JavaScript

JavaScript Fundamentals

📦 ما هي المتغيرات (variables)؟

المتغيّر هو **صندوق بنحط فيه قيمة** (زي رقم، نص، أو حتى object).
بنستخدمه لما نحب نحتفظ بمعلومة ونعرف نرجع لها أو نغيرها بعدين.

📌 مثال:

```
let userName = "Ayaat";
```

✍️ أنواع الإعلان عن المتغيرات:

الكلمة المفتاحية	الاستخدام	ينفع تغيّر القيمة؟	ينفع تعيد تعريفه؟
`let`	متغير عادي	✅ ايوة	❌ لا
`const`	ثابت (ما يتغيرش)	❌ لا (لكن تقدر تغير محتوى الـ object)	❌ لا
`var`	قديم ومش مفضل	✅ ايوة	✅ ايوة (مش آمن)

✅ الأفضل دايمًا: استخدم `let` أو `const` ، وتجنب `var` هنعرف السبب في الفرق بينهم .

```
let age = 22;      // إعلان + تعيين قيمة
age = 23;          // تحديث القيمة

const country = "Egypt"; // ثابت، مينفعش يتغير
// country = "Canada"; ❌ Error
```

🧠 امته استخدم let or const :

- `let` : لو هتغير القيمة بعدين.
- `const` : لو القيمة مش هتتغير أبدًا.

📌 قاعدة مهمة:

> **"Use `const` when you can, and `let` when you have to."**

📌 أمثلة عملية:

مثال 1 – عداد الضغط على زر:

```
let count = 1;

button.onclick = () => {
  count += 1;
  console.log(`${count} times clicked`);
};
```

مثال 2 – Object وتحديث خاصية:

```
const user = { name: "Suzan" };
user.name = "Sara"; // ✅ مسموح
```

⚠️ ملاحظات هامة:

- ال JavaScript **لغة ديناميكية**: مش لازم تقول نوع المتغير (زي string أو number).
- المتغيرات **بتفرق بين Uppercase و lowercase** :
  `userName` ≠ `username`
- مينفعش تعمل اسم متغير يبدأ برقم أو يكون كلمة محجوزة زي `let` أو `for` .

✅ نصائح لاختيار اسم متغير جيد:

- يكون واضح ومعبّر: `userScore` أفضل من `x`
- استخدم `totalScore` ، `firstName` : **camelCase**
- متستخدمش رموز أو أرقام في البداية

مصطلحات مهمة :

المصطلح	المعنى المبسط
Declare	إنشاء متغير
Initialize	تعيين قيمة أولى
Update	تغيير القيمة
Const	ثابت لا يمكن تغييره
Let	متغير قابل للتغيير

⚖️ الفرق بين `const` , `let` , `var` :

1. 🧠 نطاق الاستخدام – Scope

- ال `var` : بيشتغل داخل **الدالة (function)** فقط , يعنى لو عرفته داخل block scope {} هايكون متاح عادي برا scope لكن مش خارج function scope.

- ال `let` و `const` : بيشتغلوا داخل **البلوك {}** زي `while` , `for` , `if` .

📌 مثال:

```
function test(){
  if (true) {
  var x = "I am var";
  let y = "I am let";
  const z = "I am const";
    }
  console.log(x); // ✅ I am var
  console.log(y); // ❌ Error
  console.log(z); // ❌ Error
}
console.log(x); // ❌ Error
console.log(y); // ❌ Error
console.log(z); // ❌ Error
```

2. 🔁 إعادة التعريف – Redeclaration

- `var` : ينفع تعيد تعريف المتغير بنفس الاسم.
- `let` و `const` : مش هينفع تعيد تعريفه بنفس الاسم في نفس النطاق.

```
var name = "Ayaat";
var name = "Sara"; // ✅ عادي

let age = 22;
let age = 23; // ❌ Error

const country = "Egypt";
const country = "Morocco"; // ❌ Error
```

3. 🔼 الرفع – Hoisting

- ال `var` بيترفع (hoisting) لكن قيمته بتبقى `undefined` لحد ما توصل لسطر التعيين.
- ال `let` و `const` بيترفعوا كمان، لكن بيدخلوا في **المنطقة الميتة الزمنية TDZ**، يعني مش مسموح تستخدمهم قبل الإعلان عنهم.

> ال (Temporal Dead Zone) TDZ= المنطقة الزمنية الميتة، يعني متغير موجود بس مينفعش تستخدمه قبل ما تعلن عنه.

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

console.log(b); // ❌ Error: Cannot access 'b' before initialization
let b = 10;

console.log(c); // ❌ Error: Cannot access 'c' before initialization
const c = 15;
```

4. 🔁 إعادة التعيين – Reassignment

- ال `var` و `let` : ممكن تغير القيمة بعدين.
- ال `const` : لا يمكن تغييره، لكن لو القيمة عبارة عن object أو array، ممكن تغير **المحتوى** مش العنوان.

```
const person = { name: "Ayaat" };
person.name = "Sara"; // ✅ عادي
// person = { name: "Laila" }; // ❌ Error
```

5. 🔄 تأثيرهم داخل الـ for loop

> في الـ `let`، `for` loop بيخلق نسخة جديدة من المتغير في كل دورة، لكن `var` بيستخدم نسخة واحدة بس.

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log("var:", i), 1000);
}
// ➤ var: 3 (3 مرات)

for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log("let:", i), 1000);
}
// ➤ let: 0
// ➤ let: 1
// ➤ let: 2
```

6. 👄 التعامل مع الكائن `window` (في المتصفح)

> المتغيرات اللي بتتكتب بـ `var` بتتخزن في الكائن `window`، لكن `let` و `const` لا.

```
var x = 10;
let y = 20;
```

```
const z = 30;

console.log(window.x); // ✅ 10
console.log(window.y); // ❌ undefined
console.log(window.z); // ❌ undefined
```

🔎 ليه ده مهم؟ عشان لو بتكتب كود في المتصفح ومش عايز يحصل تضارب مع متغيرات النظام أو إضافات المتصفح، استخدم `let` أو `const`.

7. 🚨 Strict mode تلقائي في `let` و `const`

- استخدام `let` أو `const` بيشجعك على كتابة كود آمن ونظيف تلقائيًا.
- أما `var` ، فبيسمحلك تكتب حاجات ممكن تكون خطأ بدون ما يحذرك.

```
undeclaredVar = 5; // ❌ ما كتبناش var أو let أو const

console.log(undeclaredVar); // هيشتغل في var، لكن خطأ في let/const
```

📝 Variable Naming Conventions in JavaScript

✳️ المقصود بـ "naming convention":

يعني "القواعد والأسلوب" اللي بنسمي بيه المتغيرات (variables) في الكود علشان تكون:

- مفهومة
- واضحة
- متناسقة

✅ القواعد الأساسية لتسمية المتغيرات:

1. ✅ Use camelCase

- يعني تبدأ الكلمة الأولى بحرف صغير، وكل كلمة بعدها تبدأ بحرف كبير.

```
let userName;
let totalScore;
let isLoggedIn;
```

2. ❌ مينفعش تبدأ الرقم

```
let 1user;   // ❌ Error
let user1;   // ✅ مسموح
```

3. ❌ مينفعش يكون فيه مسافات أو علامات غريبة

```
let user name;     // ❌ Error
let user-name;     // ❌ Error
let user_name;     // ✅ مسموح بس camelCase أفضل
```

4. ❌ مينفعش تستخدم كلمات محجوزة

زي: `return` , `function` , `for` , `if` , `const` , `let` , `var` و غيره.

```
let for = 5;  // ❌ Error
```

💡 نصائح لكتابة أسماء متغيرات احترافية:

النصيحة	مثال جيد	مثال سيئ
استخدم أسماء واضحة	`userEmail`	`x`
عبّر عن نوع البيانات	`isActive` (Boolean)	`data1`
تجنب الكلمات الطويلة جدًا	`totalItemsCount`	`thisIsAVeryLongVariableNameThatNoOneCanRead`
خليك متناسق في كل الكود	استخدم نفس النمط (camelCase مثلاً)	

✅ متى تستخدم أسماء معينة؟

النوع	يفضل تبدأ بـ	مثال
Boolean (صح/غلط)	is / has / can	`isLoggedIn` , `hasPermission` , `canEdit`
عدد / قيمة	اسم + نوع	`userCount` , `totalPrice`
نص	اسم + Text / Name	`userName` , `errorMessage`
Array	اسم جمع	`users` , `items` , `colors`
Function	فعل (verb)	`getUserName()` , `calculateTotal()`

💬 مثال توضيحي (كود مرتب ومحترف):

```
let firstName = "Ayaat";
let isStudent = false;
let totalPoints = 90;
let colors = ["pink", "brown", "black"];
let user = { name: "Ayaat", age: 23 };
```

💎 أدوات مهمة لتحسين كتابة الكود:

1. 🧹 Prettier – منسق الكود (Code Formatter)

> مسؤولة عن "تنسيق الكود" تلقائيًا. يعني:

- ترتيب المسافات
- تنظيم الأقواس
- تنسيق الأسطر الطويلة
- إزالة الفوضى

📌 مثال قبل Prettier:

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

📌 بعد Prettier:

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

2. 🔍 ESLint – تحليل الكود (Linter)

> تبحث عن أخطاء محتملة أو أسلوب كتابة غير جيد وتعرض لك تحذيرات أو أخطاء.

✅ أمثلة لما يعمله ESLint:

- تحذيرك إنك نسيت `let` أو `const`
- منع استخدام `==` بدل `===`
- يشتكي لو فيه متغير متعرف ومش مستخدم
- يوضح لك الأماكن اللي ممكن يكون فيها كود غير نظيف أو خطير

📌 مثال:

```
x = 5; // ❌ ESLint هيقولك: "x is not defined"
```

🛠️ إزاي تستخدمهم في مشروعك (خطوات عملية):

🔷 الطريقة الأسرع مع VS Code:

1. ثبت الإضافتين:

- Prettier - Code formatter
- ESLint

2. في إعدادات VS Code:

✅ فعلّ:

- `"editor.formatOnSave": true` (علشان Prettier حفظ تشتغل لما تعملي)
- `"eslint.format.enable": true`

3. أنشئي ملف إعدادات:

في جذر المشروع (root folder)، أنشئ ملفين:

`.prettierrc`:

```
{
  "semi": true,
  "singleQuote": true,
  "tabWidth": 2,
  "trailingComma": "es5"
}
```

`.eslintrc.json`:

```
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": ["eslint:recommended"],
  "parserOptions": {
    "ecmaVersion": 12,
    "sourceType": "module"
  },
  "rules": {
    "no-unused-vars": "warn",
    "no-console": "off",
```

```
    "eqeqeq": "error"
  }
}
```

🌟 ليه تستخدمهم مع بعض؟

- Prettier = شكل الكود حلو ومرتب.
- ESLint = الكود سليم وخالي من الأخطاء.

📌 تلميحة:

> في مشاريع React مثلًا، بيكون في إعداد جاهز اسمه:
> `eslint-config-prettier` — علشان يخليهم يشتغلوا مع بعض بدون تعارض.

📚 أنواع البيانات في JavaScript

تنقسم إلى نوعين رئيسيين:

الفئة	الاسم بالإنجليزية	النوع
🟢 أنواع بسيطة	**Primitive Types**	غير قابلة للتغيير (Immutable)
🔵 أنواع غير بسيطة	**Reference Types** أو **Non-Primitive Types**	قابلة للتغيير (Mutable)

🟢 أولًا: Primitive Types (أنواع بسيطة)

> هي القيم اللي بتُخزن مباشرة في الذاكرة، ولها حجم ثابت.

✅ الأنواع الـ 7 الأساسية:

النوع	مثال	شرح
`string`	`"Ayaat"`	نص
`number`	`22` , `3.14`	أرقام (صحيحة أو عشرية)
`boolean`	`true` , `false`	قيم منطقية (نعم / لا)
`undefined`	`let x;`	قيمة غير مُعرفة
`null`	`let y = null;`	لا يوجد شيء (فارغة عن قصد)
`bigint`	`1234567890123456789012345678901234567890n`	أرقام كبيرة جدًا
`symbol`	`Symbol("id")`	لإنشاء معرف فريد

📌 **خصائص:**

- غير قابلة للتعديل (immutable)

- تُخزن كـ قيمة مباشرة

🔵 ثانيًا: Non-Primitive Types (أنواع غير بسيطة)

> هي **كائنات (Objects)** تُخزن كـ مراجع (references) في الذاكرة.

✅ الأنواع الأشهر:

النوع	مثال	شرح
`object`	`{name: "Ayaat", age: 22}`	كائن يحتوي على خصائص
`array`	`[1, 2, 3]`	مصفوفة تحتوي على مجموعة من القيم
`function`	`function sayHi() {}`	دالة يمكن استدعاؤها

📌 **خصائص:**

- قابلة للتعديل (mutable)
- تُخزن كـ مرجع (reference)
- ممكن تحتوي على بيانات متعددة بداخلها

String :

🧵 1. ما هو الـ `String` ؟

> `String` هو نوع بيانات (primitive type) يُستخدم لتمثيل النصوص.

✅ أمثلة:

```
let name = "Ayaat";
let greeting = 'Hello';
let message = `Welcome!`; // باستخدام backticks
```

📌 Strings في JavaScript:

- **Primitive** (وليست كائنات)
- **Immutable** = (لا يمكن تغيير قيمتها بعد إنشائها (لكن ممكن تنشئ نسخة جديدة بتعديل معين
-

🔗 2. الفرق بين Concatenation و Interpolation

🔶 Concatenation (الربط باستخدام `+`):

> بدمج النصوص مع بعض عن طريق علامة زائد `+`

```
let name = "Ayaat";
let msg = "Hello " + name + "!"; // "Hello Ayaat!"
```

🔷 Interpolation (الربط الذكي باستخدام `${}`):

> تُستخدم مع **template literals** (باستخدام `backticks`)

```
let name = "Ayaat";
let msg = `Hello ${name}!`; // "Hello Ayaat!"
```

📌 الفرق:

الخاصية	Concatenation	Interpolation
الشكل	`+`	`` داخل `${}`
الأسهل في الاستخدام؟	❌	✅
يدعم النصوص متعددة الأسطر؟	❌	✅

🧰 3. أهم String Methods (مع شرح مبسط)

كل الطرق دي **ما بتعدلش** النص الأصلي، بل بترجع نسخة جديدة.

الدالة	شرح مبسط	مثال عملي
`length`	ترجع عدد الحروف في النص	`text.length`
`charAt(index)`	ترجع الحرف في مكان معين	`text.charAt(0)` → H
`charCodeAt(index)`	ترجع الكود الرقمي (UTF-16) للحرف في المكان المحدد	`"A".charCodeAt(0)` → 65
`codePointAt(index)`	زي اللي فوق لكن يدعم الرموز التعبيرية	`"😊".codePointAt(0)`
`at(index)`	بديل حديث لـ `charAt` ويدعم الأرقام السالبة	`text.at(-1)` → آخر حرف
`text[index]`	طريقة للوصول لحرف كأنها مصفوفة	`text[0]` → H

✂️ تقطيع النص:

الدالة	وظيفتها	مثال
`slice(start, end)`	تقطع جزء من النص (الـ end غير شامل) ويمكن استخدام أرقام سالبة	`text.slice(7, 13)`
`substring(start, end)`	زي `slice` لكن الأرقام السالبة تتحول إلى 0	`text.substring(7, 13)`

الدالة	وظيفتها	مثال
`substr(start, length)`	تقطع عدد معين من الحروف (مهجورة، لا يُنصح بها)	`text.substr(7, 5)`

🔠 تغيير الحالة:

الدالة	وظيفتها	مثال
`toUpperCase()`	تحويل إلى حروف كبيرة	`text.toUpperCase()`
`toLowerCase()`	تحويل إلى حروف صغيرة	`text.toLowerCase()`

🖇 الدمج:

الدالة	وظيفتها	مثال
`concat()`	دمج عدة نصوص مع بعض	`"Hello".concat(" World")`
`+`	يمكن استخدام علامة `+` أيضًا	`"Hello" + " World"`

🧹 إزالة الفراغات:

الدالة	وظيفتها	مثال
`trim()`	حذف الفراغات من البداية والنهاية	`text.trim()`
`trimStart()`	حذف من البداية فقط	`text.trimStart()`
`trimEnd()`	حذف من النهاية فقط	`text.trimEnd()`

📦 الحشو:

الدالة	وظيفتها	مثال
`padStart(len,str)`	يضيف حشو (مثل صفر) في البداية حتى يصل لطول معين	`"5".padStart(3, "0")` → "005"
`padEnd(len,str)`	يضيف حشو في النهاية حتى يصل لطول معين	`"5".padEnd(3, "0")` → "500"

🔁 التكرار:

الدالة	وظيفتها	مثال
`repeat(n)`	تكرار النص `n` مرة	`"ha".repeat(3)` → "hahaha"

🔁 الاستبدال:

الدالة	وظيفتها	مثال
`replace(a, b)`	استبدال أول جزء مطابق	`text.replace("hi", "hello")`
`replaceAll(a, b)`	استبدال كل الأجزاء المطابقة (أحدث طريقة)	`text.replaceAll("hi", "hello")`

> ملاحظة: `replace()` يبدل أول جزء فقط، بينما `replaceAll()` يبدل الكل.

🧮 تحويل النص إلى مصفوفة:

الدالة	وظيفتها	مثال
`split(",")`	تقسم النص إلى أجزاء حسب الفاصل	`"a,b,c".split(",")` → `["a","b","c"]`
`split("")`	تقسم كل حرف وحده	`"hi".split("")` → `["h","i"]`

🧠 إضافات مهمة جدًا:

✅ هل `String` نوع primitive ولا object؟

- `String` نوع **primitive**
- لكن لما تستخدم methods زي `.slice()` أو `.charAt()`، JavaScript **بتحوّلها مؤقتًا** لكائن String object عشان تقدر تشتغل عليها، ثم ترجع primitive تاني.

```
let str = "  Hello Ayaat!  ";

console.log(str.trim().toUpperCase().slice(6, 11));
// ➤ "AYAAT"
```

🧠 Problem Solving (String Methods Challenge):
You are given a string that contains a name, followed by extra spaces and punctuation.
Write a function that:

1. Trims the string (remove spaces from both sides).
2. Replaces all commas `,` with hyphens `-`.
3. Converts all letters to lowercase.
4. Splits the result into an array of words.
5. Returns the length of the first word and the array itself.

Example input:

```
let messyString = "   Hello,My,Name,Is,Ayaat   ";
```

Solution :

```
function processString(messyString){

    // step 1: remove spaces from both sides

    let trimmedString=messyString.trim() ;

    // step 2: replace all commas with hyphens

    let replacedString=trimmedString.replaceAll(',' , '-');

    // step 3: convert all letters to lowercase

    let loweredString=replacedString.toLowerCase();

    // step 4: split the result into an array of words

    let wordsArray=loweredString.split('-');

    // step 5: return the length of the first word and the array itselt

    let firstwordLenght=wordsArray[0].length;

    return [firstwordLenght, wordsArray];
}

// Example usage:

let messyString= "  Hello,My,Name,Is,Ayaat  ";

console.log(processString(messyString));
```

🧮 أولاً: JavaScript Numbers

✅ نوع واحد فقط:

جافاسكريبت فيها نوع واحد للأرقام (Number)، وده بيشمل:

- الأعداد الصحيحة (integers): مثل `5`
- الأعداد العشرية (floats): مثل `5.7`

🧪 الكتابة العلمية للأرقام (Scientific Notation)

تقدري تكتبي أرقام ضخمة أو صغيرة جدًا باستخدام `e` :

```
let x = 123e5;   // 12300000 <= 5^10 × 123 يعني
let y = 123e-5;  // 0.00123 <= 5-^10 × 123 يعني
```

📏 الدقة (Precision)

1. دقة الأعداد الصحيحة:

- الأرقام بدون فواصل دقيقة حتى 15 رقم.

```
let x = 999999999999999;   // ✅ صحيح
let y = 9999999999999999;  // ❌ غير دقيق
```

2. دقة الأعداد العشرية:

العمليات الحسابية في الأعداد العشرية مش دايمًا دقيقة 100%:

```
let x = 0.1 + 0.2;  // ❌ 0.30000000000000004 بيطلع

// الحل الأفضل:
let fixed = (0.1 * 10 + 0.2 * 10) / 10;  // ✅ 0.3
```

➕ الجمع بين النصوص والأرقام

🟥 جمع نص + رقم أو رقم + نص = **نص**:

```
let x = "10";
let y = 20;
let z = x + y;  // "1020"
```

⚠️ مثال شائع للخطأ:

```
let x = 10;
let y = 20;
let z = "30";
let result = x + y + z;  // "3030"
```

🔢 التحويل التلقائي (Type Coercion)

ال JavaScript بتحاول تحوّل النص إلى رقم لو استخدمتي عمليات حسابية زي `* / -` :

```
let x = "100";
let y = "10";
let z = x / y;   // ✅ 10
let z2 = x - y;  // ✅ 90
let z3 = x * y;  // ✅ 1000

let z4 = x + y;  // ❌ "10010" (لأن + معناها دمج)
```

❌ NaN (Not a Number)

معناها: القيمة مش رقم صحيح.

```
let x = 100 / "apple";  // ❌ NaN

let y = 100 / "10";     // ✅ 10
```

♾️ Infinity

- لو قسمت رقم على صفر أو ضربت رقم كبير جدًا في نفسه → النتيجة = `Infinity`

```
let x = 2 / 0;   // ♾️
let y = -2 / 0;  // -♾️
```

🔢 القيم الست عشرية (Hexadecimal)

- أي رقم يبدأ بـ `0x` يعتبر رقم "هكس" (أساس 16):

```
let x = 0xFF;  // = 255
```

⚠️ الكائنات العددية (Number Object)

```
let x = 500;
let y = new Number(500);
```

❌ **لا يُفضل** استخدام `new Number()` لأنه:

- يبطئ الأداء
- ممكن يسبب مشاكل في المقارنة:

```
x == y   // true (مقارنة بالقيمة فقط)
x === y  // false (x قيمة - y كائن)
```

🟩 أولاً: Basic Number Methods

> الميثودز اللي بتتطبق على أي رقم (number) مباشرة، سواء كان رقم عادي أو متغير رقمي.

