JavaScript Paradigms - FP

ال Paradigms هو أسلوب لكتابة الكود (Style of code) لتحقيق أعلى كفاءة ممكنة.

يوجد نوعين من ال Paradigms وهما:

1- ال Functional programming:

حجر البناء (Building blocks) لهذا النوع هو الدوال (Functions). هذه الدوال (Functions) تنقسم إلى ثلاثة أنواع هما ال Pure Functions و Composition Functions و Higher order functions.

Pure Functions:

تُعتبر Pure Functions البنية الأساسية للبرمجة الوظيفية (Functional Programming).

تتميز Pure Functions بأنها لا تتأثر بأي متغيرات أو حالات خارجية، وإذا تم استدعاؤها بنفس المدخلات ستعيد نفس النتيجة دائمًا.

مثال:

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

Composition Functions:

هذه الدوال تُستخدم لبناء دوال أكبر من خلال إضافة دوال أخرى بداخلها.

تُستخدم هذه الدوال لتنظيم الأكواد وجعلها أكثر قابلية للصيانة وإعادة الاستخدام.

مثال:

```
const compose = (f, g) => x => f(g(x));
```

Higher Order Functions:

هذه الدوال تُستخدم للتعامل مع الدوال كمعاملات أخرى أو لإرجاع دوال أخرى.

هذه الدوال تُمكن المطورين من تجميع الدوال واستخدامها بشكل أقوى وأكثر تنظيمًا. يُمكن استخدامها لإنشاء دوال عالية المستوى.

مثال:

```
const map = (fn, array) => array.map(fn);
```

2- ال Object oriented programming

Javascript Paradigms - OOP

ال Object oriented programming:

حجر البناء (Building blocks) لهذا النوع هو Objects.

البرمجة الشيئية (OOP) هي إحدى الأنماط البرمجية التي تستخدم في لغة البرمجة جافا سكريبت (JavaScript) بشكل واسع. هذا النمط يعتمد على مفهوم الكائنات والتفاعلات بينها. الكائنات هي مجموعات من البيانات والأساليب التي تعمل على هذه البيانات.

دالة "Date" هي واحدة من الدوال المنشئة المدمجة في JavaScript التي تُستخدم لإنشاء كائن يمثل التاريخ والوقت الحالي.

```
var currentDate = new Date();

console.log(currentDate);
```

في هذا المثال، تم استخدام "new Date()" لإنشاء كائن تاريخ جديد، والذي سيحتوي على تاريخ ووقت اللحظة الحالية. بعد ذلك تم طباعة هذا الكائن باستخدام "console.log()"، وسيتم عرض التاريخ والوقت الحالي في الوحدة الزمنية المحلية.

يمكنك استخدام الـ "Prototypes" في البرمجة الشيئية في JavaScript. الـ "prototypes" تسمح لك باستخدام خصائص وأساليب موجودة بالفعل.

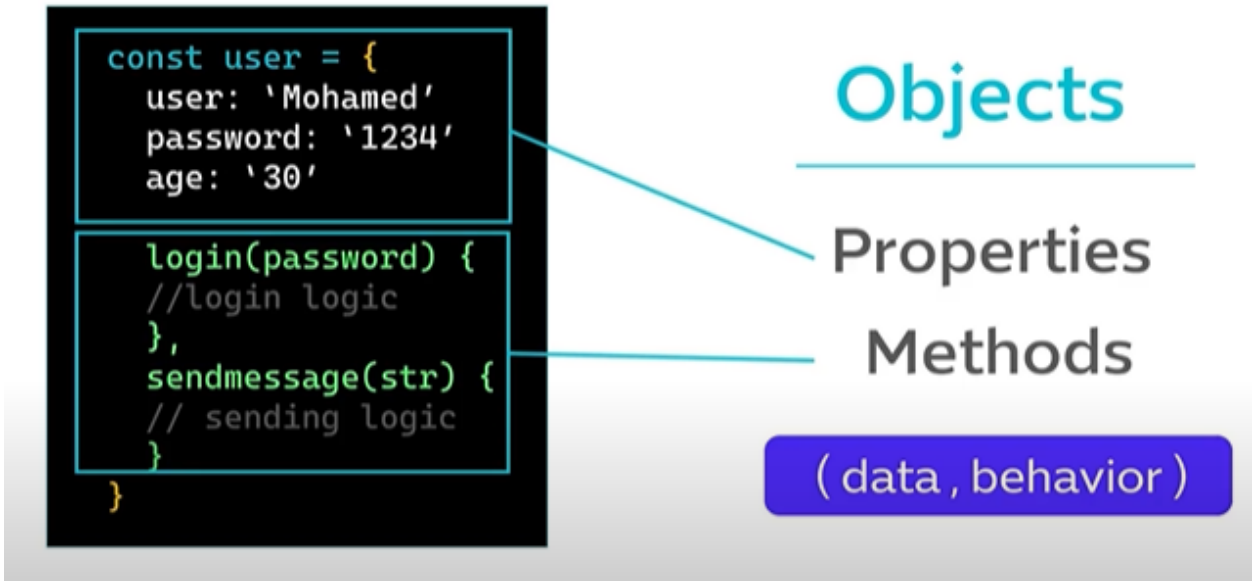
مثال:

```
Date.prototype.getFormattedDate = function() {
  var year = this.getFullYear();
  var month = this.getMonth() + 1;
  var day = this.getDate();
  return year + "-" + (month < 10 ? "0" : "") + month + "-" + (day < 10 ?
"0" : "") + day;
};
var currentDate = new Date();
var formattedDate = currentDate.getFormattedDate();
console.log(formattedDate);
```

OOP Intro

حجر البناء (Building blocks) لل OOP هو Objects.

