");
```

ال html events ؟

1. الطريقة الأولى: من داخل الـ HTML Inline Events هذه هي الطريقة الأبسط، حيث نكتب اسم الحدث كأنه خاصية (Attribute) داخل التاج.

```
<button onclick="alert('اضغط هنا')">تم الضغط!</button>

<button onclick="sayWelcome()">ترحيب</button>

<script>
  function sayWelcome() {
    console.log("أهلاً بك في أكاديمية ترميز");
  }
</script>
```

2. الطريقة الثانية: من داخل الجافا سكريبت (Event Listeners)
هذه الطريقة هي الأكثر احترافية لأنها تفصل المنطق البرمجي عن التصميم، وتسمح لك بإضافة عدة أحداث لنفس العنصر.

```
let myBtn = document.getElementById("main-btn");

// نضيف "مراقب" للحدث
myBtn.addEventListener("click", function() {
  console.log("تم استدعاء الحدث من الجافا سكريبت");
  document.body.style.backgroundColor = "lightblue";
});
```

3. أشهر الأحداث (Common Events) التي ركز عليها الدرس:
onclick: عند الضغط على العنصر.
onmouseover: عند مرور الفأرة فوق العنصر.
onmouseout: عند خروج الفأرة من العنصر.
onchange: تُستخدم بكثرة مع حقول الإدخال (Inputs) عند تغيير القيمة.
onload: عند اكتمال تحميل الصفحة أو الصورة.

مثال عملي:

```
<div id="box"
    style="width:100px; height:100px; background:red;"
    onmouseover="changeColor('green')"
    onmouseout="changeColor('red')">
</div>
```

```
<script>
   function changeColor(colorName) {
      let box = document.getElementById("box");
      box.style.backgroundColor = colorName;
   }
</script>
```

ملخص الكود لأهم العمليات (إنشاء، إضافة، وحذف):

1. إنشاء عنصر جديد (Creating Elements)

تستخدم دالة createElement لصناعة "تاغ" في ذاكرة المتصفح، لكنه لا يظهر في الصفحة حتى تخبره أين يذهب.

```
// 1. إنشاء العنصر مثلاً فقرة p
let newP = document.createElement("p");

// 2. إضافة محتوى نصي له
newP.innerHTML = "هذا عنصر جديد تم إنشاؤه بالجافا سكريبت";

// 3. إضافة ستايل أو كلاس اختياري
newP.style.color = "blue";
newP.classList.add("my-text");
```

2. إضافة العنصر إلى الصفحة (Adding Elements)

لكي يظهر العنصر، يجب أن نجعله "ابناً" لعنصر موجود مسبقاً (مثل الـ body أو div معين) باستخدام appendChild.

```
let container = document.getElementById("main-container");

// إضافة العنصر الجديد في نهاية الحاوية
container.appendChild(newP);
```

3. حذف عنصر (Removing Elements)

هناك طريقتان للحذف، إما أن يحذف العنصر نفسه، أو يطلب الأب حذف أحد أبنائه.

```
let elementToRemove = document.getElementById("old-text");

// الطريقة الحديثة يحذف نفسه مباشرة
elementToRemove.remove();

// الطريقة الكلاسيكية عن طريق الأب
// parentElement.removeChild(childElement);
```

4. تطبيق عملي: نظام "قائمة المهام" (To-Do List)

هذا المثال يلخص الدرس كاملاً؛ حيث نصنع عنصر li جديد عند كل ضغطة زر:

```
function addTask() {
    // 1. الوصول للمكان الذي سنضيف فيه (الـ ul)
    let list = document.getElementById("task-list");

    // 2. إنشاء العنصر li
    let task = document.createElement("li");

    // 3. وضع النص داخل العنصر
    task.innerHTML = "مهمة جديدة";

    // 4. إضافة زر حذف بجانب كل مهمة
    let deleteBtn = document.createElement("button");
    deleteBtn.innerHTML = "حذف";
    deleteBtn.onclick = function() {
        task.remove(); // حذف المهمة عند الضغط على الزر
    };

    // 5. دمج الزر داخل المهمة، ثم إضافة المهمة للقائمة
    task.appendChild(deleteBtn);
    list.appendChild(task);
}
```

هذا سؤال مهم جداً في JavaScript 👌 وبيجي كثير بالمقابلات.

خليني أشرحلك الفرق بشكل بسيط وواضح:

- أولاً: className

هذا بيعطيك كل الكلاسات كنص (string)

مثال:

```
let div = document.querySelector("div");
```

```
console.log(div.className);
```

📌 النتيجة ممكن تكون:

```
"box active red"
```

🖍️ التعديل:

```
div.className = "newClass";
```

⚠️ المشكلة:

هذا يمسح كل الكلاسات القديمة ويستبدلها بالجديدة

🔹 ثانياً: classList

هذا بيعطيك كائن (object) فيه دوال للتحكم بالكلاسات بسهولة

أهم الدوال:
✅ إضافة كلاس: هذا رح يضيفو اخر شي ولو بدي اضيف مجموعة من الكلاسات بحط كل كلاس داخل single or double qutation ومفصلولين بفاصلة

```
div.classList.add("active");
```

❌ حذف كلاس:

```
div.classList.remove("active");
```

🔁 تبديل (toggle):

```
div.classList.toggle("active");
```

🔍 التحقق:

```
div.classList.contains("active");
```

الخاصية	className	classList
النوع	string	object
التحكم	صعب	سهل ومرن
يحذف الكلاسات القديمة؟	نعم	لا
الاستخدام	قديم	حديث ومفضل

💡 مثال يوضح الفرق
باستخدام className:

```
div.className += " active"; // ممكن يصير فيه مشاكل مسافات، تكرار
```

باستخدام classList:

```
div.classList.add("active"); // نظيف وآمن
```

- الفرق الأساسي بين <button> و <"input type="submit>

الخاصية	<"input type="submit>	<button>
الاستخدام داخل form	يُستخدم لإرسال الـ form (submit) بشكل افتراضي	يمكن أن يكون submit أو button عادي حسب `type`
النوع الافتراضي	always submit	`"type="submit` افتراضياً، لكن ممكن تختار `"type="button` أو `"type="reset`
المحتوى الداخلي	لا يمكن أن يحتوي على HTML داخله، فقط قيمة نصية من `value`	يمكن أن يحتوي على نصوص، صور، أي عناصر HTML
مرونة التصميم	أقل	أكثر (تقدر تحط أي تصميم داخله)

شو يعني Hoising في جافاسكريبت؟ مهمه في المقابلات وكمان موضوع ال synchronous
شو ال micro stack and macro stack
شو هم ال io operation وشو دخلهم بالباك ايند لل جافاسكريبت وكمان شو هو ال event loop

```
console.log(0 == false);
```

النتيجة: true

ليش؟
لأن == بيعمل type coercion (تحويل نوع البيانات)
يعني JavaScript بتحول false إلى 0، فتصير:

0 == 0 → true

بينما يلي تحت انا عارفها

```
console.log(0 === false);
```

```
0 == false   // true  (Loose)
0 === false  // false (Strict)
```