Method	الوصف	مثال	الناتج
`toString()`	يحوّل الرقم إلى نص (String)	`(123).toString()`	`"123"`
`toExponential()`	يعرض الرقم بصيغة علمية (exponential notation)	`(123456).toExponential()`	`"1.23456e+5"`
`toFixed(n)`	يضبط عدد الأرقام بعد العلامة العشرية (ويحوّل الناتج لنص)	`(3.14159).toFixed(2)`	`"3.14"`
`toPrecision(n)`	يحدد "إجمالي" عدد الأرقام المهمة (significant digits)	`(3.14159).toPrecision(4)`	`"3.142"`
`valueOf()`	يرجّع القيمة الحقيقية للرقم (تُستخدم نادرًا عند المقارنة أو التحويل)	`(123).valueOf()`	`123`

🔶 **ملحوظة مهمة:** `toFixed()` و `toPrecision()` يرجّعوا **نص** (string)، مش رقم.

🟨 ثانيًا: Static Number Methods

> دول بيشتغلوا على الكائن `Number` نفسه، مش على متغير معين. يعني تكتبيهم بالشكل ده:
> `Number.اسم_الميثود()`.

Method	الوظيفة	مثال	الناتج
`Number.isFinite(x)`	هل x عدد نهائي؟	`Number.isFinite(10/2)`	`true`
`Number.isInteger(x)`	هل x عدد صحيح؟	`Number.isInteger(10.5)`	`false`
`Number.isNaN(x)`	هل x هو NaN؟ (يعني "ليس رقمًا")	`Number.isNaN("abc"/3)`	`true`

Method	الوظيفة	مثال	الناتج
`Number.isSafeInteger(x)`	هل x عدد صحيح آمن؟ (ما بين ± (2^53 - 1))	`Number.isSafeInteger(9007199254740991)`	`true`
`Number.parseInt(x)`	يحوّل النص إلى عدد صحيح (integer)	`Number.parseInt("42.99")`	`42`
`Number.parseFloat(x)`	يحوّل النص إلى عدد عشري (float)	`Number.parseFloat("42.99")`	`42.99`

ما هو `Math` ؟

ال `Math` هو كائن built-in (موجود جاهز في اللغة)، بيحتوي على **دوال رياضية** جاهزة تستخدمها لما تحتاج تعمل عمليات زي التقريب، الجذر التربيعي، حساب القوة، الرقم العشوائي... إلخ.

🔢 أهم دوال Math في JavaScript

الدالة	شرح مبسط	مثال	الناتج
`Math.round(x)`	يقرب الرقم لأقرب عدد صحيح	`Math.round(4.6)`	`5`
`Math.ceil(x)`	يقرب الرقم لأعلى (سقف)	`Math.ceil(4.1)`	`5`
`Math.floor(x)`	يقرب الرقم لأسفل (أرض)	`Math.floor(4.9)`	`4`
`Math.trunc(x)`	يحذف الجزء العشري بدون تقريب	`Math.trunc(4.9)`	`4`
`Math.pow(x, y)`	حساب القوة: x أس y	`Math.pow(2, 3)`	`8`
`Math.sqrt(x)`	الجذر التربيعي للعدد	`Math.sqrt(16)`	`4`
`Math.abs(x)`	القيمة المطلقة (بدون إشارة سالبة)	`Math.abs(-7)`	`7`
`Math.min(...nums)`	يرجع أقل رقم	`Math.min(5, 10, -2)`	`-2`
`Math.max(...nums)`	يرجع أكبر رقم	`Math.max(5, 10, -2)`	`10`

الدالة	شرح مبسط	مثال	الناتج
`Math.random()`	يرجع رقم عشري عشوائي من 0 إلى أقل من 1	`Math.random()`	مثلًا: `0.7261`
`Math.floor(Math.random() * n)`	رقم عشوائي صحيح من 0 إلى أقل من n	`Math.floor(Math.random() * 10)`	من 0 إلى 9

📝 ملاحظات مهمة:

- ال `Math.random()` دايمًا بين 0 (شامل) و 1 (غير شامل).
- عشان تجيب رقم عشوائي بين **1 و 10 مثلًا**:

```
Math.floor(Math.random() * 10) + 1;
```

Problem Solving :

You are building a simple calculator that helps users understand prices.
The user enters a **price** (number with decimals), and you must:

1. Round the price to **2 decimal places**.
2. If the price is **less than 0**, return `"Invalid price"`.
3. Show the **price with tax** (add 15% tax).
4. Return the price with tax:
   - Rounded to 2 decimals
   - Converted to a string
   - And padded at the **start** with `"$"` until the string is 8 characters long
   👉 Return the final result.

Solution:

```
function calculatePrice(price){

    if(price<0){

        return "Invalid price";

    }

    let roundedPrice=Number(price.toFixed(2));

    let priceWithTax=roundedPrice + (roundedPrice * 0.15);
```

```
    priceWithTax=Number(priceWithTax.toFixed(2))

    let result=priceWithTax.toString().padStart(8,'$')

    return result;

}

console.log(calculatePrice(12.987));

console.log(calculatePrice(-4));
```

🟢 `Boolean` (قيمة منطقية)

✅ التعريف:

نوع بيانات له **قيمة واحدة من اثنين فقط**:

- `true` ✅ (صح)
- `false` ❌ (غلط)

🧠 متى نستخدمه؟

لما نسأل سؤال إجابته "نعم أو لا"، مثل:

- هل المستخدم سجل دخول؟
- هل الرقم أكبر من 10؟
- هل كلمة السر صحيحة؟

✅ مثال واقعي:

```
let isLoggedIn = true;  // المستخدم سجل الدخول
let hasLicense = false; // لا يمتلك رخصة قيادة
```

🟡 `undefined` (غير مُعرّف)

✅ التعريف:

قيمة تلقائية لأي **متغير لم يُعطَ له قيمة** بعد.

🧠 متى نستخدمه؟

لو نسيت تعيّن قيمة لمتغير أو نسيت ترجعي قيمة من دالة.

✅ مثال واقعي:

```
let username;
console.log(username); // undefined (ماحطيناش اسم)
```

🔵 `null` (فارغ عن قصد)

✅ التعريف:

بنعطي `null` لما نكون عارفين إن القيمة فاضية **بقصد**، عكس `undefined` اللي معناها "لسّه مفيش قيمة".

🧠 متى نستخدمه؟

لو حابين نوضّح إن مفيش بيانات حاليًا، بس هتيجي بعدين.

✅ مثال واقعي:

```
let selectedProduct = null; // المستخدم لسه ما اختارش منتج
```

✅ اختبار بسيط لك:

```
let isOnline = false;
let age;
let favoriteBook = null;

console.log(isOnline);     // ??
console.log(age);          // ??
console.log(favoriteBook); // ??
```

الاجابة :

- `false` ✅ → لأن `isOnline = false` (قيمة منطقية).
- `undefined` ✅ → لأن `age` ما أخدتش أي قيمة.
- `object` ✅ →
  - لأن `null` نوعه في JavaScript هو `object`
    وده **بُق (bug)** قديم جدًا في JavaScript، من أيامها الأولى، بس اتحفظ كده علشان التوافق مع النسخ القديمة.
    يعني فعليًا `null` مش object، لكن JavaScript بتقوله كده بسبب غلطة قديمة.

✅ أولًا: يعني إيه Object في JavaScript؟

الـ `Object` هو **نوع بيانات (Data Type)** بيسمح لنا نخزن بيانات على شكل **أزواج من مفتاح وقيمة (key: value)**.

كل حاجة في JavaScript تقريبًا ممكن تعتبر Object:

- المتصفح → Object
- التاريخ → Object
- المصفوفات → Objects
- الدوال → كمان Objects

🧠 تخيلي بسيط:

تخيلي إن عندك شنطة فيها كروت، كل كارت مكتوب عليه:

- اسم الحاجة (key)
- وقيمتها (value)

```
let objectName = {
  key1: value1,
  key2: value2,
  ...
};
```

```
let student = {
  name: "Ayaat",
  age: 22,
  isGraduate: true
};
```

✅ إزاي أتعامل مع الـ Object؟

1. قراءة البيانات (Access):

باستخدام dot notation:

```
console.log(student.name); // "Ayaat"
```

أو باستخدام bracket notation:

```
console.log(student["age"]); // 22
```

2. تعديل البيانات (Update):

```
student.age = 23;
student["name"] = "Ayaat Elhalawany";
```

3. إضافة خصائص جديدة (Add):

```
student.city = "Cairo";
```

4. حذف خاصية (Delete):

```
delete student.isGraduate;
```

✅ خصائص مهمة في الـ Object:

الخاصية	معناها
`Object.keys(obj)`	بترجع Array فيها كل المفاتيح
`Object.values(obj)`	بترجع Array فيها القيم
`Object.entries(obj)`	بترجع Array داخلها Arrays بالشكل: `[key, value]`
`hasOwnProperty("key")`	تشوف لو الـ object فيه المفتاح ده

مثال:

```
let car = {
  brand: "Toyota",
  year: 2020
};

console.log(Object.keys(car));     // ["brand", "year"]
console.log(Object.values(car));   // ["Toyota", 2020]
console.log(Object.entries(car));  // [["brand", "Toyota"], ["year", 2020]]
console.log(car.hasOwnProperty("brand")); // true
```

✅ Nested Objects (كائنات متداخلة):

```
let user = {
  name: "Ayaat",
  address: {
    city: "Cairo",
    zip: 12345
  }

};
```

```
user.sayHello = function(){ console.log("Hello!"); };
console.log(user.address.city); // "Cairo"
```

✅ Loop على الـ Object:

لو عايزة تطبعي كل الخصائص والقيم:

```
for (let key in user) {
  console.log(key + ": " + user[key]);
}
```

✨ أهم النقاط عن Object:

السؤال	الإجابة
هل Object نوع بيانات؟	نعم
هل keys يكون String؟	دائمًا، حتى لو كتبته بدون علامات اقتباس
هل values ممكن تكون أنواع مختلفة؟	نعم: string, number, array, object, function
هل object مرتب؟	لأ، مش زي المصفوفة

علشان تعمل تجميد (Freeze) لكائن (object) في JavaScript، بتستخدم الدالة:

```
Object.freeze(object)
```

✅ إزاي بيشتغل `Object.freeze()` ؟

- بيمنع **تعديل القيم** داخل الكائن.
- بيمنع **إضافة خصائص جديدة**.
- بيمنع **حذف خصائص**.
- لكن: لو الكائن فيه خصائص عبارة عن كائنات داخلية (nested objects)، فـ `freeze` مش بيجمدهم إلا لو استخدمت **التجميد العميق** (Deep Freeze).

🧪 مثال بسيط:

```
const person = {
  name: "Ayaat",
  age: 22
};
```

```
Object.freeze(person);

person.name = "Suzan";      // ❌ مش هيتغير
person.city = "Cairo";      // ❌ مش هيتضاف
delete person.age;          // ❌ مش هيتحذف

console.log(person); // { name: "Ayaat", age: 22 }
```

⚠️ ملاحظات مهمة:

- ال `Object.freeze()` بيشتغل **فقط على المستوى الأول** من الكائن.
- لو عندك كائنات متداخلة، زي:

```
const user = {
  name: "Ayaat",
  address: {
    city: "Cairo"
  }
};

Object.freeze(user);
user.address.city = "Alex"; // ✅ ده هيتغير عادي!
```

🧊 الحل؟ Deep Freeze:

```
function deepFreeze(obj) {
  Object.freeze(obj);
  Object.keys(obj).forEach(key => {
    if (typeof obj[key] === 'object' && obj[key] !== null) {
      deepFreeze(obj[key]);
    }
  });
}

deepFreeze(user);
user.address.city = "Alex"; // ❌ مش هيتغير بقى
```

ال optional chaining (`?.`) في JavaScript

هي طريقة بتسهل علينا نقرأ بيانات من كائنات (objects) أو مصفوفات (arrays) **من غير ما الكود يقع** لو جزء من السلسلة مش موجود أو undefined/null.

✅ تعريف بسيط:

ال **optional chaining** = طريقة آمنة للوصول لخصائص الكائنات المتداخلة، حتى لو في مستوى منهم مش موجود.

📌 المشكلة بدون optional chaining:

```
const user = {
  name: "Ayaat",
  address: {
    city: "Cairo"
  }
};

console.log(user.address.city);      // ✅ Cairo
console.log(user.job.title);         // ❌ TypeError: Cannot read property
'title' of undefined
```

لو `job` مش موجود، الكود بيقع.

✅ الحل باستخدام optional chaining:

🧪 أمثلة واقعية:

1. الوصول لبيانات مستخدم:

```
const user = {
  name: "Ayaat",
  profile: {
    email: "aya@example.com"
  }
};

console.log(user?.profile?.email);      // "aya@example.com"
console.log(user?.profile?.phone);      // undefined
console.log(user?.settings?.theme);     // undefined (بدون Error)
```

2. استخدامه مع Arrays:

```
const users = [
  { name: "Ayaat" },
  null,
  { name: "Suzan" }
];
```

```
console.log(users[1]?.name);     // undefined
console.log(users[2]?.name);     // "Suzan"
```

3. مع الـ functions:

```
const user = {
  sayHello: () => "Hello!"
};

console.log(user.sayHello?.());      // "Hello!"
console.log(user.sayBye?.());        // undefined (مش بتكسر الكود)
```

🛑 تحذير بسيط:

ال *optional chaining** بيشتغل مع `null` و `undefined` بس. لو القيمة موجودة لكنها مثلاً string أو false، هيكمل عادي.

✅ Problem: Personal Info Formatter

Create a JavaScript function called `formatPersonInfo` that takes an object with information about a person and returns a formatted sentence.

The object will have the following properties:

- `name` (string)
- `age` (number)
- `city` (string)

The function should return a sentence like:
📌 `"My name is Ahmed, I am 25 years old and I live in Cairo."`

```
let person={

        name:"Ahmed",

        age:25,

        city:"Cairo",

    };

function formatPersonInfo(person){
```

```
    return `My name is ${person.name}, I am ${person.age} years old and I live in ${person.city}. `

}

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

✅ أولاً:

ما معنى function (دالة)؟

الدالة هي **قطعة كود** نكتبها مرة واحدة، ونقدر نستخدمها في أي وقت نحتاجها بدل ما نكرر نفس الكود.

🔷 **مثال واقعي:**
زي ما بتكتب وصفة في كشكول الطبخ، وكل مرة تعوز تعمل الوصفة تروح تبص عليها بدل ما تكتبها من أول وجديد وده مبدأ اسمه Don't repeat yourself(DRY) .

🔷 **مثال برمجي:**

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

sayHello();  // بتنفذ الدالة
```

ما معنى built-in function (دالة مدمجة جاهزة في JavaScript)؟

هي دوال **جاية مع لغة JavaScript نفسها**، مش إحنا اللي كتبناها.

🔷 أمثلة مشهورة:

الدالة	وظيفتها
`alert()`	تعرض رسالة تنبيه للمستخدم
`parseInt()`	تحوّل نص إلى عدد صحيح
`Math.round()`	تقرّب الرقم لأقرب عدد صحيح
`"text".toUpperCase()`	تحوّل النص لحروف كابيتال

🔷 مثال:

```
let num = "5";
let result = parseInt(num);  // built-in function
```

```
console.log(result); // 5
```

ما معنى custom function (دالة مخصصة من كتابتك)؟

هي دالة **إنت اللي بتكتبها بنفسك** عشان تحقق هدف معين، مش موجود في الجاهز.

🔷 مثال:

```
// syntax Function
function _name_(_parameter1, parameter2, parameter3_) {
  // _code to be executed_
  return;
}
// Example
function greet(name) {
  console.log(`Hello, ${name}!`);
}
// call the function
greet("Ayaat"); // Hello, Ayaat!
```

✅ يعني إيه parameters و arguments؟

📌 أولًا:

ما معنى Parameter (باراميتر)؟

هو **اسم متغير** بنكتبه في تعريف الدالة (function) علشان نقول إن الدالة هتحتاج معلومة معينة وقت التشغيل.

> Think of it like an **empty box** 🥣... وإنت مستني تحط فيه حاجة.

📌 ثانيًا:

ما معنى Argument (أرجومنت)؟

هو **القيمة الفعلية** اللي بتحطها في البارامیتر لما تنادي على الدالة.

> يعني هو **المحتوى اللي بتحطه في الصندوق** 🧃.

🟡 مثال واقعي:

تخيل إنك عملت وصفة بتقول:

> "ضيف نوع الفاكهة اللي تحبه في العصارة، وهتديك عصير".

هنا:

- "نوع الفاكهة" = **parameter** (يعني متغير اسمه `fruit`)
- "تفاح" أو "موز" اللي بتحطيه فعلاً = **argument**

🟢 مثال برمجي:

```
// name اسمه parameter تعريف الدالة وفيه
function greet(name) {
  console.log(`Hello, ${name}!`);
}

// "Ayaat" حقيقي اسمه argument هنا بنمرر
greet("Ayaat");  // Hello, Ayaat!
```

💡 ليه بنستخدمهم؟

- علشان نخلي الدالة **مرنة** وتشتغل على **بيانات مختلفة** كل مرة.
- بدل ما نكتب نفس الكود بتكرار، نستخدم دالة واحدة ونمرر لها بيانات كل مرة.

✅ Pure Function (دالة نقية):

هي دالة:

1. **ناتجها بيعتمد فقط على القيم اللي بتدخلها (parameters)**.
2. **لا تغيّر أي شيء خارجها** (يعني مفيش تعديل على متغيرات خارجية أو طباعة أو إرسال بيانات).

> ✅ نفس المدخلات = نفس النتائج دائمًا
> ❌ مفيش Side Effects (زي `alert` , `console.log` , تعديل في data خارجية)

🔷 **مثال على pure function:**

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

🔷 مثال على دالة فيها side effect (مش pure):

```
let count = 0;
function increment() {
  count++;  // side effect: بتغير في متغير خارجي
}
```

🔁 Regular Function

هي أي دالة بنكتبها بالطريقة الكلاسيكية (العادية) باستخدام `function` :

```
function greet(name) {
  return `Hello, ${name}`;
}
```

✅ ممكن تكون pure أو مش pure حسب استخدامها.

🟦 ثانيًا: ما هي Function Expression؟

هي لما تخزن دالة داخل متغير (زي ما بنخزن قيمة رقم أو نص):

```
const greet = function(name) {
  return `Hello, ${name}`;
};
```

هنا الدالة ملهاش اسم (anonymous function)، واتخزنت في متغير اسمه `greet` .

⚡ ثالثًا: ما هي Arrow Function؟

هي شكل مختصر وحديث من تعريف الدوال (تمت إضافتها في ES6):

```
const greet = (name) => {
  return `Hello, ${name}`;
};
```

❓ليه نستخدمها؟

- أقصر في الكتابة
- ما عندهاش `this` خاص بيها (مهم جدًا في الـ OOP)

📃 رابعًا: Explicit Return vs. Implicit Return

✅ Explicit Return (الرجوع الصريح):

يعني لازم تكتب `return` بشكل واضح:

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

✅ Implicit Return (الرجوع الضمني):

لو الدالة سطر واحد، تقدر **تشيل** `{}` و `return` :

```
const add = (a, b) => a + b;
```

✅ ال JavaScript هتفهم تلقائيًا إنك عايز ترجع القيمة.

🌟 1. IIFE = Immediately Invoked Function Expression

يعني **دالة بتشتغل لحالها أول ما تتكتب**.

🔶 الشكل:

```
(function() {
  const name = "Suzan";
  console.log(`Hello ${name}!`);
})();  // Hello Suzan!
```

🔶 ليه نستخدمها؟
عشان ننفّذ كود **مرة واحدة** من غير ما نضطر ننده عليه تاني، وكمان بنحافظ على المتغيرات **خاصة ومخفية** (مش global).

🛠️ 2. Method

الميثود هي **دالة داخل كائن (object)**.
🔶 الشكل:

```
const person = {
  name: "Ayaat",
  greet: function() {
    console.log(`Hi, I'm ${this.name}`);
  }
};

person.greet(); // Hi, I'm Ayaat
```

🔶 ليه نستخدم الميثودز؟
لأنها **سلوك بيخص الكائن** (زي إن الشخص يتكلم أو السيارة تمشي).

🔶 مثال واقعي:
الكائن زي بني آدم، له:

- **خصائص (properties)**: الاسم، السن
- **سلوك (methods)**: يتكلم، يتحرك، ينام

🔁 3. Callback Function

يعني دالة **بنبعتها كباراميتر** لدالة تانية، علشان **تتنفذ بعد وقت معين أو بعد حدث معين**.

🔶 الشكل:

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

function sayBye() {
  console.log("Bye!");
}

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

🔶 مثال واقعي:
زي لما تقولي لصاحبتك:

> "هاتلي العصير، **ولما تخلصي**، ناديني"

- "هاتلي العصير" = الفنكشن الأساسية
- "ناديني" = الكولباك

🔍 Debugging Tools (أدوات تصحيح الأخطاء)

1. Console Methods (دوال الكونسول)

تستخدم لعرض البيانات أو التحذيرات أو الأخطاء في تبويب Console بالمتصفح.