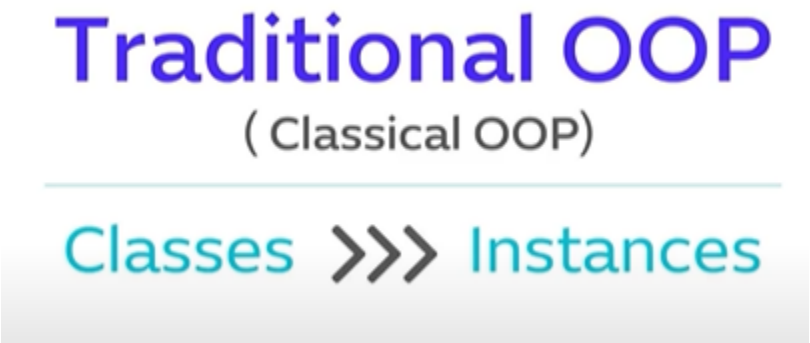
الكائن (Object) يتكون من Properties و Methods.



تذكر أن ال Paradigms تساعد في تجنب ال Spaghetti code وهو عبارة عن كود غير منظم ومتداخل.

ال OOP تعتمد علي Class لعمل الـ Object.

في ال Traditional OOP يكون لدينا Classes والتي نقوم بعمل Instances منها.



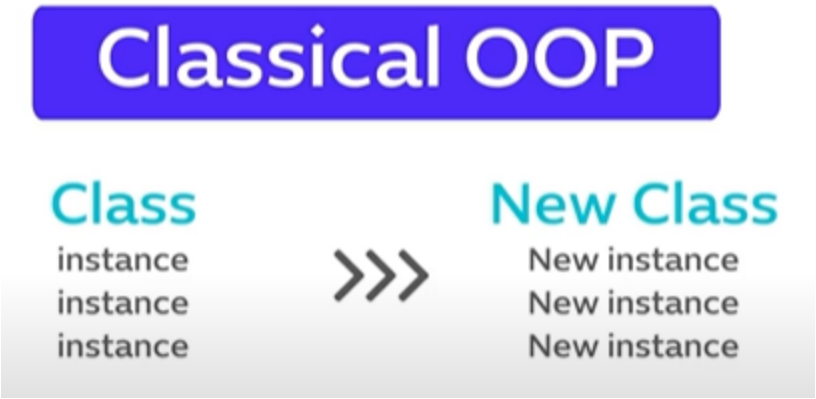
في الـ JavaScript OOP يكون لدينا Prototypes والتي نقوم بانشاء Objects منها،

ويكون ال Object مربوط ب Prototype خاص به يرث (Inherits) منه Properties

و Methods.

لماذا يختلف ال Traditional OOP عن الـ JavaScript OOP؟

في ال Traditional OOP ال Inheritance يحدث بين Two Classes.



في الـ JavaScript OOP ال Inheritance يحدث بين Instance و ال Prototype.

إزاي نعمل implement للـ OOP في الـ JavaScript ؟

Es6 classes

- تم إطلاقها فى الابديت الخاص بـEs6
- لا تعتبر Real classes
- يطلق عليها Synthetic sugar

Constractor Functions

- من أقدم الطرق في التطبيق

Construction Functions

هي دالة تُستخدم لتطبيق مبدأ ال OOP. تُستخدم لانتاج كائن (Object). يتم استدعاؤها باستخدام كلمة new.

نستخدم كلمة this بداخل الـ Construction function لتخزين البيانات الخاصة بال Object.

مثال:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
var person1 = new Person("Ali", 30);
var person2 = new Person("Mohamed", 25);
console.log(person1.name);
console.log(person2.age);
```

يُمكنك اضافة Method بداخل الـ Construction function كالتالي:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.sayHello = function() {
    console.log("Hello, my name is " + this.name);
  };
}
```

ولكن هذه ليست الطريقة الأفضل لعمل هذا لأنه عند وضع ال Method داخل Construction function سيتم إنشاء نسخة من هذه ال Method لكل كائن جديد يتم إنشاؤه باستخدام الـ new operator. وهذا يعني أن كل كائن سيحتوي على نسخة من Method وسيتم استهلاك مساحة ذاكرة إضافية لل Method بالنسبة لكل كائن.