Method	الاستخدام
`console.log()`	لطباعة معلومات عادية (للمتابعة)
`console.warn()`	لطباعة تحذير باللون الأصفر
`console.error()`	لطباعة خطأ باللون الأحمر
`console.table()`	عرض Array أو Object في شكل جدول

مثال:

```
console.log("تم تنفيذ الكود");
console.warn("تحذير: تحقق من القيم");
console.error("خطأ: شيء ما حدث!");
```

```
console.table([{name: "Ayaat", age: 22}, {name: "Suzan", age: 26}]);
```

2. Call Stack / Stack Trace (تتبّع المكالمات)

هو المكان اللي بيحتفظ بالوظائف اللي تم استدعاؤها بالترتيب.
لو حصل خطأ، يظهر في الكونسول ما يُسمى بـ "Stack trace" علشان تعرفي الخطأ حصل في أي دالة وملف.

🔷 **مثال توضيحي:**

```
function first() {
  second();
}
function second() {
  third();
}
function third() {
  console.log(undeclaredVar); // هتعمل error
}
first();
```

في الكونسول:
هتظهر Stack Trace توضح ترتيب الدوال من `first` إلى `third` ، وتقولك الخطأ في أي سطر.

3. DevTools (نصائح واستراتيجيات)

فتح أدوات المطورين:

- اضغط: `F12` أو `Ctrl + Shift + I`
- اختار تبويب **Console**, **Sources**, **Network**, إلخ.

✅ نصائح:

- في تبويب **Elements**: تابع تغييرات الـ HTML و CSS.
- في **Console**: تابع قيمة المتغيرات.
- في **Sources**: تقدر توقف تنفيذ الكود وتراقب خطوة بخطوة.

4. Breakpoints (نقاط التوقف)

نستخدمها في تبويب **Sources** لإيقاف تنفيذ الكود في سطر معين علشان نشوف القيم أثناء التنفيذ.

🔶 ازاي؟

1. افتح `Sources`
2. افتح الملف.

3. اضغط على رقم السطر بجانب الكود.
4. لما توصل للسطر ده، الكود هيتوقف.
5. هتشوف المتغيرات في الجانب الأيمن.

5. Network Requests (مراقبة الشبكة)

في تبويب **Network** تراقب:

- هل الـ API اشتغلت؟
- هل الـ request رجع Response؟
- وقت التنفيذ.
- أي Errors في الـ request.

مثال: لما تعمل fetch:

```
fetch("https://api.example.com/data")
  .then(res => res.json())
  .then(data => console.log(data));
```

تابع في Network إذا نجحت أو فشلت.

6. Break on (كسر عند التغيير)

🛠️ Break on > شرح كل اختيار:

1. Attributes Modifications

📌 **يعني:** لو اتغير أي *Attribute* (خاصية) للعنصر ده زي:

- `class`
- `id`
- `src`
- `href`
- `disabled`
- `checked`

🔍 **المتصفح هيقف (Break)** عند السطر اللي حصل فيه التغيير، وتقدر تشوف مين غيره.

✅ **مثال عملي:**

```
document.getElementById("btn").setAttribute("disabled", true);
```

لو كنت مشغّل `Break on > Attributes` , الكود هيقف هنا وتقدر تتابع.

2. Subtree Modifications

📌 **يعني:** لو اتغير المحتوى الداخلي للعنصر أو اتضاف/اتشال عناصر داخلية (أبناء – children).

🔍 المتصفح هيوقفك عند أول لحظة يحصل فيها تعديل في الـ innerHTML أو DOM الداخلي.

✅ **مثال عملي:**

```
const newDiv = document.createElement("div");
document.getElementById("box").appendChild(newDiv);
```

لو فعّلت `Subtree Modifications` على العنصر `box` , الكود هيقف.

3. Node Removal

📌 **يعني:** لو العنصر نفسه *اتشال من الصفحة* (اتمسح من الـ DOM).
🔍 المتصفح هيقفك عند اللحظة اللي فيها الكود حاول يشيله.

✅ **مثال عملي:**

```
document.getElementById("msg").remove();
```

لو فعلت `Break on > Node removal` على العنصر `msg` , المتصفح هيوقفك أول ما يوصل للسطر ده.

🔷 مفيدة جدًا لما الصفحة تتغير فجأة ومش عارف ليه.

✅ مثال واقعي Debugging:

```
function greetUser(name) {
    if (!name) {
        console.error("No name provided");
        return;
    }
    console.log(`Welcome, ${name}`);
}
greetUser(); // هتطبع Error في الكونسول
```

🎯 ازاي أستخدمها خطوة بخطوة:

1. افتح الصفحة.
2. افتح DevTools → Elements.
3. اختار العنصر اللي عايز تراقبه.
4. كليك يمين > Break on > اختار النوع.
5. شغّل الصفحة أو التفاعل اللي بيحصل.
6. المتصفح هيقف في لحظة التغيير.
7. روح لتبويب `Sources` → وشوف السطر المسؤول.

كلمة `debugger` في JavaScript من أهم أدوات المبرمجين لما بيكونوا بيحللوا مشاكل (Debugging) في الكود.

✅ يعني إيه `debugger` ؟

ال `debugger` = **كلمة مفتاحية (Keyword)** بتستخدم لإيقاف تنفيذ الكود مؤقتًا عند سطر معين عشان تقدر تشوف:

- المتغيرات فيها إيه
- ترتيب التنفيذ ماشي إزاي
- هل الشرط اشتغل ولا لأ
- هل القيم اللي المفروض تتحسب اتحسبت صح ولا لأ

🧠 طيب إزاي تشتغل؟

1. بتحط السطر `debugger;` في أي مكان من الكود.
2. بتفتح أدوات المطور (DevTools) في المتصفح.
3. لما الكود يوصل للسطر ده، **هيتوقف التنفيذ مؤقتًا**.
4. تقدر تتصفح القيم، تمشي خطوة خطوة (`Step over`) أو تشوف الـ Call Stack.

📌 مثال عملي:

```
function calculateTotal(price, tax) {
  let total = price + (price * tax);
  debugger;  // الكود هيوقف هنا
  return total;
}

let result = calculateTotal(100, 0.15);
console.log(result);
```

لما تشغل الكود وتفتح DevTools (بالضغط على F12 أو كليك يمين > Inspect > تبويب Sources)،
الكود **هيتوقف عند** `debugger;` ، وتقدر تشوف:

- قيمة `price`
- قيمة `tax`
- النتيجة اللي اتحسبت في `total`

🔒 ملاحظات مهمة:

- `debugger` **مش بيشتغل** إلا لو أدوات المطور مفتوحة.
- لازم **تشيله من الكود قبل ما ترفعيه على الإنترنت** (production)، لأنه ممكن يوقف البرنامج فجأة للمستخدم.

✅ استخداماته:

- تتبع مشكلة مش واضحة.
- تفهم الكود خطوة خطوة.
- تشوف هل الدالة دي اتنادت؟ وإمتى؟
- تحل bugs صعب تلاقي سببها من مجرد قراءة الكود.

✅ يعني إيه Scope؟

Scope = المجال اللي المتغير متاح فيه

يعني فين أقدر أشوف أو أستخدم المتغير ده؟ في كل الكود؟ ولا جوه دالة بس؟ ولا جوه بلوك صغير؟

💡 مثال واقعي:

تخيل إنك في **بيتكم**، البيت فيه:

- 🛋️ **غرفة المعيشة** (global scope)
- 🍳 **المطبخ** (function scope)
- 🗄️ **درج جوه الدولاب** (block scope)

كل مكان ليه حاجات محددة تقدر توصل لها:

- في غرفة المعيشة تقدر توصل للتلفزيون.
- في المطبخ بس تقدر توصل للثلاجة.
- في الدرج بس فيه مفاتيح سريّة، مش في باقي البيت.

1️⃣ Global Scope – المجال العام

أي متغير يتكتب برا كل الدوال أو البلوكات، بيكون متاح في كل مكان.

```
let familyName = "Elhalawany";

function greet() {
  console.log("Hello " + familyName); // ممكن أوصل له
}

greet();  // Hello Elhalawany
```

🎯 **زي التلفزيون اللي في غرفة المعيشة.. الكل يشوفه.**

2️⃣ Function Scope – داخل الدالة

المتغيرات اللي جوه دالة متاحة بس جوه نفس الدالة.

```
function sayName() {
  let name = "Ayaat";
```

```
  console.log("Hi " + name);
}

sayName();          // Hi Ayaat
console.log(name);  // ❌ Error: name is not defined
```

🧠 **ليه؟** لأن `name` موجودة بس داخل مطبخ (الدالة)، براها مش باينة.

3️⃣ Scope Lookup – البحث عن المتغير

لو المتغير مش موجود في المكان الحالي، بيبدأ يشوف في اللي برا.

```
let city = "Cairo";

function showCity() {
  let msg = "I live in";
  console.log(msg + " " + city);
}

showCity();  // I live in Cairo
```

🔎 الدالة ما لاقتش `city` جواها، فدورت في الخارج (global) ولاقته.

4️⃣ Block Scope – مجال البلوك (داخل if / for / {})

الكود اللي داخل `{}` زي في if أو for أو حتى block عادي، المتغيرات اللي جواه بتكون محصورة فيه بس.

```
if (true) {
  let mood = "Happy";
  console.log(mood); // Happy
}

console.log(mood); // ❌ Error: mood is not defined
```

📦 **ال `mood` جوه صندوق، برا الصندوق مش شايفها.**

🔁 مثال شامل واقعي:

```
let school = "Azhari";

function studentInfo() {
  let name = "Ayaat";
  if (true) {
```

```
    let age = 22;
    console.log(name, age, school); // متاحة كلها
  }
  console.log(age); // ❌ age مش متاحة هنا
}

studentInfo();
console.log(name); // ❌ name مش متاحة
```

✅ Hoisting

هوستنج = رفع المتغيرات أو الدوال لأعلى النطاق (scope) قبل تنفيذ الكود.

📌 الفكرة ببساطة:

لما تكتب كود في JavaScript، **قبل ما يتنفذ، المتصفح بيجهز كل التعريفات** (variables و functions) عن طريق ما يُعرف بالـ **Hoisting**.

لكن:

- مش كل حاجة بتترفع بنفس الطريقة.
- ولا كل حاجة ينفع تستخدمها قبل تعريفها.

🧠 نبدأ بـ `var`

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

✅ ليه الكود ده مش بيعمل Error؟

لأن المتصفح بيعمل hoisting كده:

```
var x;        // رفعت لأعلى السكوب وتعريفها حصل كـ undefined
console.log(x); // undefined
x = 5;
console.log(x); // 5
```

- ❗**لكن القيمة ما بترتفعش**، بس **التعريف بيرتفع**.

❌ `let` و `const`:

```
console.log(y); // ❌ Error: Cannot access 'y' before initialization
let y = 10;
```

- **المتغير مرفوع (hoisted)** لكن بيكون في حاجة اسمها `Temporal Dead Zone` ، يعني مش مسموح تستخدمه قبل تعريفه.

✅ Hoisting مع الدوال (Functions):

```
sayHello();

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

✅ ينفع تنادي على الدالة قبل تعريفها، **لو مكتوبة بصيغة Declaration**.

```
sayHi(); // ❌ TypeError

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

هنا الدالة عبارة عن **Expression** جوه متغير، فبتتعامل زيه، ولو `var` هتكون undefined.
لأن أي **Function Expression** (سواء `var` , `let` , `const`) بيتعامل كـ **متغير** وهنا ال sayHi مجرد متغير و في دالة.
نفس الكلام مع نوع Arrow Function مينفعش اعمل call للدالة قبل تعريفها .

📌 مثال توضيحي لكل الأنواع:

```
// ✅ Function Declaration
hello(); // OK

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

// ❌ Function Expression
hi(); // Error

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

```
// ❌ Arrow Function
hey(); // Error

const hey = () => {
  console.log("Hey");
};
```

✅ ما هو الـ Closure؟

Closure يعني:

> إن الدالة الداخلية تحتفظ بالوصول للمتغيرات اللي اتعرفت في الدالة الخارجية حتى بعد ما الدالة الخارجية خلصت تنفيذها.

🔍 ازاي كده؟ خلينا نشوف خطوة خطوة:

```
function outer() {
  let name = "Ayaat"; // دالة داخلية هتستخدمه

  function inner() {
    console.log(`Hello, ${name}`);
  }

  return inner;
}

const greet = outer(); // outer خلصت، لكن inner لسه موجودة
greet(); // Hello, Ayaat ✅
```

✨ اللي حصل هنا:

- ال `outer` بتنشئ متغير اسمه `name` .
- ال `inner` هي دالة جوه `outer` وبتستخدم `name` .
- رجّعنا `inner` برا.
- حتى بعد انتهاء `outer` ، لسه `inner` قادرة توصل للمتغير `name` .

وده هو معنى الـ **Closure**.

📦 تعريف بسيط:

> ال **Closure** هو "دالة تحتفظ بمرجع للبيئة اللي اتعرفت فيها حتى بعد ما البيئة دي اختفت".

💡 مثال واقعي (بسيط وسهل):

تخيل إنك قاعد في **كافيه**، وجالك **جرسون**:

- قولتله : "أنا عايز تشيز كيك بس من غير سكر".
- الجرسون خد المعلومة ومشي.
- بعد 10 دقايق رجعلك بالطلب، ومراعي اللي قلتيه (من غير سكر).

في البرمجة:

- الدالة الخارجية هي زي المحادثة اللي حصلت.
- الجرسون هو الدالة الداخلية.
- الـ closure هو إن الجرسون فضل فاكر كلامك حتى بعد ما المحادثة خلصت.

🧑‍💻 مثال عملي واقعي في الكود:

```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    return count;
  };
}

const counter = createCounter();

console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

💡 إيه اللي بيحصل هنا؟

- ال `createCounter` بترجع دالة تقدر تزود `count` .
- رغم إن `createCounter` خلصت، لكن `counter()` كل مرة بيفضل يفتكر القيمة القديمة ويكمل عليها.
- ده مثال عملي على الـ **closure**.

💬 استخدامات الـ Closure في الحياة الواقعية:

1. ✅ إنشاء **مؤقتات** و **عدادات**.
2. ✅ عمل **تغليف للبيانات** (Data Encapsulation).
3. ✅ في الـ **modules**.
4. ✅ لمنع التلاعب بالمتغيرات.

⚠️ إزاي ممكن ييجي في الـ Interview؟

مثال سؤال:

> What is a closure? Can you give an example?

أو:

> How can you create a private variable in JavaScript?

✅ إجابة مختصرة للمقابلة:

> A **closure** is a function that has access to its outer function's variables even after the outer function has returned.
> It helps in creating **private variables** and managing state in a controlled way.

✅ ما هو الـ DOM؟

ال**DOM** هو اختصار لـ **Document Object Model**، ومعناه "نموذج كائني لوثيقة HTML".

يعني لما تفتح صفحة HTML، المتصفح بيحولها من مجرد كود HTML إلى **أشجار من الكائنات (objects)** يقدر جافاسكربت تتعامل معاها.
كل عنصر في الصفحة (زر، صورة، فقرة، إلخ) بيتحول إلى **كائن (Object)** تقدر توصل ليه وتعدل عليه بجافاسكربت.

> 📌 مثال واقعي:
> تخيل إن صفحة HTML زي شجرة فيها فروع وأوراق.
> الفروع = العناصر الكبيرة (div, section)
> الأوراق = العناصر الصغيرة (button, img, p).
> كل عنصر في الشجرة ممكن توصل له وتغير خصائصه من خلال جافاسكربت.

ال DOM مقسم ل :

1. Element : html tag without text .
2. Node : element and text .

✅ كيف نصل لعناصر الـ DOM؟

🔷 1. Selecting Elements – اختيار العناصر

جافاسكربت بتوفرلك طرق كتير عشان توصل لأي عنصر في الصفحة:

الطريقة	الوصف	مثال
`getElementById`	يجيب عنصر حسب الـ ID	`document.getElementById("myId")`
`getElementsByClassName`	يجيب عناصر ليها نفس الكلاس	`document.getElementsByClassName("box")`

الطريقة	الوصف	مثال
`getElementsByTagName`	يجيب كل العناصر من نوع معين	`document.getElementsByTagName("p")`
`querySelector`	يجيب أول عنصر يطابق الـ CSS Selector	`document.querySelector(".box")`
`querySelectorAll`	يجيب كل العناصر اللي تطابق CSS Selector	`document.querySelectorAll(".box")`

🎯 مثال واقعي:

```
<p id="title">Hello</p>
<script>
  let title = document.getElementById("title");
  console.log(title.textContent); // Hello
</script>
```

✅ خصائص ومهام العناصر (Element Properties and Methods)

أشهر الخصائص:

الخاصية	الوصف
`innerHTML`	محتوى العنصر كـ HTML
`textContent`	النص داخل العنصر بدون أكواد HTML
`value`	قيمة العناصر زي input و textarea
`src` , `href`	مصدر الصور والروابط
`style`	التعديل على CSS مباشرة
`id` , `className`	قراءة/تعديل الكلاس أو الـ ID

🎯 مثال عملي:

```
<input id="name" value="Ayaat" />
<script>
  let input = document.getElementById("name");
  console.log(input.value); // Ayaat
  input.value = "Suzan"; // تغيير القيمة
</script>
```

🔷 `innerText` vs `textContent`

✅ `textContent`:

- بيرجع كل النص داخل العنصر، سواء ظاهر للمستخدم أو مخفي بالـ CSS.
- أسرع وأدق.

```
<div id="myDiv" style="display:none">Hello</div>
<script>
  console.log(document.getElementById("myDiv").textContent); // Hello
</script>
```

✅ `innerText`:

- بيرجع **النص الظاهر فقط** للمستخدم (اللي بيتعرض فعليًا في الصفحة).
- أبطأ لأنه بيراعي التنسيق والستايل (display, visibility).

```
<div id="myDiv" style="display:none">Hello</div>
<script>
  console.log(document.getElementById("myDiv").innerText); // (Nothing - لأنه مخفي)
</script>
```

🔷 `insertAdjacentText()`

✅ تستخدم لإضافة نص في مكان معين حول العنصر الحالي.

```
element.insertAdjacentText(position, text);
```

القيمة	مكان الإضافة
`"beforebegin"`	قبل العنصر تمامًا
`"afterbegin"`	داخل العنصر، قبل أي محتوى داخلي
`"beforeend"`	داخل العنصر، بعد كل المحتوى الداخلي
`"afterend"`	بعد العنصر تمامًا

🎯 مثال عملي:

```
<div id="box">Hi</div>
```

```
<script>
  const box = document.getElementById("box");
  box.insertAdjacentText("afterbegin", "Start-");
  box.insertAdjacentText("beforeend", "-End");
  // الناتج: <div id="box">Start-Hi-End</div>
</script>
```

✅ رابعًا: التعامل مع الـ Classes

استخدام `classList` :

`classList` بيديك طرق سريعة للتعامل مع الكلاسات:

- `add("class")` – يضيف كلاس
- `remove("class")` – يشيل كلاس
- `toggle("class")` – يضيف لو مش موجود، ويشيل لو موجود
- `contains("class")` – لو الكلاس موجود بيرجع true

> 🎯 مثال عملي:

```
let box = document.querySelector(".box");
box.classList.add("active");
box.classList.remove("hidden");
box.classList.toggle("dark-mode");
```

✅ خامسا : يوجد نوعين من ال attributes :

🔷 أولًا: ما هي Built-in Attributes في DOM؟

المقصود بـ Built-in Attributes:

هي **السمات الجاهزة** اللي بتكون موجودة في عناصر HTML، وجافاسكربت بتقدر توصل ليها من خلال العنصر نفسه.

يعني أي عنصر في HTML ممكن يكون ليه:

- خصائص (Properties)
- سمات (Attributes)

وجافاسكربت تقدر تتعامل مع الاتنين.

✅ الفرق بين `attribute` و `property` :

Property (الخاصية)	Attribute (السمة)
بتكون متاحة ككائن في جافاسكربت	بتكتب في كود HTML

Attribute (السمة)	Property (الخاصية)
`disabled` , `class` , `id` , `href` , `value`	`element.disabled` , `element.className` , `element.id` , `element.href`
تستخدم `getAttribute()` و `setAttribute()` للتعامل معها	تستخدم مباشرة عبر dot syntax (`.`)

```
<input type="text" id="username" value="Ayaat" />

let input = document.getElementById("username");

// Property
console.log(input.value); // Ayaat

// Attribute
console.log(input.getAttribute("value")); // Ayaat
```

✅ الفرق هنا إن:

- `value` كـ **property** ممكن تتغير ديناميكيًا.
- `getAttribute("value")` بيرجع **القيمة الأصلية** اللي اتكتبت في الـ HTML.

✅ ثانيا: Data Attributes (البيانات المخصصة)

دي خصائص مخصصة بتبدأ بـ `-data` بتضيفها على العناصر وتقدر توصل ليها بجافاسكربت.

> 🎯 مثال:

```
<button data-user-id="1234">Click Me</button>

<script>
  let btn = document.querySelector("button");
  let userId = btn.dataset.userId;
  console.log(userId); // 1234
</script>
```

✅ سادسًا: إنشاء عناصر HTML بجافاسكربت (Creating HTML Elements)

1. إنشاء عنصر:

```
let newElement = document.createElement("p");
newElement.classList.add("greeting");
```

```
newElement.dataset.name='greetin';
newElement.textContent = "Hello Ayaat";
```

2. إضافته داخل عنصر موجود:

```
document.body.appendChild(newElement);
// or
newElement.appendChild(img)
```

3. استخدام innerHTML:

```
document.body.innerHTML += "<p>Hello Ayaat</p>";
```

⚠️ لكن استخدام `innerHTML` ممكن يسبب مشاكل أمنية زي:

سابعا : HTML From Strings

✅ 1. `insertAdjacentHTML`

🎯 تعريف:

دي طريقة بتسمحلك تضيف **HTML string** في مكان محدد بالنسبة لعنصر موجود، من غير ما تمسح المحتوى القديم.

📌 الصيغة:

```
element.insertAdjacentHTML(position, htmlString);
```

🟢 القيم الممكنة لـ `position` :

القيمة	المعنى
`"beforebegin"`	قبل العنصر نفسه مباشرةً
`"afterbegin"`	داخل العنصر، قبل أول عنصر فرعي
`"beforeend"`	داخل العنصر، بعد آخر عنصر فرعي
`"afterend"`	بعد العنصر نفسه مباشرةً

🔍 مثال:

```
let box = document.getElementById("box");
let html = `<p>Hello Ayaat 🌸</p>`;
```

```
box.insertAdjacentHTML("beforeend", html);
```

✅ ده هيضيف الـ `<p>` جوا العنصر `box#` من غير ما يمسح محتواه الأصلي.

✅ 2. `document.createRange().createContextualFragment()`

🎯 تعريف:

دي طريقة قوية وآمنة لإنشاء شجرة DOM من نص HTML. بتكون مفيدة جدًا في المواقف اللي عايز تتجنب فيها XSS أو محتاجة أداء أعلى.
بتُستخدم لتحويل النص (String) إلى عناصر HTML حقيقية (DOM Elements) قبل إضافتها للصفحة، وده بيكون:

🟢 بديل آمن وقوي عن `innerHTML`

💡 ليه نستخدمه؟

لأنه:

- ✅ بيحوّل النص إلى عناصر DOM مباشرة.
- ✅ **أكثر أمانًا** من `innerHTML` (ما بيشغلش السكربتات الغريبة أو الضارة بسهولة).
- ✅ **أكثر كفاءة** لما بنضيف عناصر كثيرة أو مركبة.
- ✅ بيقلل عمليات إعادة الرسم في DOM (reflows/repaints).

```
const htmlString = `<li>Hello Ayaat</li><li>Welcome back!</li>`;

// 1. أنشئ Range
const range = document.createRange();

// 2. حول الـ string إلى fragment (شجرة DOM)
const fragment = range.createContextualFragment(htmlString);

// 3. أضف الـ fragment للعناصر الحقيقية
document.querySelector("ul").appendChild(fragment);
```

✅ الـ `fragment` هنا هو عنصر **خفيف الوزن (lightweight)** بيمثّل مجموعة من عناصر DOM، وبيتم إضافته مرة واحدة، فالأداء أعلى ومفيش إعادة رسم DOM كتير.

🤔 طيب إزاي بيحول الـ string لعنصر؟

اللي بيحصل تحت الغطا هو إن `createContextualFragment` بيقرأ HTML string ويفهم البنية ويحولها لكائنات DOM حقيقية (مش مجرد نص)، زي `<li>` أو `<div>` ... إلخ.

⛔ ليه ما نستخدمش `innerHTML` ؟

```
element.innerHTML = htmlString;
```

- ❌ بيمسح كل المحتوى القديم داخل العنصر.
- ❌ بيعيد تحليل كل HTML داخل العنصر.
- ❌ ممكن يشغّل سكربتات أو يسمح بحقن برمجي (XSS).

✅ نصيحة:

لو المشروع صغير أو مؤقت، `insertAdjacentHTML` تكفي.
لكن لو المشروع كبير، وفيه إدخالات من المستخدم أو بيانات API، استخدم `createContextualFragment` + `appendChild`.

✅ ثامنا: XSS - Cross Site Scripting (هجوم البرمجة الضارة)

لو استخدمت `innerHTML` لإضافة محتوى جاي من المستخدم، ممكن هاكر يكتب كود ضار يتنفذ على موقعك.

> ❌ مثال خطير:

```
let comment = "<img src='x' onerror='alert(\"Hacked\")' />";
document.body.innerHTML += comment;
```

> ✅ الحل:

- استخدم `textContent` أو `()createElement` بدلًا من `innerHTML` لو هتستقبل بيانات من المستخدم.

✅ ليه `createElement` و `appendChild` أفضل من ناحية الاداء ؟

1. بيتعاملوا مع DOM ككائنات مباشرة (Node)

يعني بيضيفوا عنصر جديد بدل ما يعيدوا رسم الصفحة كلها زي `innerHTML`.

```
// هذا أسرع:
const div = document.createElement("div");
div.textContent = "Hello";
document.body.appendChild(div);
```

2. ما بيعملوش إعادة تحليل (Reparse) لكل الـ HTML الموجود

لكن:

```
// هذا أبطأ وممكن يعمل مشاكل:
document.body.innerHTML += "<div>Hello</div>";
```

ده بيمسح ويعيد كل عناصر الصفحة = **أداء سيء كل ما زاد الحجم**.

❗لكن في نقطة مهمة:

لو كل مرة هتعمل `appendChild` مباشرة، فده معناه إنك بتعمل **أكثر من Access على الـ DOM**
وده = مكلف.

✅ الحل الاحترافي: استخدام DocumentFragment

🎯 ال `DocumentFragment` عبارة عن حاوية مؤقتة في الذاكرة، تجهز فيها العناصر كلها مرة واحدة، وبعدين تضيفهم مرة واحدة للـ DOM.

🧠 مثال:

```
const ul = document.createElement("ul");
const fragment = document.createDocumentFragment();

for(let i = 0; i < 1000; i++){
  const li = document.createElement("li");
  li.textContent = `Item ${i}`;
  fragment.appendChild(li);
}

ul.appendChild(fragment);
document.body.appendChild(ul);
```

✅ كده أنت لمسْت الـ DOM مرتين بس:

1. لما أضفت `fragment` إلى `ul`
2. لما أضفت `ul` إلى `body`

📌 بدل ما تلمسه 1000 مرة لو عملت `appendChild` جوه اللوب على طول!

🧭 Traversing the DOM

يعني "التنقل بين عناصر الصفحة" باستخدام خصائص DOM للوصول للعناصر القريبة من بعضها مثل:

✅ 1. `parentNode` أو `parentElement`

ترجع العنصر الأب (اللي يحتوي العنصر الحالي).

📋 جدول توضيحي لخصائص Traversing the DOM:

الخاصية	الوصف	نوع القيمة المرجعة	مثال واقعي
`children`	ترجع **الأبناء (العناصر فقط)** بدون النصوص أو المسافات	HTMLCollection	الوصول لكل `<li>` داخل `<ul>`
`childNodes`	ترجع **كل العقد** (عناصر + نصوص + تعليقات)	NodeList	بتشمل المسافات بين العناصر
`firstElementChild`	أول عنصر (وليس نص أو تعليق)	عنصر (Element)	أول عنصر `<li>` داخل `<ul>`
`lastElementChild`	آخر عنصر (وليس نص أو تعليق)	عنصر (Element)	آخر `<div>` داخل `<section>`
`firstChild`	أول عقدة (قد تكون نص)	عقدة (Node)	ممكن ترجع نص لو في مسافة
`lastChild`	آخر عقدة (قد تكون نص)	عقدة (Node)	ممكن ترجع تعليق أو نص
`parentElement`	ترجع العنصر الأب المباشر	Element	من الزر نرجع `div` الحاوي ليه
`parentNode`	ترجع الأب سواء عنصر أو عقدة	Node	زي `parentElement` لكن أوسع
`nextElementSibling`	العنصر التالي (نفس المستوى)	Element	من عنصر `<li>` نروح للي بعده
`previousElementSibling`	العنصر السابق (نفس المستوى)	Element	من عنصر نروح للي قبله
`nextSibling`	العقدة التالية (قد تكون نصًا)	Node	ممكن ترجع مسافة
`previousSibling`	العقدة السابقة (قد تكون نصًا)	Node	ممكن ترجع تعليق

🎯 ملاحظات مهمة:

- ال **`Element`** ⟵ عنصر HTML زي `< div>`, `<p>` , `<ul >`
- ال `Node` ⟵ أي شيء في شجرة الـ DOM (عنصر، نص، تعليق...)
- ال `children` و `childNodes` عبارة عن *قوائم* وليست Arrays عادية، لكن تقدرِ تحوليهم بـ:

```
Array.from(element.children)
```

💡 مثال تطبيقي صغير:

```
<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
</ul>
```

```
const secondLi = document.querySelectorAll('li')[1];
console.log(secondLi.previousElementSibling.textContent); // "One"
console.log(secondLi.nextElementSibling.textContent);     // "Three"
console.log(secondLi.parentElement.tagName);              // "UL"
```

JavaScript Intermediate.

🔷 يعني إيه Event Listener؟

هو طريقة **للاستماع لحدث معين** بيحصل على عنصر في الصفحة (زي زرار أو input)، ولما يحصل الحدث ده، **تشغّل دالة معينة** (callback function).

📌 الشكل العام:

```
element.addEventListener("event", function);
```

مثال:

```
const button = document.querySelector("button");
button.addEventListener("click", () => {
  alert("تم الضغط على الزر!");
});
```

🔷 ثانياً: جدول بأشهر أنواع الأحداث (Events Table)

الحدث (Event)	بيحصل إمتى؟	مثال واقعي
`click`	لما المستخدم يضغط بالماوس على عنصر	ضغط على زرار "أرسل"
`dblclick`	لما المستخدم يعمل double click	فتح الصورة بتكبير عند الضغط مرتين
`mouseover`	لما الماوس يدخل على عنصر معين	إظهار قائمة عند المرور على زر
`mouseout`	لما الماوس يخرج من عنصر	إخفاء القائمة بعد الخروج
`mousemove`	عند تحريك الماوس داخل العنصر	تتبع الماوس في لعبة
`mousedown`	عند الضغط على الزر لكن قبل الإفلات	الضغط المطوّل
`mouseup`	عند الإفلات بعد الضغط	إنهاء السحب والإفلات
`keydown`	عند الضغط على زر في الكيبورد	كتابة نص في input
`keyup`	عند الإفلات من زر في الكيبورد	إنهاء إدخال كلمة المرور
`keypress`	عند الضغط على حرف (قديمة وغير مفضلة الآن)	استبدل بـ `keydown` أو `keyup`
`input`	عند إدخال أي شيء في input أو textarea	تحديث قيمة حية في صفحة تسجيل

الحدث (Event)	بيحصل إمتى؟	مثال واقعي
`change`	لما المستخدم يغيّر قيمة input وبعد يخرج	اختيار قيمة من dropdown
`submit`	لما الفورم يتم إرساله	إرسال بيانات تسجيل الدخول
`focus`	لما عنصر مثل input يحصل عليه تركيز	المستخدم ضغط داخل input
`blur`	لما العنصر يفقد التركيز	خرج من input بدون كتابة
`scroll`	عند تمرير صفحة أو عنصر	إظهار زر "ارجع للأعلى"
`resize`	عند تغيير حجم النافذة	إعادة ترتيب عناصر الصفحة تلقائيًا
`load`	لما الصفحة أو صورة يتم تحميلها	تشغيل أنيميشن بعد التحميل
`unload`	لما المستخدم يخرج من الصفحة (قديم)	حفظ حالة المستخدم
`contextmenu`	لما تضغط كليك يمين	تخصيص القائمة اللي تظهر بدل القائمة الافتراضية

🔷 ثالثاً: ملاحظات مهمة:

1. **يمكنك إضافة أكثر من EventListener** لنفس العنصر ولنفس الحدث.

```
button.addEventListener("click", sayHello);
button.addEventListener("click", sayWelcome);
```

2. **عشان تشيل event listener:**
لازم يكون عندك reference للدالة:

```
function greet() {
  alert("Hello");
}
button.addEventListener("click", greet);
button.removeEventListener("click", greet);
```

3. **تقدر تمرر event object** وتستخدم معلومات زي الزر اللي ضغط، الكيبورد، وغيرها:

```
input.addEventListener("keydown", function(e) {
  console.log(e.key); // يظهر المفتاح اللي تم الضغط عليه
});
```

الفرق بين `onclick` و `addEventListener` :

- ال `onclick` بيشيل أي دالة كانت قبله.
- ال `addEventListener` يسمح بوجود أكتر من listener لنفس الحدث.

🟣 أولاً: `event.target` vs `event.currentTarget`

✅ `event.target`

هو **العنصر الفعلي** اللي حصل عليه الحدث (اللي المستخدم ضغط عليه أو تعامل معه مباشرة).

✅ `event.currentTarget`

هو **العنصر اللي عليه الـ event listener نفسه** (سواء الحدث حصل عليه أو على عنصر داخله).

📌 مثال واقعي:

```
// html
<div class="parent">
  <button class="child">Click me</button>
</div

// JS
const parent = document.querySelector('.parent');

parent.addEventListener('click', function(event) {
  console.log("target:", event.target);
  console.log("currentTarget:", event.currentTarget);
});
```

إذا ضغطنا على الزر `<button>` :

- `event.target` → `<button>` ✅
- `event.currentTarget` → `<div class="parent">` ✅

> **ليه؟** لأن الزر هو اللي اتضغط (target)، لكن الحدث متعلق بـ (currentTarget) `div` اللي عليه الـ listener.

🧠 مثال واقعي من الحياة:

- ال **target**: إيدك لمست الزر نفسه.
- ال**currentTarget**: لكن الكاميرا (الـ listener) متعلقة بالسقف، وبتسجل كل اللي بيحصل في الغرفة.

🟣 ثانياً: Event Bubbling & Capturing

لما بيحصل حدث (زي `click`) في عنصر، ممكن يتنقل لأعلى أو لأسفل في شجرة الـ DOM:

🔶 1. Bubbling (الافتراضي)

- الحدث بيبدأ من العنصر اللي حصل عليه الحدث فعليًا (child)
- وبعدين **يصعد** إلى العناصر اللي فوقه (الأب، الجد...)

```
<div class="parent">
  <button class="child">Click</button>
</div>

document.querySelector(".parent").addEventListener("click", () => {
  console.log("Parent clicked");
});

document.querySelector(".child").addEventListener("click", () => {
  console.log("Button clicked");
});
```

✅ لو ضغطت على الزر:

```
Button clicked
Parent clicked
```

🔶 2. Capturing (اختياري)

- الحدث بيبدأ من أعلى الشجرة (الجد مثلاً) وينزل لتحت حتى العنصر اللي اتضغط عليه.

علشان تفعله لازم تكتب `{ capture: true }` كـ option في `addEventListener`

```
document.querySelector(".parent").addEventListener(
  "click",
  () => {
    console.log("Parent (capturing)");
  },
  { capture: true }
);
```

النتيجة هتكون:

```
Parent (capturing)
Button clicked
```

✅ استخدام Capturing (نادرًا)

> ال Capturing بنستخدمه **في حالات خاصة** جدًا، لما نحتاج نتصرف قبل ما يوصل الحدث للعنصر الصغير.

🟠 1. منع تنفيذ أحداث معينة مبكرًا

مثلاً: عايز أمنع كليك على زر معين **قبل** ما يوصل الحدث له.

```
document.body.addEventListener(
  "click",
  function (e) {
    console.log("Captured click");
    e.stopPropagation(); // هيوقف الحدث من الوصول للعناصر اللي بعده
  },
  true // ده معناه capture mode
);
```

🟠 2. التعامل مع عناصر بتعتمد على جهات خارجية

لو عندك مكون جاي من مكتبة خارجية بيستخدم `bubbling` وبيأثر على التطبيق، ممكن تضبطي سلوك الحدث عن طريق `capturing`.

🔴 تحذير:

> **Don't overuse capturing!**

- دايمًا ابدئ بـ bubbling، واستخدم **capturing فقط لو فعلاً محتاج تتدخل قبل الحدث ما يوصل للعنصر.**

🔷 `preventDefault()`

هي **دالة موجودة داخل كائن الحدث (event object)**، وظيفتها ببساطة:

> ❌ **تمنع السلوك الافتراضي (الطبيعي) للعنصر اللي عليه الحدث.**

🧠 يعني إيه "السلوك الافتراضي"؟

هو التصرف اللي بيحصل بشكل تلقائي لما يحصل حدث معين.

✅ أمثلة توضح `preventDefault()` بشكل واقعي:

1. روابط `<a>`

لما تضغط على رابط، المتصفح بيروح مباشرة للعنوان المكتوب في `href`.
لكن ممكن تمنع ده باستخدام `preventDefault()`:

```
<a href="https://www.google.com" id="myLink">Click Me</a>

const link = document.getElementById("myLink");

link.addEventListener("click", function (event) {
  event.preventDefault(); // ❌ بيمنع فتح الرابط
```

```
  console.log("Link clicked, but no navigation");
});
```

2. نموذج Form

بشكل افتراضي، لما تضغط "submit"، الصفحة بتعمل **reload** أو بتروح لعنوان جديد.
لو عايز تمنع ده:

```
<form id="myForm">
  <input type="text" placeholder="Your name" />
  <button type="submit">Send</button>
</form>
```

```
const form = document.getElementById("myForm");

form.addEventListener("submit", function (e) {
  e.preventDefault(); // ❌ يمنع الـ reload
  console.log("Form submitted without reload!");
});
```

3. زر داخل `<form>`

لو عندك زر جوه form وعايز تخليه ينفذ حاجة بدون إرسال البيانات:

```
<form id="signupForm">
  <input type="text" name="email" />
  <button id="checkBtn">Check</button>
</form>
```

```
document.getElementById("checkBtn").addEventListener("click", function (e) {
  e.preventDefault(); // يمنع إرسال الفورم
  console.log("Checking email before submit...");
});
```

📌 ما معنى "Accessibility"؟

إمكانية الوصول (Accessibility) معناها إن الموقع أو التطبيق يقدر يستخدمه **أي شخص، بما فيهم الأشخاص ذوي الإعاقة** — مثل المكفوفين، أو من يستخدمون الكيبورد فقط، أو من لديهم مشاكل سمعية أو حركية.

✅ الهدف من Accessibility:

أن تكتب كود JavaScript (وHTML + CSS) **يدعم أدوات المساعدة** مثل:

- قارئات الشاشة (Screen Readers)
- التنقل بالكيبورد (Tab, Shift+Tab, Enter, etc)
- التباين العالي (High Contrast)
- إمكانية تكبير النص بسهولة

◆ Accessibility ودوره في JavaScript

1. ✅ إضافة الخصائص الصحيحة لعناصر DOM

أحيانًا نستخدم عناصر مش تفاعلية (زي `<div>` أو `<span>`) لكن بنخليها تشتغل زي زرار، فمهم نضيف لها خصائص الوصول المناسبة:

❌ خطأ:

```
<div onclick="doSomething()">Click me</div>
```

✅ صحيح:

```
<div role="button" tabindex="0" onclick="doSomething()"
onkeydown="handleKey(event)">
  Click me
</div>
```

ليه؟ 👇

الخاصية	معناها
`role="button"`	بتقول لقارئ الشاشة إن العنصر ده زرار
`tabindex="0"`	يخلي العنصر قابل للتنقل بالـTab
`onkeydown`	يدعم الضغط على Enter/Space بالكيبورد

2. ◆ استخدم عناصر HTML الحقيقية بدل ما "تخترع"

بدل ما تعمل زر `<div>` ، استخدم `<button>` الحقيقي لأنه مدعوم تلقائيًا بقارئات الشاشة و keyboard.

✅ مثال:

```
<button onclick="submitForm()">Submit</button>
```

3. ◆ إضافة attributes `aria-*`

لو عندك عناصر تفاعلية، ممكن تساعد قارئ الشاشة بمزيد من الوصف:

مثال:

```
<input type="text" aria-label="Your name">
```

ال `aria-label` بتحدد وصف واضح للعنصر لقارئ الشاشة

🔷 أمثلة شائعة لاستخدام JavaScript لتحسين الـ Accessibility

✅ التحكم بالفوكس (Focus)

```
document.getElementById("username").focus();
```

لما تفتح صفحة تسجيل دخول، ممكن تحط الفوكس على أول خانة تلقائيًا.

✅ التنقل بالكيبورد

```
element.addEventListener("keydown", (event) => {
  if (event.key === "Enter") {
 // `doSomething` هو اسم الدالة (function name).
    doSomething();
  }
});
```

✅ الرسائل الحية (Live Regions)

لو في عنصر محتوى بيتغير، ممكن تخليه يُقرأ تلقائيًا:

```
<div aria-live="polite" id="message">Welcome!</div>
```

```
document.getElementById("message").textContent = "Form submitted successfully!";
```

💡 ملاحظات عملية

- ✅ استخدم عناصر HTML المناسبة دائمًا.
- ❌ تجنب استخدام JavaScript لتعطيل خصائص HTML الطبيعية (زي إزالة التركيز أو إلغاء التفاعل).
- ✅ اختبر شغلك بأداة screen reader (زي NVDA أو VoiceOver).
- ✅ استخدم الكيبورد فقط لتجرب التنقل في موقعك.

مصطلح Order of Operations في JavaScript (وكل لغات البرمجة) يعني:

ترتيب تنفيذ العمليات الرياضية أو المنطقية في التعبيرات (expressions)، لما يكون فيه أكثر من عملية.