ولذلك يوجد طريقة أخرى لعمل هذا وهي:

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

```
  this.age = age;
}
Person.prototype.sayHello = function() {
  console.log("Hello, my name is " + this.name);
};
```

عند إضافة ال Method باستخدام الـ Prototype، فإن هذه الأساليب تشترك بين جميع الكائنات التي تم إنشاؤها باستخدام نفس الـ Construction function. هذا يعني أنه لا يتم إنشاء نسخ من Method لكل كائن، بل يتم مشاركة Method بين كافة الكائنات.

في لغة الجافا سكريبت"this" و "bind" هما من المفاهيم المهمة التي تتعلق بكيفية التعامل مع الكائنات (Objects) والدوال (Functions).

this

هي كلمة مفتاحية في JavaScript تُستخدم للإشارة إلى الكائن الحالي الذي يتم تنفيذ الكود داخله.

قيمة "this" تعتمد على سياق تنفيذ الكود وكيف تم استدعاء الدالة التي يتم تنفيذها. كما أنها لا تأخذ قيمتها من المكان الذي تم تعريف الدالة (Function) فيه بل المكان الذي تم استدعاء الدالة (Function) فيه.

مثال:

```
var person = {
  firstName: "Ali",
  lastName: "Fathy",
  fullName: function() {
    console.log(this.firstName + " " + this.lastName);
  }
```

```
};

person.fullName();
```

bind

هي دالة مدمجة في JavaScript تُستخدم لربط دالة بكائن معين كقيمة "this".

تُستخدم عادةً لحل مشكلة فقدان قيمة "this" داخل الدوال في سياقات معينة. تُستخدم bind عندما تريد ان تحدد الكائن (Object) الذي تريد ربط this به.

مثال:

```
var person = {
  firstName: "Ali",
  lastName: "Fathy",
  fullName: function() {
    console.log(this.firstName + " " + this.lastName);
  }
};
var printFullName = person.fullName;
```

```
printFullName();
```

// سيُظهر خطأ: "Cannot read property 'firstName' of undefined"

في المثال أعلاه عند استدعاء ()printFullName، يتم فقدان قيمة "this" داخل الدالة ويتم تفسيرها على أنها "undefined". لحل هذه المشكلة، يمكن استخدام "bind" كما يلي:

```
var person = {
  firstName: "John",
  lastName: "Doe",
  fullName: function() {
    console.log(this.firstName + " " + this.lastName);
  }
}
var printFullName = person.fullName.bind(person);
printFullName(); // سيطبع: "John Doe"
```

Classes

من الطرق المستخدمة لتطبيق الـ OOP في لغة الجافا سكريبت هي Classes>

في لغة الجافا سكريبت (JavaScript) يُمكنك استخدام الـ Classes لإنشاء أنواع مخصصة من الكائنات. الـ Classes تسمح لك بتعريف بيانات وسلوك مشتركين لمجموعة من الكائنات.

مثال:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}
```

```
const person1 = new Person("John", "Doe");

const person2 = new Person("Jane", "Smith");

console.log(person1.getFullName());

console.log(person2.getFullName());
```

sync VS async

التنفيذ المتزامن (Synchronous Execution):

عندما تقوم بتنفيذ الأكواد بشكل متزامن، يتم تنفيذ الأوامر بترتيبها واحدة تلو الأخرى، والبرنامج ينتظر حتى انتهاء كل أمر قبل أن يبدأ في تنفيذ الأمر التالي. هذا يعني أن الاكواد تعمل بشكل متسلسل و متتالي.

مثال: إذا كنتَ في طابور لشراء تذكرة مثلاً، فإنه يتوجب عليك الانتظار حتى ينتهي الشخص الذي يأتي قبلك في الطابور قبل أن تتمكن من الدخول.

المثال:

```
console.log("بداية");

alert("سيتم عرض هذه الرسالة تمامًا قبل الاستمرار في التنفيذ");

console.log("نهاية");
```

في هذا المثال، ستظهر رسالة alert قبل أن يتم طباعة "نهاية" في Console. هذا لأن التنفيذ هنا متزامن.

التنفيذ الغير متزامن (Asynchronous Execution):

عندما تستخدم الأكواد الغير متزامنة، يتم تنفيذ الأوامر دون انتظار انتهاء الأمر السابق.

وبدلاً من ذلك، يتم وضع الأوامر الغير تزامنية في طوابير (queues) خاصة وتنفيذها في وقت لاحق عندما تتوفر الفرصة.

مثال: إذا كنتَ في مطعم، فليس عليك الانتظار حتى يتطلب الأخرون قبل أن تقوم بطلب وجبتك. يمكنك أن تطلب على الفور وبجانبك شخص آخر يمكنه أيضًا أن يقوم بالطلب.

المثال:

```
console.log("بداية");

setTimeout(function() {

  console.log("أهلا علي");

}, 3000); // setTimeout ينفذ بعد 3 ثواني

console.log("نهاية");
```

في هذا المثال، سيتم عرض "بداية" ثم "نهاية" على الفور في Console، ثم ستظهر رسالة "أهلا علي" بعد مرور 3 ثوانٍ. هذا لأن setTimeout تُعتبر وظيفة غير متزامنة تُسجل للتنفيذ في وقت لاحق.

Single Thread Vs. Multi-Thread

لغة الجافا سكريبت (JavaScript) تُعتبر في الأساس Single-threaded، وهذا يعني أنها تعمل بسياق تنفيذ واحد (واحدة فقط) في الوقت نفسه. يعني هذا أن JavaScript تُنفذ الأوامر بتسلسل واحد فقط، ولا يمكنها تنفيذ أكثر من مهمة واحدة في نفس اللحظة.

ولكي تفهم الفرق بين Single Thread و Multi-Thread إليك المثال التالي:

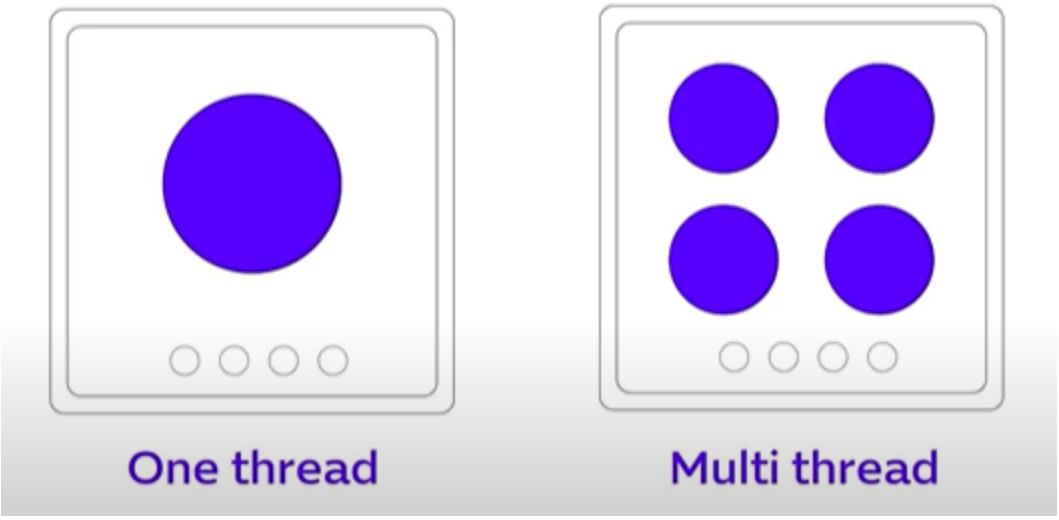
Single Thread

فلنفترض أن لديك بوتوجازًا ذو عينة واحدة فقط. هذا يعني أنه يمكنك طهي وجبة واحدة في وقت واحد. إذا قمت بوضع إناء على العين وبدأت في طهي الطعام، فإن أي عملية أخرى يجب أن تنتظر حتى انتهاء العملية الحالية. إذا كنت ترغب في تسخين الماء أو طهي الأرز في وعاء آخر، فإنك ستضطر إلى الانتظار حتى تنتهي عملية الطهي الأولى.

Multi-Thread

لنفترض أن لديك بوتوجاز متعدد العيون، ولديك أربعة عيون للطهي. في هذا الحال، يمكنك وضع عدة أواني في الوقت نفسه على العيون المختلفة وبدء عمليات الطهي بشكل متزامن. على سبيل المثال، يمكنك طهي اللحم على إحدى العيون، وتسخين الماء على العين الأخرى، وطهي

الأرز على العين الثالثة، والقيام بعمليات متعددة في نفس الوقت دون أن تتأثر أي عملية بأخرى.



هياكل البيانات (Data Structures)

هي طرق نتبعها لتنظيم البيانات والتعامل معها. من أشهر أنواع هياكل البيانات (Data Structures) هما الرصة (Stack) و الطابور (Queue).

الرصة (Stack)

الرصة (Stack) هي هيكل بيانات يعتمد على مبدأ (Last In, First Out) واختصاره LIFO. هذا يعني أن آخر عنصر يدخل هو أول عنصر يخرج.

الطابور (Queue)

الطابور هو هيكل بيانات يعتمد على مبدأ (First In, First Out) واختصاره FIFO. هذا يعني أن العناصر التي يتم وضعها في الطابور تخرج منه بنفس الترتيب الذي تم وضعها فيه. أي أن العنصر الذي يدخل أولًا يخرج أولًا.

الجافا سكريبت تتعامل مع عدة مفاهيم مهمة لتنظيم تنفيذ الكود مثل:

Call Stack

الـ Call Stack هي هيكل بيانات يتم استخدامه لتتبع ترتيب تنفيذ الدوال (Functions) في الجافا سكريبت.

عندما تُستدعى دالة (Function) في الكود، يتم وضعها في Call Stack، وعند انتهاء تنفيذ الدالة (Function)، يتم إزالتها من Call Stack.

Web API

واجهة برمجة تطبيقات الويب هي مجموعة من الوظائف والأدوات التي توفرها المتصفحات للجافا سكريبت للتفاعل مع الواجهة الرسومية للمستخدم (UI) والمزيد.

تشمل Web API أشياء مثل DOM API (للتلاعب بعناصر الصفحة والأحداث) و XMLHttpRequest و Fetch API وغيرها.

يتم ارسال الكود الغير متزامن مثل setTimeout إلى الـ Web API.

Callback Queue