✅ القاعدة الذهبية:

🧮 PEMDAS أو BODMAS

- **P / B** → Parentheses / Brackets `( )`
- **E / O** → Exponents (القوى – مش شائعة في JS)
- **MD** → Multiplication and Division (من اليسار لليمين)
- **AS** → Addition and Subtraction (من اليسار لليمين)

🟢 مثال بسيط:

```
let result = 5 + 3 * 2;
console.log(result); // 11
```

✅ لأن:

1. الضرب (`*`) يتم أولًا → `6 = 2 * 3`
2. بعدين الجمع → `11 = 6 + 5`

🔵 مثال باستخدام الأقواس:

```
let result = (5 + 3) * 2;
console.log(result); // 16
```

✅ لأن:

1. ما داخل القوس أولًا → `8 = 3 + 5`
2. بعدين الضرب → `16 = 2 * 8`

🟡 مثال أطول:

```
let result = 10 + 4 * 3 - (6 / 2);
```

الترتيب:

1. القوس → `3 = 2 / 6`
2. الضرب → `12 = 3 * 4`
3. ثم الجمع والطرح من اليسار لليمين:

- `10 + 12 = 22`
- `22 - 3 = 19`

⚠️ ملاحظة مهمة:

- في JavaScript، **القوس له أولوية قصوى**.
- العمليات بتُنفذ **من اليسار إلى اليمين** إذا لها نفس الأولوية.

✅ ما هي if statement؟

هي طريقة في البرمجة بنستخدمها عشان **ننفّذ كود معين فقط إذا تحقق شرط معيّن**.

🧠 القاعدة الأساسية:

```
if (condition) {
  // كود هيتنفذ لو الشرط true
}
```

```
let age = 20;

if (age >= 18) {
  console.log("You are allowed to vote.");
}
```

📌 **الشرط** هو: `age >= 18`
✅ لو الشرط true (يعني العمر 18 أو أكتر)، هيتطبع: `".You are allowed to vote"`
❌ لو الشرط false مش هيطبع حاجة.

🔁 إضافة else:

```
let age = 16;

if (age >= 18) {
  console.log("You are allowed to vote.");
} else {
  console.log("You are too young to vote.");
}
```

📌 دلوقتي:

- لو `age >= 18` ⟶ هيطبع الجملة الأولى
- غير كده ⟶ الجملة التانية

🔀 استخدام else if:

لو عندك أكتر من شرط، استخدم `else if`

```
let score = 85;

if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else if (score >= 70) {
  console.log("Grade: C");
} else {
  console.log("Fail");
}
```

💡 مثال واقعي جدًا:

```
let temperature = 30;

if (temperature > 35) {
  console.log("It's too hot today!");
} else if (temperature > 20) {
  console.log("Nice weather.");
} else {
  console.log("It's cold.");
}
```

✅ 1. الطريقة المختصرة (Ternary Operator)

```
let age = 20;
let status = (age >= 18) ? "Adult" : "Minor";
console.log(status);
```

✅ 2. استخدام `&&` للتنفيذ إذا كان الشرط صحيح:

```
let isLoggedIn = true;
isLoggedIn && console.log("Welcome back!");
```

✅ 4. استخدام `||` لو عايزة قيمة بديلة لو الشرط مش متحقق:

```
let userName = "";
let displayName = userName || "Guest";
console.log(displayName); // "Guest"
```

🧠 ثانيًا: شرح أنواع الـ Operators (العمليات)

🔢 1. Comparison Operators (مقارنة)

الرمز	المعنى	مثال	الناتج
`==`	يساوي (القيمة فقط)	`5 == "5"`	✅ true
`===`	يساوي (قيمة ونوع)	`5 === "5"`	❌ false
`!=`	لا يساوي	`5 != "5"`	❌ false
`!==`	لا يساوي (قيمة ونوع)	`5 !== "5"`	✅ true
`>`	أكبر من	`7 > 5`	✅ true
`<`	أصغر من	`3 < 10`	✅ true
`>=`	أكبر من أو يساوي	`5 >= 5`	✅ true
`<=`	أصغر من أو يساوي	`4 <= 5`	✅ true

🔁 2. Logical Operators (منطقية)

الرمز	الاسم	المعنى	مثال
`&&`	AND	يتحقق لو الشرطين معًا صح	`age > 18 && isLoggedIn`
` `		`	OR
`!`	NOT	يعكس القيمة (صح ⇨ غلط والعكس)	`!isLoggedIn` ⇨ false ⇨ true

➕ 3. Arithmetic Operators (رياضية)

الرمز	المعنى	مثال	الناتج
`+`	جمع	`5 + 2`	7
`-`	طرح	`5 - 2`	3
`*`	ضرب	`5 * 2`	10
`/`	قسمة	`6 / 2`	3
`%`	باقي القسمة	`5 % 2`	1
`**`	أس (Power)	`2 ** 3`	8

🧮 4. Assignment Operators (إسناد)

الرمز	المعنى	مثال	الناتج
`=`	إسناد قيمة	`x = 5`	`x = 5`
`+=`	إضافة ثم إسناد	`x += 3` ⇨ `x = x+3`	
`-=`	طرح ثم إسناد	`x -= 2` ⇨ `x = x-2`	
`*=`	ضرب ثم إسناد	`x *= 2`	
`/=`	قسمة ثم إسناد	`x /= 2`	
`%=`	باقي القسمة ثم إسناد	`x %= 2`	

✅ مثال شامل واقعي:

```
let userAge = 20;
let isStudent = true;

if (userAge >= 18 && isStudent) {
  console.log("You're an adult student.");
} else if (userAge >= 18) {
  console.log("You're an adult, but not a student.");
} else {
  console.log("You're underage.");
}
```

🌟 ما معنى Truthy و Falsy في JavaScript؟

في JavaScript، **كل القيم** يتم تحويلها إلى **Boolean (true أو false)** عندما نستخدمها في شرط (مثل `if` , `while` , `for`).

💡 فيه قيم تعتبر **"false" تلقائيًا (Falsy)**، والباقي يعتبر **"true" تلقائيًا (Truthy)**.

✅ القيم Falsy (يعني JavaScript بتعتبرها false في الشرط)

القيمة	المعنى
`false`	القيمة المنطقية false
`0`	الرقم صفر
`-0`	سالب صفر (نادر الاستخدام)
`""`	نص فاضي
`null`	لا يوجد قيمة
`undefined`	متغير لم يتم تعيينه بعد
`NaN`	Not a Number

✅ مثال عملي:

```
let name = "";

if (name) {
  console.log("Hello " + name);
} else {
  console.log("Please enter your name.");
}
// النتيجة: Please enter your name.
```

✅ القيم Truthy (يعني JavaScript بتعتبرها true في الشرط)

كل القيم **ما عدا** اللي فوق تعتبر Truthy، مثل:

القيمة	تعتبر true لأن:
`"Ayaat"`	نص غير فاضي
`42`	أي رقم غير صفر
`-5`	سالب لكن مش صفر
`{}`	كائن فاضي يعتبر قيمة
`[]`	مصفوفة فاضية تعتبر قيمة
`function() {}`	دالة
`Infinity`	قيمة لا نهائية تعتبر true

✅ مثال عملي:

```
let age = 25;

if (age) {
  console.log("Age is defined");
} else {
  console.log("Age is missing");
}
// النتيجة: Age is defined
```

💡 استخدامات عملية لـ Truthy و Falsy:

✅ 1. التحقق من وجود قيمة:

```
if (userName) {
  greet(userName);
} else {
  askForName();
}
```

✅ 2. القيم الافتراضية:

```
let name = inputName || "Guest";
console.log(name);
```

إذا `inputName` فاضي أو Falsy، هيطبع "Guest".

🔄 Coercion (التحويل التلقائي أو الإجباري للنوع)

📌 ما معنى Coercion؟

هو لما JavaScript **تغيّر نوع البيانات تلقائيًا أو يدويًا** من نوع لآخر، زي مثلًا من `string` إلى `number` أو العكس.

🔷 أنواع Coercion:

1. Implicit Coercion (التحويل الضمني)

> تغيّر نوع البيانات **تلقائيًا** بدون ما تطلب منها JavaScript.

🔶 **مثال:**

```
let result = "5" * 2;  // 10
```

- ال `"5"` (string) تم **تحويلها تلقائيًا** إلى `5` (number) لأن فيه ضرب.
- فتكون النتيجة `10` .

📌 لكن لو كانت زائد `+` بدل ضرب:

```
let result = "5" + 2;  // "52"
```

هنا حصل **concatenation** (دمج)، والرقم 2 اتحول إلى `"2"` ، والنتيجة سلسلة نصية `"52"` .

2. Explicit Coercion (التحويل الصريح / الإجباري)

> انت **بتطلب** من جافاسكريبت تحوّل النوع بنفسك باستخدام دوال أو طرق.

🔶 **مثال:**

```
let str = "123";
let num = Number(str);  // num = 123
```

أمثلة أخرى:

التحويل	الطريقة
إلى رقم	`Number("5")`
إلى نص	`99` أو `..toString()` `String(99)`
إلى Boolean	`Boolean(1)` أو `Boolean("")`

💡 مثال عملي:

```
let score = "80";

// implicit coercion
let result1 = score > 50;   // true  → "80" تم تحويلها لرقم

// explicit coercion
let result2 = Number(score) + 20;  // 100
```

✅ تدريب صغير:

```
console.log("10" - 5);     // ?
console.log("5" + true);   // ?
console.log(Number(false)); // ?
console.log(Boolean(" ")); // ?
```

قولي ناتج كل سطر وليه.

✅ 1. `console.log("10" - 5);` → 5

- ✔ صح.
- `"10"` (string) اتحولت تلقائيًا إلى رقم 10.
- ✅ السبب: علامة `-` دايمًا بتجبر جافاسكريبت إنها **تحوّل النص لرقم** (Implicit coercion).
- 🔁 العملية: `10 - 5 = 5`.

❌ 2. `console.log("5" + true);` → "5true"

- ✔ الإجابة صح، لكن خليني أوضح أكتر ليه:
- `+` هنا دمج لأنه لو فيه **string**، جافاسكريبت تحوّل الباقي **لنص**.

- `"5" + true` → `"5" + "true"` → `"5true"` ✅
- **نوع القيمة الناتجة:** `string` .

✅ 3. `console.log(Number(false));` → 0

- ✅ صح.
- التحويل صريح: `false` عند تحويله إلى `number` يعطي `0` .

القيمة	Number(...)
`false`	`0`
`true`	`1`
`null`	`0`
`undefined`	`NaN`

✅ 4. `console.log(Boolean(" "));` → true

- ✅ صح.
- أي **string غير فاضي حتى لو فيها مسافة** تعتبر Truthy.
- `" "` فيها مسافة = `Boolean(" ") = true` → **Truthy**

الـ statement `switch`

في JavaScript هي طريقة بديلة لاستخدام سلسلة طويلة من جمل `if...else if` . بتستخدم لما يكون عندك متغير أو قيمة وبتقارنها بعدة اختيارات (حالات - `cases`).

✅ الشكل العام (Syntax):

```
switch(expression) {
  case value1:
    // الكود لو كانت القيمة تساوي value1
    break;
  case value2:
    // الكود لو كانت القيمة تساوي value2
    break;
  default:
    // الكود لو ما فيش حالة مطابقة
}
```

💡 شرح:

- **expression**: المتغير أو القيمة اللي هتقارنها.

- **case**: كل حالة بتجرب تطابق مع الـ expression.
- **break**: بيوقف تنفيذ باقي الحالات بعد ما يلاقي حالة مطابقة.
- **default**: (اختياري) حالة افتراضية لو مفيش ولا حالة مطابقة.

🟢 مثال بسيط:

```
let day = 3;

switch(day) {
  case 1:
    console.log("Sunday");
    break;
  case 2:
    console.log("Monday");
    break;
  case 3:
    console.log("Tuesday"); // Tuesday
    break;
  default:
    console.log("Another day");
}
```

🟠 مثال واقعي:

```
let role = "admin";

switch(role) {
  case "admin":
    console.log("You have full access");
    break;
  case "editor":
    console.log("You can edit content");
    break;
  case "viewer":
    console.log("You can only view content");
    break;
  default:
    console.log("Unknown role");
}
```

❗ ملاحظات مهمة:

- **لو نسيت break**: هيكمل ينفذ باقي الـ cases حتى لو الحالة مش مطابقة (اسمه fall-through).
- ال `switch` بيقارن باستخدام `===` (مطابقة تامة في النوع والقيمة).

🕒 ما هي Intervals and Timers؟

عامل زي مثلا مظبط المنبه على 15 دقيقة و بعدها يديلي انذار , وده بيشتغل مرة واحدة : Timers
نفس مثال المنبه ولكن هنا الفرق ان كل 15 دقيقة المنبه هايضرب لانه الموضوع بيتكرر : Intervals

ال JavaScript فيها دوال مدمجة للتحكم في **تأخير تنفيذ كود** أو **تنفيذه بشكل متكرر بعد وقت معين**.
فيه نوعين أساسيين:

Timer Function	وظيفة الدالة
`setTimeout()`	يشغل الكود **مرة واحدة بعد وقت معين**
`setInterval()`	يشغل الكود **كل فترة زمنية متكررة**
`clearTimeout()`	يوقف مؤقت `setTimeout`
`clearInterval()`	يوقف مؤقت `setInterval`

🔷 1. `setTimeout()`

بتستخدم لتأخير تنفيذ كود معين **لمدة محددة مرة واحدة فقط**.

✅ الصيغة:

```
setTimeout(callback, delayInMilliseconds);
```

- `callback` : الدالة أو الكود اللي هيتنفذ.
- `delay` : الوقت بالملي ثانية (1000 ms = 1 ثانية).

🧠 مثال:

```
setTimeout(() => {
  console.log("Welcome, Ayaat! (after 2 seconds)");
}, 2000);
```

⏱ بعد 2 ثانية، يطبع الرسالة.

🔷 2. `setInterval()`

بيكرر تنفيذ الكود كل فترة زمنية معينة **بدون توقف** (إلا لو استخدمت `clearInterval`).

✅ الصيغة:

```
setInterval(callback, intervalInMilliseconds);
```

🧠 مثال:

```
setInterval(() => {
  console.log("Time is passing...");
}, 1000);
```

⏱️ كل ثانية، يطبع "Time is passing..."

🔷 3. `clearTimeout()` و `clearInterval()`

بيوقف المؤقت (سواء timeout أو interval).

🧠 مثال مع `setTimeout()` :

```
let timer = setTimeout(() => {
  console.log("You will not see this");
}, 3000);

clearTimeout(timer); // يوقف المؤقت قبل ما يشتغل
```

🧠 مثال مع `setInterval()` :

```
let counter = 0;
let myInterval = setInterval(() => {
  counter++;
  console.log(`Count: ${counter}`);
  if (counter === 5) {
    clearInterval(myInterval);
    console.log("Done counting");
  }
}, 1000);
```

🟣 أمثلة واقعية:

✅ ⏰ مثال: عرض رسالة ترحيب بعد فتح الصفحة بـ 3 ثواني

```
setTimeout(() => {
  alert("Welcome to my website!");
}, 3000);
```

✅ 🔁 مثال: عداد يعد من 1 إلى 10 ويقف

```
let i = 1;
let interval = setInterval(() => {
  console.log(i);
  i++;
  if (i > 10) {
    clearInterval(interval);
  }
}, 1000);
```

⚠️ ملاحظات مهمة:

- ال JavaScript مش "توقف" البرنامج؛ هي بس تأجل تنفيذ الكود، وكل حاجة تانية شغالة.
- التوقيت مش دايمًا دقيق جدًا، لأنه بيعتمد على الـ **Event Loop**.
- ال `setInterval` ممكن يتراكم لو الكود جوه بياخد وقت طويل.

🧪 تدريب سريع:

🎯 *اكتب كود يعرض "Typing..." كل 2 ثانية، ويقف بعد 6 ثواني فقط.*

```
let interval=setInterval(()=>{
console.log("Typing...")
},2000);

let timer=setTimeout(()=>{
clearInterval(interval);
console.log('stopped typing')
},6000)
```

Object reference V.S Value :

🧠 أولًا: Types in JavaScript

في JavaScript عندنا نوعين من البيانات من حيث طريقة التخزين والتعامل:

النوع	أمثلة	طريقة التخزين
Primitive types	string, number, boolean, null, undefined, symbol, bigint	يتم تخزين القيمة نفسها مباشرة
Reference types	object, array, function	يتم تخزين **مرجع (عنوان)** للمكان اللي فيه البيانات

✳️ القيمة (Value) – Primitive

يعني لما تعمل متغير فيه رقم أو نص، بيتم **نسخ القيمة**.

مثال:

```
let a = 5;
let b = a;

b = 10;

console.log(a); // 5
console.log(b); // 10
```

🔎 **الشرح:**

ال `b` أخدت نسخة من قيمة `a` .
فلما غيرنا `b` ، ما أثرش على `a` .

✅ لأنهم **قيم منفصلة**.

🧱 المرجع (Reference) – Object, Array

لما تعمل متغير فيه كائن أو مصفوفة، بيتخزن **العنوان** اللي فيه البيانات، مش البيانات نفسها.

مثال:

```
let person1 = { name: "Ayaat" };
let person2 = person1;

person2.name = "Suzan";

console.log(person1.name); // Suzan ❗
console.log(person2.name); // Suzan
```

🔎 **الشرح:**
ال `person2` مش نسخة، هو **نفس الكائن** اللي في `person1` .
فلما غيرنا `name` من خلال `person2` ، اتغيرت كمان في `person1` ، لأنهم **بيشيروا لنفس المرجع**.

🎯 عايز تنسخ كائن بدون ما تأثر على الأصلي؟ استخدم:

✅ Object spread:

```
let person1 = { name: "Ayaat" };
let person2 = { ...person1 };

person2.name = "Suzan";

console.log(person1.name); // Ayaat
```

✅ JSON methods (نسخة عميقة):

```
let person1 = { name: "Ayaat" };
let person2 = JSON.parse(JSON.stringify(person1));
```

🧪 مثال واقعي من الحياة:

تخيل إن:

- ال**Primitive value** زي ورقة مكتوب فيها رقم، كل واحد بياخد نسخة منها.
- ال **Reference value** زي مفتاح بيتفتح بيه نفس الدولاب، لو واحد فتح الدولاب وغير حاجة، التاني هيشوف التغيير.

✅ ما هو الـ `Map` ؟

ال `Map` هي **بنية بيانات جديدة** ظهرت مع ES6 في JavaScript.
تشبه الكائن (`object`) لأنها تخزن أزواج `key-value` ، لكن بتوفر **مزايا إضافية وقوة أكبر**.

🔶 إنشاء Map:

```
const myMap = new Map();
```

🔶 إضافة عناصر:

```
myMap.set("name", "Ayaat");
myMap.set("age", 23);
```

🔶 الحصول على قيمة:

```
console.log(myMap.get("name")); // Ayaat
```

🔶 حذف عنصر:

```
myMap.delete("age");
```

🔶 التحقق من وجود مفتاح:

```
myMap.has("name"); // true
```

🔶 عدد العناصر:

```
console.log(myMap.size); // عدد العناصر
```

🔶 حذف الكل:

```
myMap.clear();
```

🔶 التكرار (looping):

```
for (let [key, value] of myMap) {
  console.log(`${key}: ${value}`);
}
```

✅ الفرق الكامل بين `Map` و `Object`

الخاصية	`Object`	`Map`
🔑 أنواع المفاتيح	فقط string أو symbol	أي نوع (string, number, object, function)
🔄 الترتيب	غير مضمون الترتيب	يحافظ على الترتيب
🔍 البحث عن مفتاح	أبطأ في الكائنات الكبيرة	أسرع
📏 عدد العناصر	يجب العد يدويًا	جاهزة property `size`
🔁 التكرار	لازم تستخدم `for..in` أو `Object.entries`	يدعم `for..of` مباشرًا
🧹 التنظيف	لازم تحذفي كل مفتاح	موجود `.clear()`
📦 مناسب لـ	هيكل بيانات بسيط وثابت	بيانات متغيرة أو حجم كبير أو dynamic keys
🔐 مفاتيح معقدة	لا يمكن استخدام object كمفتاح	نعم يمكن

✅ متى تستخدم كل واحد؟

🟢 استخدم `Object` لما:

- بتمثل كيان حقيقي مثل `product` , `user`.

- بتتعامل مع بيانات ثابتة أو بسيطة.
- مش مهم الترتيب.
- مفاتيحك دايمًا نصوص.

```
const user = {
  name: "Ayaat",
  age: 22,
  isStudent: true
};
```

🟢 استخدم `Map` لما:

- عندك عدد كبير من البيانات أو بتعدل البيانات كثير.
- مفاتيحك مش كلها نصوص (object, function...).
- عايز تحافظ على ترتيب الإدخال.
- محتاج أداء أسرع.

```
const map = new Map();
map.set({ id: 1 }, "user1");
map.set(() => {}, "callback");
```

✅ سيناريو انترفيو – سؤال شائع:

🧠 **المقابلة:**

"What's the difference between a Map and an Object? When would you choose one over the other?"

🧑‍💻 **إجابتك المثالية:**

> Objects are great for representing structured data with known keys, like users or products. But Maps offer more flexibility for dynamic key-value storage, better performance on large datasets, and support for any key type including objects. I use Map when I need ordered keys, fast lookup, and complex keys, and Object when I need a simple data structure with fixed fields.