هو هيكل بيانات يُستخدم لتخزين وظائف الرد (callbacks) التي يجب تنفيذها بعد اكتمال الأحداث أو الاستجابات.

عادةً ما تُستخدم وظائف الرد (callbacks) في سياق الأحداث المؤجلة، مثل استجابة الخادم لطلب HTTP أو انتهاء تنفيذ وظيفة زمنية (setTimeout).

عندما تتم إضافة وظيفة رد إلى Callback Queue، يجب استدعاؤها بواسطة Event Loop عندما تكون الـ Call Stack فارغة.

Event Loop

هي جزء أساسي من تنفيذ الجافا سكريبت في المتصفح.

تعمل Event Loop على مراقبة الـ Call Stack و Callback Queue.

إذا كانت الـ Call Stack فارغة، يتم استدعاء callback التالية من Callback Queue وتنفيذها.

Promises Overview

ال Promise يُستخدم لإدارة العمليات الغير متزامنة أو العمليات التي تحتاج إلى وقت للإنتهاء مثل استدعاء البيانات من الخادم (Server). ال Promise يُساعدك على تنظيم تنفيذ العمليات والتعامل مع النتائج بشكل فعال دون تداخل مع السياق الرئيسي لتطبيقك. الفكرة الأساسية لـ Promise هي توفير وسيلة للتعامل مع العمليات الغير متزامنة بطريقة تجعل الكود أكثر قراءةً وفهمًا وصيانةً. يتيح لك Promise تنفيذ الأكواد بشكل متسلسل وتحديد ما إذا تم الانتهاء بنجاح أم لا وكيفية التعامل مع الأخطاء إذا حدثت.

ال Promise تتيح لك استخدام الدوال مثل then و catch و finally للتحكم في سير العملية والتعامل مع النتائج والأخطاء. يمكنك استخدام then لتحديد ما يجب القيام به بعد انتهاء الوعد بنجاح، و catch للتعامل مع الأخطاء في حالة فشل الوعد، و finally لتحديد الأكواد التي يجب تنفيذها بغض النظر عن نجاح أو فشل الوعد.

ال Promise يؤدي نفس مهمة ال Callback ولكنه أفضل من ناحية readability والتعديل والتعامل معه.

ويوجد حالتين لـ **Promise** هما:

في لغة JavaScript الـ Promise هو عبارة عن نوع من الكائنات الذي يمثل قيمة مستقبلية (Future value) تمثل إتمام أو رفض عملية غير متزامنة. يساعد الـ Promise في التعامل مع الأكواد الغير متزامنة بشكل فعال.

الأساليب الأساسية المستخدمة مع الـ Promise هي then و catch و resolve و reject.

then: تُستخدم للتعامل مع قيمة ال Promise عندما يتم حلها (fulfilled). تستقبل دالة callback وتُنفذ عندما يتم حل الـ Promise بنجاح. يمكن أن يكون هناك عدة then متسلسلة لتنفيذ الأكواد بشكل متسلسل.

catch: تُستخدم للتعامل مع حالة الرفض (rejected) لـ Promise. تستقبل دالة callback وتُنفذ عندما يتم رفض الـ Promise.

resolve: تُستخدم لتغيير حالة الـ Promise من Pending إلى تم القبول. يتم استخدامها عادة داخل الدوال التي تقوم بالعمليات الغير متزامنة عند الانتهاء منها بنجاح.

reject: تُستخدم لتغيير حالة الـ Promise من Pending إلى تم الرفض. يتم استخدامها عادة داخل الدوال التي تقوم بالعمليات الغير متزامنة عند حدوث خطأ أو فشل.

مثال:

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('تم القبول');
  }, 1000);
});
myPromise
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  });
```

في لغة JavaScript تُستخدم الوعود (Promises) لإدارة العمليات الغير متزامنة. الوعد هو كائن يمثل قيمة ليتم حسابها في المستقبل ويتميز بقدرته على التعامل مع العمليات الغير متزامنة بشكل فعال. توفر الوعود (Promises) العديد من الدوال (Functions) المفيدة للتحكم في سير تنفيذ الوعود، ومن بينها:

Promise.all()

تُستخدم لانتظار اكتمال مصفوفة (Array) من الوعود و ارجاع قيمها مجتمعة كنتيجة واحدة. إذا فشل أي من الوعود، سيتم رفض الوعد الناتج.

مثال:

```
const promise1 = Promise.resolve(1);

const promise2 = Promise.resolve(2);

const promise3 = Promise.resolve(3);

Promise.all([promise1, promise2, promise3])

  .then((values) => {

    console.log(values); // [1, 2, 3]

  })
```

```
  .catch((error) => {
    console.error(error);
  });
```

Promise.any()

تُستخدم لانتظار اكتمال مصفوفة (Array) من الوعود وإرجاع قيمة أي وعد نجح أولاً.

إذا فشلت جميع الوعود، سيتم رفض الوعد الناتج.

مثال:

```
const promise1 = new Promise((resolve, reject) => setTimeout(reject, 100,
"Rejected"));
const promise2 = new Promise((resolve) => setTimeout(resolve, 200,
"Resolved"));
const promise3 = new Promise((resolve) => setTimeout(resolve, 300,
"Resolved"));
Promise.any([promise1, promise2, promise3])
  .then((value) => {
```

```
  console.log(value); // "Resolved"
 })
 .catch((error) => {
  console.error(error); // "All promises were rejected"
 });
```

Promise.allSettled()

تُستخدم لانتظار اكتمال مصفوفة (Array) من الوعود وإرجاع نتائجها بغض النظر عما إذا كان أي منها قد نجح أو فشل.

مثال:

```
const promise1 = Promise.resolve(1);
const promise2 = Promise.reject("Error");
const promise3 = Promise.resolve(3);
Promise.allSettled([promise1, promise2, promise3])
 .then((results) => {
  console.log(results);
```

```
  /*
  [
    { status: "fulfilled", value: 1 },
    { status: "rejected", reason: "Error" },
    { status: "fulfilled", value: 3 }
  ]
  */
});
```

Promise.race()

تُستخدم لانتظار الوعد الذي يتم حسابه أولاً من بين مصفوفة (Array) من الوعود. سيتم إرجاع نتيجة الوعد الذي تم حسابه أولاً، سواء نجح أو فشل.

مثال:

```
const promise1 = new Promise((resolve) => setTimeout(resolve, 100,
"Promise 1"));
```

```
const promise2 = new Promise((resolve) => setTimeout(resolve, 200,
"Promise 2"));
Promise.race([promise1, promise2])
  .then((value) => {
    console.log(value); // "Promise 1"
  });
```

Method	Resolves when	Rejects when	Returns
Promise.all	All promises resolve	Any promise rejects	Array of resolved values (in the same order)
Promise.allSettled	All promises settle	N/A	Array of objects representing the settled promises
Promise.race	First promise settles (resolve or reject)	Rejects if the first promise rejects	Value or reason of the first settled promise
Promise.any	First promise resolves	Rejects if all promises reject	Value of the first resolved promise

async-await

في لغة الجافا سكريبت، async و await هما جزء من ميزة معالجة الوعود (Promises) والتي تساعد في إدارة العمليات غير المتزامنة بشكل أفضل. تُستخدم async لتعريف دالة تعمل بشكل غير متزامن، بينما يُستخدم await داخل دالة async للانتظار حتى اكتمال وعد (Promise) والمضمون الذي يتم إعادته من هذا الوعد.

الفكرة الأساسية هي أنك تستخدم ال async لتحديد الدوال التي تنفذ بشكل غير متزامن، وبالتالي، يمكنك استخدام await داخل هذه الدوال للانتظار حتى تنتهي العمليات غير المتزامنة ويمكن الاستفادة من نتائجها بسهولة.

مثال:

```
// دالة تنفذ بشكل غير متزامن تقوم بالانتظار لمدة محدده ومن ثم تعيد رسالة.
function waitForTwoSeconds() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('تم انتهاء الانتظار لمدة ثانية');
```

```
    }, 2000);
  });
}
```

// دالة تستخدم async و await للانتظار حتى اكتمال دالة waitForTwoSeconds ومن ثم تقوم بعرض النتيجة.

```
async function main() {
  console.log('بدء التنفيذ');
  try {
    const result = await waitForTwoSeconds();
    console.log(result);
  } catch (error) {
    console.error('حدث خطأ: ', error);
  }
  console.log('انتهاء التنفيذ');
}
```

```
main();
```

في هذا المثال، تم إنشاء دالة waitForTwoSeconds التي تعيد وعد (Promise) بعد انتظار لمدة ثانيتين. ثم تم إنشاء دالة main التي تستخدم async و await للانتظار حتى انتهاء العملية غير المتزامنة في waitForTwoSeconds، ومن ثم تقوم بعرض النتيجة.

fetch

هي دالة مدمجة في لغة الجافا سكريبت تُستخدم لجلب البيانات (Data) من الخوادم (Servers). تُستخدم عادة لجلب ملفات أو بيانات من خوادم الويب أو الـ APIs. تعمل دالة "fetch" بشكل غير متزامن، مما يعني أنها لا تعلق تنفيذ البرنامج أثناء انتظار البيانات. كما أنها تقوم بإرجاع وعد (Promise).