✅ ملاحظات مهمة:

- ال `Map` لا يحتوي على خصائص من الـ prototype، فـ `for..in` لن يظهر لك المفاتيح.
- ال `Map` يتعامل مع المفتاح `1` والمفتاح `"1"` على أنهم مختلفين، بعكس الكائن.
- يمكنك تحويل كائن إلى خريطة بسهولة:

```
const obj = { a: 1, b: 2 };
```

```
const map = new Map(Object.entries(obj));
```

📚 ما هو الـ Array في JavaScript؟

✅ تعريف:

ال `Array` هو نوع خاص من المتغيرات بيقدر يخزن **أكثر من قيمة** داخل متغير واحد.

```
let cars = ["BMW", "Volvo", "Mini"];
```

✅ لماذا نستخدمه؟

بدل ما تعمل متغير لكل قيمة:

```
let car1 = "BMW";
let car2 = "Volvo";
let car3 = "Mini";
```

بتخزنيهم في مصفوفة واحدة

✨ طريقة إنشاء Array

✅ 1. باستخدام الأقواس `[]` :

```
let fruits = ["apple", "banana", "mango"];
```

✅ 2. باستخدام `new Array()` :

```
let fruits = new Array("apple", "banana", "mango");
```

🚫 لكن الأفضل استخدام الطريقة الأولى لأنها أبسط وأوضح.

🔢 الوصول إلى عناصر المصفوفة (Accessing Elements)

```
let fruits = ["apple", "banana", "mango"];
console.log(fruits[0]); // apple
```

📌 المصفوفة تبدأ من **index 0**

🔄 تغيير القيم (Change an Element)

```
fruits[1] = "orange"; // banana تتغير إلى orange
```

➕ طول المصفوفة (Length)

```
console.log(fruits.length); // 3
```

➕ آخر عنصر في المصفوفة

```
let last = fruits[fruits.length - 1];
```

✅ Looping through Arrays

🔁 باستخدام for:

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

🔁 باستخدام forEach:

```
fruits.forEach(function(fruit) {
  console.log(fruit);
});
```

🧠 أنواع البيانات داخل Array

ممكن تحتوي على أنواع مختلفة:

```
let mixed = ["Ayaat", 22, true];
```

ال Arrays نوعها object

```
typeof fruits; // "object"
```

📌 لكن `Array` بيبقى نوع خاص من `object` .

🧱 التحقق هل القيمة مصفوفة

```
Array.isArray(fruits); // true
```

🛠️ أهم دوال ومميزات الـ Array

الطريقة	الشرح
`push()`	يضيف عنصر في نهاية المصفوفة
`pop()`	يزيل آخر عنصر
`shift()`	يزيل أول عنصر
`unshift()`	يضيف عنصر في البداية
`indexOf()`	يرجع رقم العنصر داخل المصفوفة
`includes()`	يرجع true أو false إذا كان العنصر موجود
`slice(start, end)`	يرجع نسخة من جزء من المصفوفة
`splice(start, count, ...)`	يضيف أو يحذف عناصر
`concat()`	يدمج مصفوفتين
`join()`	يحول المصفوفة إلى string
`reverse()`	يعكس ترتيب العناصر
`sort()`	يرتب العناصر

🌟 مثال واقعي:

📦 إدارة سلة منتجات:

```
let cart = ["T-shirt", "Pants"];
cart.push("Shoes");       // أضف منتج
cart.pop();               // احذف آخر منتج
cart.includes("Pants");   // هل السلة بها "Pants"؟
```

🧪 اختبار سريع:

1. ما هو الإندكس الخاص بأول عنصر في المصفوفة؟
2. كيف تضيف عنصر في البداية؟
3. ما الفرق بين `push()` و `unshift()` ؟
4. كيف تتأكد أن متغير معين هو مصفوفة؟

🧠 يعني إيه "mutable" و "immutable"؟

- **Mutable** = بتغير المصفوفة الأصلية (بتعدل فيها).

- **Immutable** = ما بتغيرش المصفوفة الأصلية، بترجع نسخة جديدة.

✅ 1. Mutable Array Methods

> بتعدل مباشرة على المصفوفة الأصلية.

Method	Description (الوصف)	مثال
`push()`	يضيف عنصر في **نهاية** المصفوفة	`arr.push(5)`
`pop()`	يحذف آخر عنصر	`arr.pop()`
`shift()`	يحذف أول عنصر	`arr.shift()`
`unshift()`	يضيف عنصر في **بداية** المصفوفة	`arr.unshift(1)`
`splice()`	يضيف أو يحذف عناصر في مكان معين	`arr.splice(2, 1, "new")`
`sort()`	يرتب العناصر و**يغير الترتيب الأصلي**	`arr.sort()`
`reverse()`	يعكس ترتيب العناصر	`arr.reverse()`

🔧 مثال عملي:

```
let names = ["Ayaat", "Suzan", "Tarek"];
names.push("Ahmed");
console.log(names); // ["Ayaat", "Suzan", "Tarek", "Ahmed"]
```

> ✅ `push()` غيّرت المصفوفة الأصلية، فـ ده **mutable**.

✅ 2. Immutable Array Methods

> ما بتعدلش في المصفوفة الأصلية، لكن بترجع نسخة جديدة.

Method	Description (الوصف)	مثال
`slice()`	يقطع جزء من المصفوفة ويرجعه في نسخة جديدة	`arr.slice(1,3)`
`concat()`	يدمج مصفوفتين ويرجع نسخة جديدة	`arr.concat([4,5])`
`map()`	ينفذ دالة على كل عنصر ويرجع نسخة جديدة	`arr.map(x => x * 2)`
`filter()`	يرجع العناصر اللي تحقق شرط معين	`arr.filter(x => x > 10)`
`reduce()`	يحوّل المصفوفة لقيمة واحدة	`arr.reduce((a,b) => a + b)`
`join()`	يحول المصفوفة إلى نص (string)	`arr.join(", ")`
`includes()`	يبحث عن عنصر ويرجع true أو false	`arr.includes("Ayaat")`
`find()`	يرجع أول عنصر يحقق شرط	`arr.find(x => x > 5)`

🔍 مثال عملي:

```
let numbers = [1, 2, 3];
let doubled = numbers.map(num => num * 2);
console.log(numbers); // [1, 2, 3]
console.log(doubled); // [2, 4, 6]
```

✅ `map()` ما عدلش `numbers` الأصلية، فده **immutable**.

📌 ملاحظات مهمة:

- استخدم **immutable methods** لما تحب تحفظ البيانات الأصلية بدون تعديل.
- استخدم **mutable methods** لما يكون تعديل المصفوفة الأصلية مقصود.
- في React أو أي إطار عمل حديث، بيفضّلوا **immutable methods** عشان الكود يبقى predictable وسهل التتبع.

💡 سؤال إنترفيو محتمل:

Q: What's the difference between `slice()` and `splice()` ?
A:

- `slice()` → **immutable**، بيرجع نسخة جديدة بدون تغيير الأصل.
- `splice()` → **mutable**، بيعدل في المصفوفة الأصلية (يحذف أو يضيف).

✅ باقي الميثود Array Methods

🧠 الطريقة	🔍 الوظيفة	🔁 Mutable؟	✅ مثال
`length`	ترجع عدد العناصر في المصفوفة	❌ Immutable	`arr.length`
`toString()`	يحول المصفوفة إلى نص مفصول بفواصل	❌ Immutable	`arr.toString()`
`at(index)`	يرجع العنصر عند index معين (يدعم القيم السالبة)	❌ Immutable	`arr.at(-1)`
`delete arr[index]`	يحذف عنصر لكن يسيب مكانه فارغ (`undefined`)	✅ Mutable (خطير)	`delete arr[2]`
`copyWithin()`	ينسخ جزء من المصفوفة ويضعه في مكان آخر داخلها	✅ Mutable	`arr.copyWithin(1, 3)`
`flat()`	يحول مصفوفة متداخلة إلى مصفوفة مسطحة (flat)	❌ Immutable	`[1, [2,3]].flat()`
`toSpliced()`	نفس splice لكن ما بيعدلش الأصل، بيرجع نسخة جديدة (ES2023)	❌ Immutable	`arr.toSpliced(1, 2, "Ayaat")`

✅ ملاحظات مهمة:

- ال **delete** يسيب `undefined` ، فاستخدامه غير مستحب، استبدله بـ `splice()` .
- ال **toSpliced()** حديث، ومفيد لو عايز تشتغل زي `splice()` لكن من غير ما تعدل الأصل.
- ال **copyWithin()** ممكن يسبب لخبطة لأنه بيعدّل في المصفوفة بشكل داخلي.

💬 مثال عملي:

```
let items = ["Ayaat", "Suzan", "Mohamed", "Tarek"];
let sliced = items.slice(1, 3);
let spliced = items.splice(1, 2, "Ali");

console.log(sliced);   // ["Suzan", "Mohamed"]
console.log(spliced);  // ["Suzan", "Mohamed"]
console.log(items);    // ["Ayaat", "Ali", "Tarek"]
```

- ✅ ما عدلش الأصل `slice`
- ❌ عدل في الأصل `splice`

يوجد 3 انواع من الميثود الخاصة بال Array :

🔷 أولاً: Static Methods

هي الميثودات الموجودة على كائن `Array` نفسه، مش على كل مصفوفة.

🧠 يعني: تستخدمها بالشكل ده: `Array.methodName()` مش `array.methodName()` .

✅ أمثلة:

الميثود	الوظيفة
`Array.isArray()`	يتحقق إذا كانت القيمة مصفوفة أو لا.
`Array.from()`	يحول كائن شبه مصفوفة أو قابل للتكرار إلى مصفوفة.
`Array.of()`	ينشئ مصفوفة من القيم المعطاة.

📌 مثال:

```
console.log(Array.isArray([1, 2, 3])); // true
console.log(Array.of(7, 8));           // [7, 8]
console.log(Array.from("Ayaat"));     // ['A', 'y', 'a', 'a', 't']
```

🔷 ثانياً: Instance Methods

دي هي الميثودات اللي بتستخدم على المصفوفة نفسها،

🧠 يعني: `myArray.methodName()` .

✅ أمثلة:

الميثود	الوظيفة
`push()`	يضيف عنصر في النهاية
`pop()`	يحذف آخر عنصر
`shift()`	يحذف أول عنصر
`unshift()`	يضيف عنصر في البداية
`slice()`	يرجع جزء من المصفوفة (نسخة)
`splice()`	يحذف أو يضيف عناصر (يغير المصفوفة الأصلية)
`join()`	يحول المصفوفة إلى نص بفاصل محدد

📌 مثال:

```
let arr = [1, 2, 3];
arr.push(4);         // [1, 2, 3, 4]
let part = arr.slice(1, 3);  // [2, 3] (نسخة جديدة)
```

🔷 ثالثاً: Callback Methods

هي الميثودات اللي **تأخذ دالة كـ parameter** وتطبقها على العناصر داخل المصفوفة.

✅ أمثلة:

الميثود	الوظيفة
`forEach()`	تنفذ دالة على كل عنصر
`map()`	تنشئ مصفوفة جديدة بناء على دالة
`filter()`	تنشئ مصفوفة جديدة بالعناصر التي تحقق شرط معين
`reduce()`	تجمع العناصر إلى قيمة واحدة حسب دالة معينة
`find()`	ترجع أول عنصر يحقق شرط معين
`some()`	ترجع true لو على الأقل عنصر واحد حقق الشرط
`every()`	ترجع true لو كل العناصر حققت الشرط

📌 مثال:

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

```
let doubled = numbers.map(num => num * 2);       // [2, 4, 6, 8]
let even = numbers.filter(num => num % 2 === 0); // [2, 4]
```

✅ ما هي `sort()` ؟

ال `sort()` هي **ميثود من ميثودات المصفوفات (Arrays)** في JavaScript، وظيفتها ترتيب العناصر داخل المصفوفة.

🟨 الاستخدام الأساسي:

```
let fruits = ["Banana", "Apple", "Orange"];
fruits.sort();
console.log(fruits);
// ["Apple", "Banana", "Orange"]
```

🔶 ملاحظات:

- بترتب العناصر **حسب الترتيب الأبجدي (alphabetical)**.
- الحروف الكبيرة تيجي قبل الصغيرة.
- بترتب العناصر **كنصوص (strings)** حتى لو أرقام!

🟥 المشكلة مع الأرقام:

```
let numbers = [100, 15, 3, 42];
numbers.sort();
console.log(numbers);
// [100, 15, 3, 42] ❌ (ترتيب نصوص مش أرقام)
```

ليه؟

لأن `sort()` بتحول العناصر لنصوص وبعدين ترتبهم حسب ترتيب الحروف، فـ `"100"` تيجي قبل `"15"` .

✅ الحل: استخدام دالة مقارنة (compare function)

```
let numbers = [100, 15, 3, 42];
numbers.sort((a, b) => a - b);
console.log(numbers);
// [3, 15, 42, 100]
```

📌 القاعدة:

```
array.sort((a, b) => a - b); // ترتيب تصاعدي
array.sort((a, b) => b - a); // ترتيب تنازلي
```

🔵 كيف تعمل دالة المقارنة؟

```
function compare(a, b) {
  if (a < b) return -1; // a قبل b
  if (a > b) return 1;  // b قبل a
  return 0;             // متساويين
}
```

🟢 أمثلة متنوعة:

🔷 ترتيب أسماء حسب الطول:

```
let names = ["Ayaat", "Mohamed", "Ali", "Fatma"];
names.sort((a, b) => a.length - b.length);
console.log(names);
// ["Ali", "Ayaat", "Fatma", "Mohamed"]
```

🔷 ترتيب كائنات (Objects) حسب خاصية:

```
let people = [
  { name: "Ayaat", age: 22 },
  { name: "Tarek", age: 30 },
  { name: "Suzan", age: 18 }
];

people.sort((a, b) => a.age - b.age);
console.log(people);
/*
[
  { name: "Suzan", age: 18 },
  { name: "Ayaat", age: 22 },
  { name: "Tarek", age: 30 }
]
*/
```

🟣 ملاحظة مهمة:

الميثود `sort()` **تُغير المصفوفة الأصلية** (✅ **mutable**)، وده ممكن يعمل مشاكل أحيانًا لو مش واخد بالك.

📌 Problem Solving:

You are given an array of student objects. Each student has a name and a list of scores. Write a function that:

1. Filters only students who passed (average score ≥ 60). *(callback method)*
2. Sorts them by average score descending. *(sort method)*
3. Extracts only their names into a new array. *(callback method)*
4. Adds a student at the beginning and another at the end. *(instance method)*
5. Finally, check if the final result is really an array or not. *(static method)*

```
let students = [
  { name: "Ayaat", scores: [70, 80, 90] },
  { name: "Tarek", scores: [40, 50, 30] },
  { name: "Suzan", scores: [60, 65, 70] },
  { name: "Khaled", scores: [55, 59, 58] },
  { name: "Atyaa", scores: [95, 90, 93] }
];
✅ لازم نحسب المتوسط أولًا:
let studentsPassed = students.filter(student => {
  let avg = student.scores.reduce((a, b) => a + b, 0) / student.scores.length;
  return avg >= 60;
});
✅ لازم نحسب المتوسط لكل طالب داخل المقارنة
let sortedStudents = [...studentsPassed].sort((a, b) => {
  let avgA = a.scores.reduce((a, b) => a + b, 0) / a.scores.length;
  let avgB = b.scores.reduce((a, b) => a + b, 0) / b.scores.length;
  return avgB - avgA;
});

let names = sortedStudents.map(student => student.name);

names.unshift("Mohamed");
names.push("Fatma");

console.log(names);                  // النتيجة النهائية
console.log(Array.isArray(names));  // true ✅
```

✅ أولاً: `for` loop

📌 التعريف:

تستخدم لما تعرف عدد التكرارات اللي محتاجاها، مثل تكرار 5 مرات.

📦 الشكل العام:

```
for (initialization; condition; increment) {
  // الكود اللي هيتنفذ
}
```

🎯 مثال:

```
for (let i = 1; i <= 5; i++) {
  console.log(`Item number ${i}`);
}
```

🧠 مثال واقعي:

بتطبع فواتير لـ 5 عملاء.

✅ ثانيًا: loop `while`

📌 التعريف:

تستخدم لما *مش متأكد من عدد التكرارات*، وبتستنى شرط يتحقق علشان توقف.

📦 الشكل العام:

```
while (condition) {
  // الكود اللي هيتنفذ
}
```

🎯 مثال:

```
let i = 1;
while (i <= 3) {
  console.log(`Processing order ${i}`);
  i++;
}
```

🧠 مثال واقعي:

بتعمل معالجة لطلبات طالما في طلبات معلقة.

✅ ثالثًا: loop `for...in`

📌 التعريف:

تستخدم لتكرار **خصائص (مفاتيح)** كائن (Object).

📦 الشكل العام:

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

🎯 مثال:

```
let person = { name: "Ayaat", age: 22, city: "Cairo" };

for (let key in person) {
  console.log(`${key} : ${person[key]}`);
}
```

🧠 مثال واقعي:

بتستعرض بيانات شخص من كائن.

📌 **ملاحظة:**
لا تستخدم `for...in` مع Arrays لأنها بترجع الـ index كـ string، مش الأفضل.

✅ رابعًا: `for...of` loop

📌 التعريف:

تستخدم لتكرار **عناصر القوائم** (Array, String, Sets, Maps).

📦 الشكل العام:

```
for (let item of iterable) {
  console.log(item);
}
```

🎯 مثال:

```
let names = ["Ayaat", "Suzan", "Tarek"];

for (let name of names) {
  console.log(`Welcome ${name}`);
}
```

🧠 مثال واقعي:

بتبعت رسائل ترحيب لكل اسم في القائمة.

✅ `do...while` Loop

📌 التعريف:

حَلقة بتنفّذ الكود **على الأقل مرة واحدة** حتى لو الشرط مش صحيح من البداية.

📦 الشكل العام:

```
do {
  // الكود اللي هيتنفذ
} while (الشرط);
```

🧠 الفرق بين `while` و `do...while` :

loop	هل الكود بيتنفذ قبل الشرط؟
`while`	لأ، بيتحقق من الشرط أولًا
`do...while`	نعم، بينفذ الكود أولًا حتى لو الشرط خطأ

🎯 مثال واقعي:

عايزة تسأل المستخدم يدخل كلمة المرور، حتى لو كتبها غلط أول مرة — هتخليه يدخلها "على الأقل مرة واحدة".

```
let password;
do {
  password = prompt("Enter your password:");
} while (password !== "1234");

console.log("Access granted!");
```

حتى لو المستخدم دخل حاجة غلط من البداية، هيرجع يتسأل لحد ما يدخل الصح.

📌 إمتى أستخدم `do...while` ؟

استخدمها لما تكون **عايز الكود يتنفذ على الأقل مرة واحدة**، زي:

- إدخال بيانات من المستخدم.
- عرض رسالة ترحيب أول مرة حتى لو الشرط غير محقق.
- تنفيذ كود مبدئي قبل التحقق من شرط الاستمرار.

✅ ما هي `forEach()` ؟

ال `forEach()` هي **طريقة (method)** تستخدم مع الـ **Arrays فقط**، وتسمح لك إنك تمرّ على كل عنصر في المصفوفة وتنفّذ عليه كود معين.

📦 الشكل العام:

```
array.forEach(function(element, index, array) {
  // الكود اللي بيتنفذ لكل عنصر
});
```

المعاملات (parameters):

المعامل	المعنى
`element`	العنصر الحالي في الـ array
`index`	رقم العنصر (اختياري)
`array`	المصفوفة نفسها (اختياري)

🧠 مثال بسيط:

```
let fruits = ["apple", "banana", "orange"];

fruits.forEach(function(fruit, index) {
  console.log(`${index + 1}: ${fruit}`);
});
/*
1: apple
2: banana
3: orange

*/
```

✅ مثال عملي:

تخيلّ عندك قائمة بأسامي طلاب، وعايز تبعت لكل واحد رسالة ترحيب:

```
let students = ["Ayaat", "Suzan", "Khaled"];

students.forEach(function(student) {
  console.log(`Hello, ${student}! Welcome to the class.`);
});
```

💡 ليه نستخدم `forEach()` بدل `for` ؟

`forEach()`	التقليدية `for`
أسهل وأبسط للقراءة	أكتر تحكم
لا يمكن استخدام break أو return لوقف التكرار	ممكن توقفيها بـ break

⚠️ ملاحظات مهمة:

- ال `forEach()` **ما بترجعش قيمة** (يعني مش زي `map()`).
- ما ينفعش تستخدم `break` أو `continue` جوا `forEach()` .
- هي **mutable** لو عدّلت على المصفوفة نفسها.