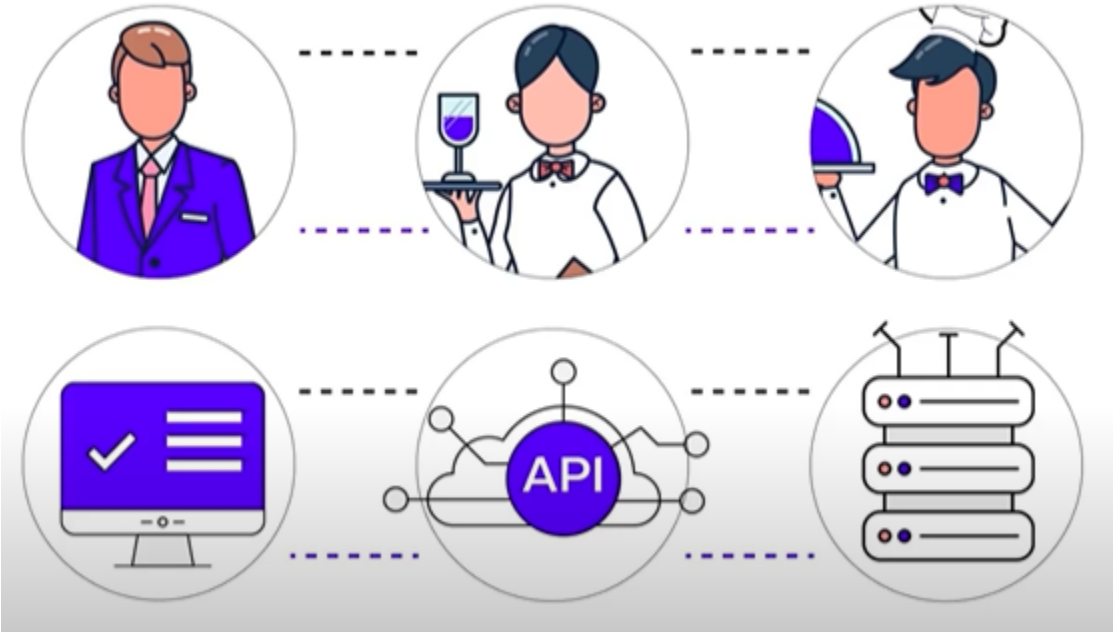
JSON

هو اختصار لـ "JavaScript Object Notation"، وهو تنسيق نصي يُستخدم لتبادل البيانات بين الخوادم (Servers) والمتصفحات والتطبيقات. يُتيح JSON للبيانات أن تكون في شكل نصي مقروء وسهل الفهم وفي الوقت نفسه يمكن تحويلها بسهولة إلى كائنات JavaScript. وهو واحدة من الطرق المستخدمة لنقل ال Objects من خادم (Server) إلى خادم (Server) أو من خادم (Server) الي مستخدم.

API

اختصار ل "Application programming interface". وهي طريقة تسمح للتطبيقات (Applications) أو ال Softwares بالتواصل. تشير عادة "API" إلى واجهة برمجة

التطبيقات عبر الإنترنت التي تتيح للمطورين الوصول إلى الخدمات والبيانات من خوادم الويب.



Fetch API

تستخدم لجلب الموارد من الخوادم عبر الشبكة، وتسمح بإرسال طلبات مختلفة مثل GET و POST و PUT و DELETE إلى الخوادم ومعالجة الاستجابات بشكل مرن.

مثال:

```
fetch('https://api.example.com/data')

  .then(response => {
```

```
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
                                        // قراءة البيانات كـ JSON
})
.then(data => {
                                        // يمكنك استخدام البيانات هنا
  console.log(data);
})
.catch(error => {
  console.error('There was a problem with the fetch operation:', error);
});
```

في لغة الجافا سكريبت، هناك مفاهيم تسمى "Task Queue" و "Microtask Queue" أو "Promise Queue" تلعب دورًا مهمًا في تنظيم تنفيذ الأكواد الغير متزامنة. تلك المفاهيم تساعد في تحديد أي جزء من الكود يجب تنفيذه أولاً عندما تكون هناك أكواد متعددة تنتظر التنفيذ.

Task Queue

هي قائمة تخزينية تحتوي على المهام (tasks) التي يجب تنفيذها بعد انتهاء تنفيذ الأكواد الحالية في السياق الرئيسي (main execution context) للبرنامج مثل console.log.
عندما تنتهي الأكواد الحالية من التنفيذ، يتم فحص Task Queue للتحقق من وجود مهام جديدة للتنفيذ. في ال Task Queue يتم تخزين setTimeout و setIntervals و Events.

Microtask Queue (Promise Queue)

هي قائمة تخزينية أخرى تُستخدم لتنظيم تنفيذ المهام الدقيقة (microtasks). المهام التي يتم تخزينها في Microtask Queue هي مهام ذات أولوية أعلى من المهام العادية في Task Queue، وعادة ما تكون مهام تمثل استجابة ل Promises والدوال الجاهزة (resolved promises)، مثل

.then() و .catch() و .finally().

ال Microtask Queue تُنفذ قبل Task Queue، مما يعني أن المهام الدقيقة تأخذ الأسبقية على المهام العادية.