🎯 هل تشتغل مع `Objects` ؟

لا، `forEach()` تشتغل مع **Arrays فقط**. لو عايز تشتغل على كائنات، تستخدم مثلاً:

```
Object.keys(obj).forEach(...)
```

forEach() , map() , find ()

📍 أولًا: الفرق بينهم بشكل سريع

تستخدم إمتى؟	بتشتغل على إيه؟	ترجع إيه؟	Method
لما تحب تعمل تعديل على كل العناصر	كل عنصر (وتعدله أو ترجعه بشكل مختلف)	مصفوفة جديدة بنفس الطول	`map()`
لما تحب "تختار" عناصر معينة	العناصر اللي تحقق شرط معين	مصفوفة جديدة أقل أو تساوي الأصل	`filter()`
لما تحب "تلاقي" عنصر معين	أول عنصر يحقق الشرط فقط	عنصر واحد فقط أو undefined	`find()`

✅ `map()` — التعديل على كل عنصر

> ترجع **مصفوفة جديدة** بنفس عدد عناصر القديمة لكن بتعديل معين

🎓 مثال عملي:

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

let doubled = numbers.map(function(num) {
  return num * 2;
```

```
});

console.log(doubled); // [2, 4, 6, 8]
```

✅ `filter()` — فلترة العناصر حسب شرط

> ترجع **مصفوفة جديدة** فيها العناصر اللي حققت الشرط

🎓 مثال عملي:

```
let ages = [18, 25, 30, 15, 10];

let adults = ages.filter(function(age) {
  return age >= 18;
});

console.log(adults); // [18, 25, 30]
```

✅ `find()` — إيجاد أول عنصر يحقق شرط

> ترجع **أول عنصر** يحقق الشرط فقط
> لو مفيش عنصر بيحقق الشرط => `undefined`

🎓 مثال عملي:

```
let users = [
  { name: "Ayaat", age: 22 },
  { name: "Suzan", age: 25 },
  { name: "Khaled", age: 17 }
];

let found = users.find(function(user) {
  return user.age >= 18;
});

console.log(found); // { name: "Ayaat", age: 22 }
```

💡 مثال موحد على 3 الدوال

```
let students = [
  { name: "Ayaat", score: 90 },
  { name: "Suzan", score: 70 },
```

```
  { name: "Tarek", score: 40 }
];

// 1. map => خلي كل Score درجات 10 يزيد
let updatedScores = students.map(student => {
  return { ...student, score: student.score + 10 };
});

// 2. filter => اختار بس الطلاب اللي عدوا
let passedStudents = students.filter(student => student.score >= 60);

// 3. find => 50 لاقي أول طالب جاب أقل من
let weakStudent = students.find(student => student.score < 50);

console.log(updatedScores);
console.log(passedStudents);
console.log(weakStudent);
```

✅ `Array.prototype.reduce()`

📌 ما هي `reduce()` ؟

دالة بتستخدم **لتقليل عناصر المصفوفة إلى قيمة واحدة**
يعني بتمشي على كل العناصر وتجمعهم أو تبني بيهم ناتج واحد.

✨ الصيغة الأساسية:

```
array.reduce(callback, initialValue)
```

✅ المعاملات:

العنصر	المعنى
`callback`	دالة بيتم استدعاؤها على كل عنصر في المصفوفة
`initialValue`	القيمة الأولية (بيبدأ منها التجميع – مهمة جدًا!)

📌 الـ `callback` شكلها:

```
function(accumulator, currentValue, index, array) {
  // logic
}
```

المعامل	معناه
`accumulator`	القيمة المتراكمة اللي بتتجمع فيها النتيجة
`currentValue`	العنصر الحالي في المصفوفة

💡 مثال بسيط:

✅ جمع الأرقام:

```
let numbers = [10, 20, 30, 40];

let total = numbers.reduce(function(acc, current) {
  return acc + current;
}, 0);

console.log(total); // 100
```

- أول مرة: `acc = 0` , `current = 10` → return 10
- ثاني مرة: `acc = 10` , `current = 20` → return 30

🎓 مثال واقعي:

✅ نحسب المجموع الكلي لدرجات الطلاب:

```
let students = [
  { name: "Ayaat", score: 90 },
  { name: "Suzan", score: 70 },
  { name: "Tarek", score: 40 }
];

let totalScore = students.reduce((acc, student) => {
  return acc + student.score;
}, 0);

console.log(totalScore); // 200
```

✅ استخدام متقدم:

🟢 تحويل مصفوفة لأوبجكت:

```
let colors = ["red", "blue", "green"];

let result = colors.reduce((acc, color, index) => {
  acc[index] = color;
  return acc;
}, {});

console.log(result);
// {0: 'red', 1: 'blue', 2: 'green'}
```

✋ ملاحظات مهمة:

- ال `reduce()` مش بترجع نفس عدد العناصر ← بترجع **قيمة واحدة**
- لو مفيش `initialValue` ، بياخد أول عنصر كـ accumulator
- لكن دا ممكن يسبب أخطاء، فـ **يفضل تكتبي** `initialValue` **دايمًا**

❓أسئلة ممكن تيجي في إنترفيو:

1. إيه الفرق بين `reduce` و `map + filter` ؟
2. امتى تستخدم `reduce` بدل `forEach` ؟
3. اكتب كود باستخدام `reduce` لتحويل array لأوبجكت.

🟣 أولاً: ما هي `localStorage` ؟

ال `localStorage` هي **مساحة تخزين في المتصفح** (Browser) تقدر تخزن فيها بيانات **بشكل دائم**، حتى لو المستخدم قفل الصفحة أو المتصفح.

> يعني تحفظ بيانات المستخدم على جهازه، من غير ما تحتاج قاعدة بيانات أو سيرفر.

✅ أهم مميزاتها:

الميزة	التوضيح
تخزين دائم	البيانات تفضل محفوظة حتى بعد ما تقفلي المتصفح
سريعة وسهلة	بتستخدميها بكود بسيط جدًا
بس تخزن نصوص	لازم تحولي أي بيانات معقدة إلى نص `JSON`

🔶 فين بتتخزن البيانات دي؟

بتتخزن جوه المتصفح في حاجة اسمها **Web Storage API**. وفي نوعين:

1. `localStorage` : دائم
2. `sessionStorage` : مؤقت (بيروح لما تقفلي التاب)

🟢 أهم الدوال اللي بنستخدمها:

الدالة	الاستخدام
`setItem(key, value)`	تخزين قيمة
`getItem(key)`	استرجاع قيمة
`removeItem(key)`	حذف قيمة
`clear()`	حذف كل البيانات
`length`	عدد العناصر

🧠 أمثلة عملية:

✅ 1. نخزن اسم المستخدم:

```
localStorage.setItem("username", "Ayaat");
```

✅ 2. نسترجع الاسم:

```
let name = localStorage.getItem("username");
console.log(name); // "Ayaat"
```

✅ 3. نحذف الاسم:

```
localStorage.removeItem("username");
```

✅ 4. نمسح كل البيانات:

```
localStorage.clear();
```

💡 مثال واقعي في مشروع:

🎯 فكرة: تطبيق ملاحظات (Notes App)

1. تخزين الملاحظات (كمصفوفة):

```
let notes = ["Do homework", "Go to gym", "Call mom"];
localStorage.setItem("myNotes", JSON.stringify(notes));
```

> استخدمنا `JSON.stringify` علشان نحول المصفوفة لنص

2. استرجاعها:

```
let savedNotes = JSON.parse(localStorage.getItem("myNotes"));
console.log(savedNotes);
// ["Do homework", "Go to gym", "Call mom"]
```

> استخدمنا `JSON.parse` علشان نحول النص لمصفوفة تاني

❗ملاحظات مهمة:

المعلومة	التوضيح
بتخزن نص بس	فلازم نحول الأوبجكت أو المصفوفة بـ `JSON.stringify()`
حجم التخزين محدود	تقريبًا 5MB
متاحة في كل المتصفحات	حديثة زي Chrome, Firefox...

🧠 متى نستخدم `localStorage` في المشاريع؟

الحالة	نستخدم `localStorage`؟
تطبيق بسيط مفيهوش تسجيل دخول؟	✅ أيوه
تخزين الوضع الليلي أو لغة الموقع؟	✅ أيوه
بيانات حساسة زي الباسورد؟	❌ لأ – استخدم سيرفر آمن
تخزين مؤقت زي سلة التسوق؟	✅ أيوه (لو عايز تحتفظ بيها بعد الإغلاق)

JavaScript Advance

✅ يعني إيه "Programming Paradigm"؟

البرادايم (Paradigm) في البرمجة هو:

> "الطريقة أو الأسلوب اللي بتكتب بيه الكود عشان تحل بيه مشكلة."

بمعنى أبسط:
هو **نمط التفكير** اللي بتبني عليه الكود بتاعك وتنظم بيه البرنامج.

🧠 مثال واقعي:

زي ما في طرق مختلفة لتنظيم البيت (مثلاً طريقة ترتيب المطبخ، أو الدولاب).
في البرمجة برضو، في طرق مختلفة لتنظيم الكود وحل المشاكل.

🔵 النوعين المشهورين اللي لازم تعرفهم كويس هما:

1. البرمجة الإجرائية (Procedural Programming)

> تركز على *"الأوامر والخطوات اللي بيعملها البرنامج خطوة خطوة."*

🧩 الخصائص:

- بنقسم الكود إلى **Functions**.
- بنستخدم **متغيرات** و **if** و **loops**.
- الكود بيتنفذ من فوق لتحت بالتسلسل.

🔧 أمثلة:

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

sayHello("Ayaat");
```

✅ لما بتدرس الأساسيات في JavaScript أو Python، فأنت بتتعلم البرمجة الإجرائية.

✅ Functional Programming Paradigm

البرمجة الدالية (أو الوظيفية)

> هي نمط من البرمجة بيهتم إن الكود يكون مبني على دوال (Functions) نقية، ومفيهوش تغيير في الحالة (No Side Effects)، والبيانات فيه ثابتة (Immutable).

📌 المفاهيم الأساسية داخل Functional Programming:

1. ✅ Pure Function (الدالة النقية)

> هي دالة ناتجها بيعتمد فقط على المُدخلات، ومش بتغيّر أي شيء خارجها.

📌 الخصائص:

- نفس المُدخلات ⟵ نفس الناتج دائمًا.
- مفيش **side effects** (يعني مش بتعدل متغيرات خارجها).

🧠 مثال:

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

2. ✅ Function Composition

> هي فكرة **دمج أكتر من دالة صغيرة مع بعض** علشان نعمل عملية أكبر.

🧠 مثال:

```
const double = x => x * 2;
const square = x => x * x;

const composed = x => square(double(x));
console.log(composed(3)); // (3 * 2)^2 = 36
```

3. ✅ Higher-Order Function (الدالة العليا)

> هي دالة:

- **بتاخد دالة كوسيط** (parameter)،
- أو **بترجع دالة** كنتيجة.

🧠 مثال:

```
function greet(name) {
  return `Hello, ${name}`;
}

function processUser(name, callback) {
  return callback(name);
}

console.log(processUser("Ayaat", greet)); // Hello, Ayaat
```

💡 أمثلة في JavaScript على Higher Order Functions:

الدالة	نوعها	الوصف
`map()`	higher-order	بتاخد دالة وتطبقها على كل عنصر
`filter()`	higher-order	بتاخد دالة شرط وتفلتر العناصر
`reduce()`	higher-order	بتجمع القيم بطريقة دالية
`setTimeout()`	higher-order	بتاخد دالة تتنفذ بعد وقت معين

2. البرمجة الكائنية (Object-Oriented Programming - OOP)

> تركز على *"تمثيل كل شيء ككائن (object) له خصائص وسلوكيات."*

🧩 الخصائص:

- بتستخدم **كائنات (Objects)** تمثل أشياء حقيقية.
- كل كائن بيحتوي على:
  - **خصائص (Properties)** زي الاسم أو السن ، البيانات.
  - **وظائف (Methods)** زي يتكلم أو يمشي ، الوظائف.

🧠 مثال:
تخيل كائن (Object) بيمثل "موظف":

```
let employee = {
  name: "Ayaat",
  age: 25,
  job: "Front-End Developer",
  work: function () {
    console.log("Ayaat is coding...");
  }
};

employee.work();  // "Ayaat is coding..."
```

🔶 `name` , `age` , `job` → دي خصائص (properties).
🔶 `work()` → دي سلوك أو method.

✳️ لماذا نستخدم OOP؟

1. **تنظيم الكود** بدل ما يكون فوضوي.
2. **إعادة الاستخدام (Reusability)** عن طريق الوراثة.
3. **التوسع بسهولة**.
4. **سهولة التعاون في فريق**.

🧱 أساسيات OOP:

المفهوم	المعنى بالعربي	التعريف البسيط
Class	الصنف	قالب أو وصف للكائن.
Object	كائن	نسخة من الكلاس.
Property	خاصية	بيانات بداخل الكائن.
Method	دالة	وظيفة يقوم بها الكائن.
Constructor	مُنشئ	دالة لإنشاء كائن جديد.
this	هذا الكائن	تشير للكائن الحالي.
Inheritance	الوراثة	كائن يرث خصائص من كائن آخر.
Encapsulation	التغليف	إخفاء تفاصيل الكود الداخلي.

المفهوم	المعنى بالعربي	التعريف البسيط
Polymorphism	تعدد الأشكال	نفس الدالة تعمل بطرق مختلفة.
Abstraction	التجريد	إظهار المهم وإخفاء التفاصيل.

🧑‍💻 مثال باستخدام Class في JavaScript :

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

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

let person1 = new Person("Ayaat", 22);
person1.greet();  // Hello, my name is Ayaat
```

🔶طريقتين لكتابة OOP في JavaScript

الطريقة القديمة (Traditional)	الطريقة الحديثة (Modern)
باستخدام الدوال العادية + prototype	باستخدام `class` و `constructor`
أقرب لطريقة JavaScript الكلاسيكية	أقرب لطريقة OOP في لغات تانية زي Java و Python
أقل وضوحًا	أوضح وأسهل في القراءة والفهم

🧐 الطريقة القديمة (Constructor Function + Prototype)

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function () {
  console.log(`Hi, I am ${this.name}`);
};

let person1 = new Person("Ayaat", 22);
```

```
person1.greet(); // Hi, I am Ayaat
```

- `function Person(...)` : دي اسمها **Constructor Function**.
- `this.name = name` : هنا بنحفظ الاسم اللي المستخدم هيبعته لما ينشئ الشخص.
- `this.age = age` : نفس الفكرة، بنحفظ العمر.

> ✅ الكلمة المفتاحية `this` هنا بتمثل **الكائن الجديد اللي هيتكون**.

🧠 إزاي أستخدم الدالة دي لإنشاء كائن جديد؟

let person1 = new Person("Ayaat", 22);

🔍 اللي بيحصل هنا:

- الكود بينشئ كائن جديد فارغ.
- بيربط الكائن ده بالدالة `Person` .
- بيحط `"name = "Ayaat` و `age = 22` جواه.
- النتيجة: كائن كامل جاهز!

🎭 إضافة سلوك للكائن باستخدام Prototype

```
Person.prototype.greet = function () {
  console.log(`Hi, I am ${this.name}`);
};
```

📌 ليه استخدمنا `prototype` ؟

- لأنك **مش عايز تكرر** دالة `greet()` في كل كائن جديد.
- بدل ما كل شخص يكون عنده نسخة مستقلة من `greet()` ، نحطها مرة واحدة في الـ prototype.

> ✅ الكائنات بتقدر "تورّث" من `prototype` وتستخدم الدوال دي كأنها جزء منهم.

🙋 دلوقتي نده الدالة:

```
person1.greet(); // Hi, I am Ayaat
```

- `person1` بيروح يدور على `greet` جواه.
- مش بيلاقيها؟ يروح يدور في `Person.prototype` ويلاقيها هناك!
- ينفذها عادي.

🧠 كأنك بتقول:

> "كل الناس اللي اتعملوا بوصفة `Person` ، هيكونوا عارفين يقولوا اساميهم لما يتنده عليهم".

🧑‍💻 الطريقة الحديثة (class syntax - ES6)

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

  greet() {
    console.log(`Hi, I am ${this.name}`);
  }
}

let person1 = new Person("Ayaat", 22);
person1.greet(); // Hi, I am Ayaat
```

📌 شرح السطر بسطر

السطر	المعنى
`class Person`	تعريف كلاس اسمه Person
`constructor(name, age)`	ده زي `function Person(name, age)` في الطريقة القديمة – هو اللي بينشئ الكائن
`this.name = name`	تخزين القيمة داخل الكائن
`greet()`	طريقة (method) للكائن، بتضاف تلقائيًا على prototype
`new Person("Ayaat", 22)`	بننشئ كائن جديد بنفس الطريقة

✅ مقارنة مع الطريقة القديمة

العنصر	الطريقة القديمة	الطريقة الحديثة (`class`)
تعريف الكائن	`function Person(...) {}`	`class Person { constructor(...) {} }`
إضافة method على prototype	`Person.prototype.greet = function()`	داخل الكلاس `greet() { ... }` مباشرة
إنشاء كائن جديد	`new Person(...)`	نفس الشيء: `new Person(...)`

🧠 ملاحظات مهمة:

1. **الفرق الأساسي في الشكل فقط (syntax).**
2. الكود اللي بتكتبه بكلمة `class` بيتحوّل داخليًا لنفس الكود اللي كنا بنكتبه بالطريقة القديمة (constructor + prototype).
3. لكن `class` بيخلي الكود **أوضح وأسهل** في المشاريع الكبيرة.

💡 ملاحظة مهمة من MDN:

حتى لو الطريقة القديمة شغالة، استخدم الـ **syntax** `class` في أي كود جديد، لأن:

- أوضح في المعنى.
- أقل في الأخطاء.
- أقرب لمفاهيم OOP التقليدية.

🧠 سؤال شائع في المقابلة:

هل `class` في JavaScript حقيقي زي Java؟

🔶 الإجابة:
لأ، JavaScript تحت الغطاء ما زالت تستخدم `prototype-based` inheritance، بس syntax `class` هي طريقة حديثة فقط لكتابة نفس الفكرة بشكل أوضح.

this , bind

✅ أولاً: ما معنى `this` ؟

📌 ال `this` تعني: الكائن (object) الحالي اللي بننفذ عليه الكود.

لكن **قيمتها بتتغير حسب المكان اللي `this` اتكتبت فيه.**

✅ أمثلة على `this`

1. داخل object method:

```
const person = {
  name: "Ayaat",
  greet() {
    console.log(`Hello, I am ${this.name}`);
  }
};

person.greet();  // Hello, I am Ayaat
```

🟢 هنا `this` تشير إلى `person`.

2. خارج object (في الدالة العادية):

```
function show() {
  console.log(this);
}

show();  // في المتصفح: window (أو undefined في strict mode)
```

🔴 هنا `this` تشير إلى الكائن العام `window`.

3. داخل `arrow function`:

```
const person = {
  name: "Ayaat",
  greet: () => {
    console.log(this.name);
  }
};

person.greet();  // undefined ❌
```

❗السبب: `arrow function` **لا تمتلك this خاص بيها**
=> تاخد `this` من المكان اللي اتعرفت فيه (غالباً window أو undefined).

✅ ثانيًا: `bind()` – يعني إيه؟

📌 ال `bind()` هي method بتستخدم لتثبيت قيمة `this` داخل دالة.

✅ مثال:

```
const person = {
  name: "Ayaat",
};

function greet() {
  console.log(`Hello, I am ${this.name}`);
}

const greetAyaat = greet.bind(person);

greetAyaat();  // Hello, I am Ayaat ✅
```

🔶 استخدمنا `bind(person)` عشان نربط `this` بالدالة على كائن `person`.

✅ مثال واقعي من زرار في الصفحة:

```
const button = document.querySelector("button");

const user = {
  name: "Ayaat",
  handleClick() {
    console.log(`Button clicked by ${this.name}`);
  }
};

button.addEventListener("click", user.handleClick.bind(user));
```

❗من غير `this.name` ، `bind` هيكون `undefined` لأن `this` هتشير للزر نفسه مش `user` .

✅ Synchronous vs Asynchronous في JavaScript

أولاً: 🔶 يعني إيه Synchronous (متزامن)؟

- ال **JavaScript تنفذ الأوامر سطر بسطر**.
- كل سطر لازم يخلص قبل ما تروح للسطر اللي بعده.
- يعني: **البرنامج بيستنى كل خطوة تخلص قبل ما يكمل**.

🔷 مثال:

```
console.log("1");
console.log("2");
console.log("3");
// 1
// 2
// 3
```

ثانيًا: 🔶 يعني إيه Asynchronous (غير متزامن)؟

- بعض العمليات بتاخد وقت (زي: طلب API، قراءة ملف، انتظار user يضغط زر).
- بدل ما توقف الكود، JavaScript بتكمل تنفذ باقي الأوامر.
- ولما تخلص العملية الطويلة، **ترد النتيجة في وقت لاحق**.

🔷 مثال:

```
console.log("Start");
```

```
setTimeout(() => {
  console.log("Timeout finished");
}, 2000);

console.log("End");

// Start
// End
// Timeout finished
```

❗لاحظ إن `"Timeout finished"` ظهرت **بعد تأخير**، لأن `setTimeout` بتشتغل بشكل **Asynchronous**.

💬 مثال واقعي:

تخيل بتطبخ أكلة:

- ال**Synchronous** = لازم تخلص كل حاجة واحدة واحدة. تستني البوتاجاز يخلص عشان تبدئي تقطع السلطة.
- ال**Asynchronous** = تشغل البوتاجاز، وفي نفس الوقت تقطع السلطة، وكل حاجة تمشي مع بعض.

📌 في JavaScript نستخدم أدوات للـ Async:

- `setTimeout`, `setInterval`
- `fetch()` – جلب بيانات من API
- `Promises`
- `async/await`

🧠 أولاً: JavaScript Language Nature

✅ JavaScript is a **Single-Threaded Language**

يعني:

- الجافاسكرييت بتنفذ **تعليمة واحدة بس في نفس اللحظة**.
- زي طابور: تنفذ أمر، تخلصه، بعدين تروح للي بعده.

❗ يعني **مافيش تنفيذ أوامر في نفس الوقت**.

> 🔄 بس إزاي تعمل حاجات زي الانتظار أو التحميل بدون توقف؟ (هنشوف لاحقًا مع الـ Web APIs والـ Event Loop)

🔄 الفرق بين Single-threaded و Multi-threaded

	Single-threaded(JavaScript)	Multi-threaded (C++, Java…)
التنفيذ	أمر واحد في كل لحظة	أكثر من أمر في نفس الوقت
الأداء	أبسط، لكن ممكن يبطأ لو فيه تأخير	أسرع لو تم إدارة الـ threads صح

	Single-threaded(JavaScript)	Multi-threaded (C++, Java…)
الصعوبة	سهل لأن مافيش تداخل	أصعب لإدارة التداخل بين الـ threads

🧱 Data Structures: Stack vs Queue

🔷 Call Stack (المكدس)

- **Stack = LIFO (Last In, First Out)** يعني آخر حاجة دخلت هي أول حاجة تطلع.
  - الجافاسكريبت تستخدمه لتتبع **مكانك في الكود**.

مثال:

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

function greet() {
  sayHi();
}

greet();
```

- تدخل `()greet` ← تدخل `()sayHi` ← تطبع "Hi" ← تطلع `()sayHi` ← تطلع `()greet`.

🎯 زي لما تدخل غرفة جوه غرفة وتطلع عكس الترتيب اللي دخلته.

🔶 Callback Queue (الطابور)

- **Queue = FIFO (First In, First Out)**: أول حاجة تدخل هي أول حاجة تطلع.
  - لما تعمل حاجة async (زي `setTimeout`) تتحط في الطابور بعد انتهاء وقتها.

🌐 Web APIs (الجزء من المتصفح)

ال JavaScript **نفسها مش بتعرف تتعامل مع الانتظار أو الـ timers** أو **HTTP requests**.

❗ المتصفح (Chrome مثلًا) بيوفر Web APIs مثل:

- `setTimeout`
- `fetch`
- `DOM Events` (click, keydown…)

💡 الجافاسكريبت تبعت المهمة للـ Web API، ولما تخلص، ترجع النتيجة في الـ **Callback Queue**.

🔁 Event Loop (المايسترو!)

- شغلته: **يراقب** الـ Call Stack و Callback Queue.
- لو الـ Call Stack فاضي، **يسحب مهمة من الـ Callback Queue ويحطها على الـ Stack**.

🎯 هو السبب إن الكود asynchronous في الجافاسكريبت بيشتغل بشكل صحيح.

🧠 خلينا نوضح كل ده بمثال:

```
console.log("Start");

setTimeout(() => {
  console.log("Inside Timeout");
}, 2000);

console.log("End");
```

✅ إيه اللي بيحصل:

1. `console.log("Start")` → يدخل Stack → يخرج → يطبع.
2. `setTimeout(...)` → يروح للـ Web API → يبدأ يعد 2 ثواني.
3. `console.log("End")` → يدخل Stack → يخرج → يطبع.
4. بعد 2 ثواني → Web API تحط الـ callback في **Callback Queue**.
5. **Event Loop** يلاحظ إن الـ Stack فاضي → يدخل callback من الـ Queue → يطبع "Inside Timeout".

📌 ملخص بصيغة واقعية:

> تخيل عندك مطبخ (JavaScript Engine):

- **الشيف** = Call Stack، بيطبخ وجبة واحدة في المرة.
- **المساعدين (Web APIs)** بيشتغلوا في الخلفية (يسخنوا، يجهزوا حاجات).
- **الجرسون (Event Loop)**، ينقل الطبخة الجاهزة للشيف لما يكون فاضي.
- **الطلبات الجاهزة (Callback Queue)** تستنى دورها لحد ما الشيف يكون جاهز.

🎯 ليه ده مهم؟ عشان:

- لما تفهم ده، تقدر تكتب كود async ذكي بدون مشاكل.
- هتفهم إمتى تستخدم `setTimeout`, `fetch`, `async/await` بثقة.
- هتفهم إنك متخليش عمليات طويلة تعلق الكود.

✅ ما هو الـ Promise؟ (ببساطة)

الـ `Promise` هو **وعد** إن الكود هيكمل بعدين — إما بنجاح ✅ أو بفشل ❌.

❓ليه نستخدم Promise؟

لأن العمليات مثل:

- تحميل من الإنترنت (fetch),
- انتظار وقت (setTimeout),
- التعامل مع قواعد بيانات،

كلها **لا تحدث فورًا**، فلازم نعرف:

- هتنجح؟ نكمل ✅.
- فشلت؟ نتصرف ❌.

🧠 تخيلي واقعي:

أنت طلبت كتاب من متجر أونلاين (زي أمازون):

- المتجر يقولك: **وعد، هيوصل في 3 أيام.**
- يوم ما يوصل → **تستلمي الكتاب** (resolve).
- لو ضاع أو حصلت مشكلة → **يقولك فشلنا** (reject).

✍️ Syntax

```
let promise = new Promise(function(resolve, reject) {
  // Async operation هنا
  if (كل شيء تمام) {
    resolve("نجحنا ✅");
  } else {
    reject("في مشكلة ❌");
  }
});
```

🧪 مثال واقعي:

```
let bookOrder = new Promise((resolve, reject) => {
  let delivered = true;

  if (delivered) {
    resolve("Your book has arrived 📦");
  } else {
    reject("Delivery failed ❌");
  }
});
```

```
bookOrder
  .then((message) => {
    console.log("Success: " + message);
  })
  .catch((error) => {
    console.log("Error: " + error);
  });
```

🔗 then / catch / finally

Method	الوظيفة
`.then()`	ينفذ لو الـ Promise نجح ✅
`.catch()`	ينفذ لو الـ Promise فشل ❌
`.finally()`	ينفذ دائمًا سواء نجح أو فشل

⏳ مثال مع وقت:

```
let loadingData = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Data loaded successfully!");
  }, 3000);
});

loadingData
  .then((data) => console.log(data))
  .catch((err) => console.log(err))
  .finally(() => console.log("Operation completed"));
```

💡 Notes مهمة:

1. ال **Promise تبدأ تنفذ على طول** بمجرد إنك تكتب `new Promise(...)`
2. تقدر تبعت `Promise` لدالة أو تستخدمها مع `fetch` وغيره.
3. الـ `Promise` **لا تحل مشكلة الـ Callback Hell** وحدها، لكنها **أسهل في القراءة**.

🔁 الفرق بين Promise و Callback:

Promise	Callback
شكل أنظف وسهل القراءة	callback hell وممكن تدخل في nested
فيها `.catch()` تلقائي للتعامل مع الأخطاء	لازم تتأكد من `error` بنفسك

Callback	Promise
مش مرنة مع async operations	ممتازة للـ async والـ chaining

🔁 تمرين بسيط:

```
function isEvenNumber(num) {
  return new Promise((resolve, reject) => {
    if (num % 2 === 0) {
      resolve(`${num} is even ✅`);
    } else {
      reject(`${num} is not even ❌`);
    }
  });
}

isEvenNumber(4)
  .then((result) => console.log(result))
  .catch((err) => console.log(err));
```

💡 ليه نستخدم Promise methods؟

لو عندي كذا Promise (مثلاً: تحميل بيانات من 3 APIs)، ساعات:

- ✅ عايز أستنى الكل ينجح.
- ⏱️ عايز أول واحد يرد.
- ❌ عايز أتأكد حتى لو بعضهم فشل.
- ☂️ عايز أتجاهل اللي فشل وأكتفي بأي واحد نجح.

كل ده نعمله من خلال:

Method	معناها بالعربي
`Promise.all`	استنى **كل** الـ promises تنجح ✅ وإلا ترفض ❌
`Promise.any`	يكفي **واحد ينجح** ✅ والباقي أتجاهله
`Promise.allSettled`	استنى الكل يخلصوا، سواء نجحوا أو فشلوا 🟡
`Promise.race`	أول Promise تخلص (نجاح أو فشل) ⏱️

🧪 1. `Promise.all()`

ينفع لو عندي 3 عمليات ولازم التلاتة ينجحوا عشان أكمل.

```
const p1 = Promise.resolve("Data 1");
const p2 = Promise.resolve("Data 2");
const p3 = Promise.resolve("Data 3");

Promise.all([p1, p2, p3])
  .then((results) => console.log("✅ All succeeded:", results))
  .catch((err) => console.log("❌ One failed:", err));
```

❗ لو واحدة بس فشلت ⟵ يدخل في `catch` .

🧪 2. `Promise.any()`

يكفي أول **واحدة تنجح** والباقي يتجاهله.

```
const p1 = Promise.reject("❌");
const p2 = Promise.resolve("✅ I win");
const p3 = Promise.resolve("✅ I'm ignored");

Promise.any([p1, p2, p3])
  .then((result) => console.log("🎉 First success:", result))
  .catch((err) => console.log("😢 All failed:", err));
```

❗ لو **كلهم فشلوا** ⟵ يدخل في `catch` .

🧪 3. `Promise.allSettled()`

ينتظر **كل الـ promises** سواء نجحوا أو فشلوا — ويديلي **حالتهم كلها**.

```
const p1 = Promise.resolve("✅ Done");
const p2 = Promise.reject("❌ Error");
const p3 = Promise.resolve("✅ Finished");

Promise.allSettled([p1, p2, p3])
  .then((results) => {
    results.forEach((result) => {
      console.log(result.status, result.value || result.reason);
    });
  });
```

النتيجة:

```
fulfilled ✅ Done
rejected ❌ Error
```

```
fulfilled ✅ Finished
```

🧪 4. `Promise.race()`

ينفذ أول **Promise تخلص** سواء نجحت أو فشلت.

```
const p1 = new Promise((res) => setTimeout(() => res("🐢 Slow"), 3000));
const p2 = new Promise((res) => setTimeout(() => res("🚀 Fast"), 1000));

Promise.race([p1, p2])
  .then((result) => console.log("🏁 First to finish:", result))
  .catch((err) => console.log("❌ Failed first:", err));
```

🛠️ استخدامات واقعية:

- ال `Promise.all` ← لما أجيب بيانات من كذا API وكلهم لازم يوصلوا قبل ما أكمل.
- ال `Promise.any` ← لو عايز أول نتيجة من أكتر من سيرفر.
- ال `Promise.allSettled` ← لما يهمني أعرف النتيجة حتى لو بعضهم فشل.
- ال `Promise.race` ← لو عايز أسرع رد فعل — أول سيرفر يرد.

✅ يعني إيه `async` و `await` ؟

الكلمة	معناها
`async`	بتحول الدالة (function) لـ دالة **غير متزامنة**، يعني بترجع `Promise` تلقائيًا.
`await`	بنستخدمها **جوه دالة async** عشان نستنى تنفيذ Promise قبل ما نكمل السطر اللي بعده.

🧠 ليه أستخدم `async/await` ؟

لأن:

- الكود بيكون شبه الكود العادي (synchronous) في الشكل، لكنه فعليًا غير متزامن.
- بنبعد عن تداخل `.then().then()` اللي بيصعب القراءة.
- أسهل في استخدام `try...catch` مع الأخطاء.

🔧 مثال بسيط:

```
function getUserData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("User: Ayaat");
    }, 2000);
```

```
  });
}

async function showData() {
  console.log("⏳ Loading...");
  const result = await getUserData();
  console.log("✅ Done:", result);
}

showData();
```

النتيجة:

```
⏳ Loading...
(بعد ثانيتين)
✅ Done: User: Ayaat
```

🧪 مثال فيه أكثر من await:

```
function getName() {
  return new Promise((resolve) => setTimeout(() => resolve("Ayaat"), 1000));
}
function getAge() {
  return new Promise((resolve) => setTimeout(() => resolve(22), 1000));
}

async function getInfo() {
  const name = await getName();
  const age = await getAge();
  console.log(`👋 Hello, I am ${name} and I am ${age} years old.`);
}

getInfo();
```

⚠️ Error Handling باستخدام `try...catch`

```
function fetchData() {
  return new Promise((_, reject) => {
    setTimeout(() => reject("❌ Failed to fetch data"), 1000);
  });
}
```

```
async function showInfo() {
  try {
    const data = await fetchData();
    console.log(data);
  } catch (err) {
    console.log("🚨 Error:", err);
  }
}

showInfo();
```

💡 ملاحظات مهمة:

1. لازم تستخدم `await` **جوه async function فقط**.
2. ال `async function` دايمًا بترجع `Promise` حتى لو رجعت قيمة عادية.

```
async function sayHi() {
  return "Hi Ayaat!";
}

sayHi().then(console.log); // Hi Ayaat!
```

📌 مقارنة بين async/await و .then()

```
// باستخدام then
fetch('https://api.com/data')
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));

// باستخدام async/await
async function getData() {
  try {
    const res = await fetch('https://api.com/data');
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.log(err);
  }
}
```

🧠 استخدامات واقعية:

- جلب بيانات من API.
- تنفيذ عمليات متتابعة بينهم وقت انتظار.
- استخدام `try...catch` لمعالجة الأخطاء بسلاسة

async-await-HOF

🧠 أولاً: مراجعة سريعة

✅ 1. ما هو `async/await` ؟

- ال `async` = بتحول الدالة لدالة بترجع Promise تلقائيًا.
- ال `await` = بتخلي الكود ينتظر تنفيذ promise قبل ما يكمل.

✅ 2. ما هو Higher Order Function (HOF)؟

- أي دالة بتاخد **دالة تانية كوسيلة** أو بترجع دالة.
- زي: `addEventListener` , `setTimeout` , `reduce` , `forEach` , `filter` , `map` , إلخ.

✨ تطبيق حقيقي: دمج async/await مع HOF

💡 **السيناريو: عندنا Array من المستخدمين، بنجيب بيانات كل مستخدم من API (محاكاة باستخدام promise)، وعايزين نعرض النتيجة.**

```
const users = ["Ayaat", "Tarek", "Suzan"];

function fetchUserData(username) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({ name: username, age: Math.floor(Math.random() * 30) + 20 });
    }, 1000);
  });
}
```

✅ نستخدم `async/await` + `map` (لكن ناخد بالنا بقى!)

```
async function processUsers(users) {
  const userPromises = users.map(async (user) => {
    const data = await fetchUserData(user);
    return `${data.name} is ${data.age} years old.`;
  });
```

```
  const results = await Promise.all(userPromises);
  console.log(results);
}

processUsers(users);
```

🔍 شرح:

السطر	المعنى
`map(async user => ...)`	استخدمنا HOF (map) لكن كتبنا جوّاها دالة async
`await fetchUserData(user)`	بنجيب بيانات كل مستخدم
`Promise.all([...])`	نستنى كل الـ promises اللي طلعوا من map

❌ تحذير هام:

لو استخدمت `forEach` مع `async/await` مش هيشتغل زي ما تتوقع!

```
// ❌ لا يُنصح باستخدام forEach داخل async
users.forEach(async (user) => {
  const data = await fetchUserData(user);
  console.log(data); // هيتطبع لكن مش ممكن تنتظرهم كلهم مرة واحدة
});
```

✅ استخدم بدلها `for...of` أو `map + Promise.all`

✅ أولًا: ما هو `fetch` ؟

ال `fetch()` هي دالة مدمجة في JavaScript تُستخدم لجلب البيانات من مصدر خارجي (زي API).
بترجع **Promise**، وده معناه إنك لازم تستخدم معاها:

- `.then()` و `.catch()`
  أو
- `async/await`

✅ شكل الطلب الأساسي

```
fetch('https://api.example.com/data')
  .then(response => response.json())  // تحويل الرد إلى JSON
  .then(data => console.log(data))     // استخدام البيانات
  .catch(error => console.error('Error:', error));  // التعامل مع الخطأ
```

✅ باستخدام async/await

```
async function getData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}
```

✅ مثال عملي بجد (نستخدم JSON API حقيقي)

📌 نستخدم موقع مجاني اسمه JSONPlaceholder

رابط الـ API:

`https://jsonplaceholder.typicode.com/users`

✅ مثال:

```
async function fetchUsers() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users");
    const users = await response.json();
    console.log("Users:", users);
  } catch (error) {
    console.log("Failed to fetch users:", error);
  }
}

fetchUsers();
```

💡 طيب إزاي أستخدم البيانات دي؟

مثال: نطبع أسماء المستخدمين في صفحة HTML

```
async function showUserNames() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  const data = await res.json();

  data.forEach(user => {
    const p = document.createElement("p");
```

```
    p.textContent = `User: ${user.name}`;
    document.body.appendChild(p);
  });
}

showUserNames();
```

❗ ملحوظة مهمة:

لو الـ API رجع حالة خطأ (زي 404 أو 500)، لازم تفحص `response.ok` :

```
const res = await fetch(url);
if (!res.ok) {
  throw new Error(`HTTP Error: ${res.status}`);
}
```

✅ ما هو الـ Module؟

ال**Module** هو ملف JavaScript مستقل يحتوي على كود يمكن إعادة استخدامه في ملفات أخرى.

> يعني بدل ما تكتبي كل الكود في ملف واحد كبير، بتقسميه لأجزاء (Modules) صغيرة ومنظمة.

✅ ليه بنستخدم Modules؟

1. **تنظيم الكود**: مايبقاش ملف واحد فيه كل شيء.
2. **إعادة الاستخدام**: تكتب دالة مرة وتستخدمها في أكتر من مكان.
3. **تجنب التعارض**: المتغيرات والدوال داخل كل Module مش بتأثر على غيره.
4. **سهولة الاختبار والتطوير**.

✅ أنواع Modules:

1. ES6 Modules (الأحدث - Modern JS) ✅

- نستخدم `export` و `import` .
- مدعومة في المتصفحات الحديثة.
- الأفضل مع React و المشاريع الكبيرة.

2. CommonJS Modules (قديمة - تستخدم في Node.js)

- نستخدم `require()` و `module.exports`
- مش محتاج دلوقتي لو بتشتغل على المتصفح فقط.

✅ إزاي نكتب ES6 Module؟

📦 1. Export (تصدير)

✅ تصدير شيء واحد (Default Export):

```
// file: greet.js
export default function greet(name) {
  console.log(`Hello ${name}`);
}
```

✅ تصدير أكثر من شيء (Named Export):

```
// file: math.js
export const add = (a, b) => a + b;
export const multiply = (a, b) => a * b;
```

📦 2. Import (استيراد)

✅ لاستيراد الـ default:

```
// file: main.js
import greet from './greet.js';
greet("Ayaat");
```

✅ لاستيراد الـ named:

```
// file: main.js
import { add, multiply } from './math.js';

console.log(add(2, 3));        // 5
console.log(multiply(2, 3));   // 6
```

✅ ممكن تغير اسمهم:

```
import { add as sum } from './math.js';
console.log(sum(5, 6)); // 11
```

✅ ملاحظات مهمة:

- لازم ملفات الـ Modules تنتهي بـ `.js` أو `.mjs`
- لازم تضيف `"type="module` في HTML:

```
<script type="module" src="main.js"></script>
```

مش هينفع تستخدم `import` و `export` في ملف عادي بدون ما تحدد إنه Module.

🧠 ما معنى Bundler؟

> ال **Bundler** هو أداة بتجمع كل ملفات المشروع (JS, CSS, Images...) في ملف أو مجموعة ملفات منظمة وجاهزة للاستخدام في المتصفح.

🟡 ليه بنستخدم Bundlers؟

في المشاريع البسيطة، ممكن نشتغل بكود HTML وCSS وJS في ملفات منفصلة عادي.
لكن مع المشاريع الكبيرة بيحصل الآتي:

- بيكون عندك عشرات أو مئات الملفات (modules)
- كل ملف عنده dependencies (يعتمد على ملفات تانية)
- فيه أنواع ملفات مختلفة (JS - CSS - HTML - صور - Fonts)
- عايز تعمل **Optimization** (تصغير الملفات - حذف الكود الزائد - ضغط الصور)
- عايز تكتب كود حديث (ES6) والمتصفح مش بيدعمه بالكامل

🔧 عشان كده بنحتاج **Bundler** يعمل الآتي:

وظيفة	شرح
Bundling	جمع كل الملفات في ملف واحد أو أكثر
Minification	تقليل حجم الملفات بحذف الفراغات والكومنتات
Transpiling	تحويل كود حديث (مثل ES6) إلى كود متوافق مع المتصفحات القديمة
Code Splitting	تقسيم الكود لأجزاء تُحمّل عند الحاجة
Live Reload	تحديث تلقائي عند حفظ الكود أثناء التطوير

🔵 أشهر أدوات الـ Bundlers

اسم الباندلر	مميزاته
Webpack	الأكثر استخدامًا - قوي جدًا - مرن
Parcel	سهل الاستخدام - لا يحتاج إعدادات كثيرة
Vite	سريع جدًا - حديث - مفضل في React و Vue
Rollup	ممتاز لمكتبات JavaScript - خفيف وسريع

✅ مثال واقعي

لو عندك مشروع React فيه عشرات المكونات (components) وكل مكون مستورد من ملف مختلف، إزاي هتشتغل بيهم بدون bundler؟

- لو استخدمت HTML وJS عادي، هتحتاج تضيفي 50 `<script>` في الـHTML.
- وده بيبقى بطئ، ومش عملي.
- الباندلر بياخد كل ده ويطلعلك `bundle.js` فيه كل شيء منظم.

🔧 إزاي نستخدم Bundler (مثل Parcel مثلًا):

1. تهيئة المشروع:

```
npm init -y
npm install parcel --save-dev
```

في النهاية، رحلة التعلُّم والمعرفة لم تنتهِ بعد...
بل هذه مجرد بداية لطريق طويل مليء بالاكتشافات والتحديات والنجاحات.
كل سطر تعلمته، وكل مفهومة فهمته، هو لبنة جديدة في بناء مستقبلك التقني.
استمر في التعلم، لا تخاف من الخطأ، ولا تتوقف عن السؤال،
فالعقل الذي يسعى للفهم لا يشيخ أبدًا.
وتذكر دائمًا:
"العلم نور، والعمل به نجاح، والاستمرار عليه هو طريق التميز."

🌟 بالتوفيق دائمًا في رحلتك البرمجية 💻💪
— آيات