لمعرفة كيف تعمل هذه المفاهيم معًا، يمكنك أن تفكر في Event Loop كمراقب للـ Task Queue وال Microtask Queue. عندما ينتهي التنفيذ الحالي يتحقق Event Loop من وجود مهام في Microtask Queue ويقوم بتنفيذها جميعًا قبل أن يتحقق من وجود مهام في Task Queue ويقوم بتنفيذها بالترتيب.

الوحدات (Modules) في لغة الجافا سكريبت (JavaScript) هي طريقة لتنظيم وتجزئة الكود إلى ملفات منفصلة بهدف جعل الكود أكثر تنظيمًا وإعادة استخدام. حيث أنها تساعدك على مشاركة الكود بين الملفات المختلفة. ولكي تصل للبيانات (Data) الخاصة بالوحدات (Modules) نستخدم import و export.

الوحدات (Modules) في لغة الجافا سكريبت (JavaScript) هي طريقة لتنظيم وتجزئة الكود إلى ملفات منفصلة بهدف جعل الكود أكثر تنظيمًا وإعادة استخدام. حيث أنها تساعدك على مشاركة الكود بين الملفات المختلفة. ولكي تصل للبيانات (Data) الخاصة بالوحدات (Modules) نستخدم import و export.

يتم استدعاء ملف الـ module.js كأي ملف js ولكن بإضافة type="module"

```
<!DOCTYPE html>
<html>
  <body>
    <script type="module" src="module.js"></script>
  </body>
</html>
```

التصدير (Export)

يُتيح لك تصدير دالة (Function) أو كود معين. يمكنك تصدير الأشياء باستخدام الكلمة الرئيسية export. وهناك ثلاثة أنواع رئيسية للتصدير:

Default Export

يمكنك تصدير وحدة واحدة كافتراضي من كل ملف. عند استيراد الوحدة، يمكنك إعطاء اسم مختلف لها. مثال:

```
// في ملف math.js

export default function add(a, b) {

  return a + b;

}

// في ملف main.js

import customAdd from './math.js';

console.log(customAdd(2, 3)); // ستطبع 5
```

Named Exports:

يمكنك تصدير متغيرات أو دوال معينة باستخدام الكلمة الرئيسية export. عند استيراد الوحدة، يجب استخدام نفس الاسم الذي تم تصديره. مثال

مثال:

```
// في ملف utils.js
```

```
export function multiply(a, b) {
  return a * b;
}


export const PI = 3.14159;
```

// في ملف main.js

```
import { multiply, PI } from './utils.js';
console.log(multiply(2, 3)); // 6 ستطبع
console.log(PI); // 3.14159 ستطبع
```

Re-exports

مثال:

// في ملف helpers.js

```
export function square(x) {
  return x * x;
}
```

// في ملف main.js

export { square } from './helpers.js'; // إعادة تصدير الدالة square

الاستيراد (Import)

يُتيح لك الاستيراد استخدام الأشياء التي تم تصديرها من ملف آخر. يتم استيراد الوحدات باستخدام الكلمة الرئيسية import.

مثال:

// في ملف main.js

import add from './math.js';

import { multiply, PI } from './utils.js';

console.log(add(2, 3)); // ستطبع 5

console.log(multiply(2, 3)); // ستطبع 6

console.log(PI); // ستطبع 3.14159

On demand import

في ES Modules في الجافا سكريبت، يمكنك استخدام "On Demand Import" أو "Dynamic Import" لاستيراد وقت الحاجة فقط أي استيراد وحده تحتاجها في وقت تنفيذ البرنامج بدلاً من استيرادها مسبقًا في بداية البرنامج. هذا يتيح لك تحميل الوحدات حسب الحاجة، مما يقلل من وقت التحميل الأولي واستهلاك الموارد ويساعد في تحسين أداء التطبيقات.

مثال:

```
// module.js
export function greet(name) {
  return `Hello, ${name}!`;
}
// main.js
document.getElementById('button').addEventListener('click', async () => {
  // استخدم دالة import() لاستيراد الوحدة عند الحاجة
  const module = await import('./module.js');
```

```
  const message = module.greet('John');

  alert(message);

});
```

في هذا المثال قمنا بإنشاء وحدة "module.js" التي تحتوي على دالة greet وتصديرها باستخدام export. ثم في ملف "main.js"، نستخدم دالة ()import لاستيراد "module.js" عندما يتم النقر على الزر.

Bundlers

ال Bundlers في الجافا سكريبت هي أدوات تستخدم لجمع وإدارة والمكتبات (libraries) المختلفة التي يحتاجها تطبيق الويب الخاص بك. تعتبر عملية bundling مهمة في تطوير الويب حيث تقوم بجمع ملفات الجافا سكريبت والأنماط (CSS) والصور والمكتبات وتحويلها إلى ملف واحد أو عدة ملفات صغيرة يمكن تضمينها في صفحة الويب الخاصة بك.

ما هي مميزات ال Bundlers؟

- تُساعد في تقسيم الكود على شكل Modules.
- تُساعد عمل دعم للميزات (Features) الجديدة.
- تُساعد في تحويل ال Preprocessors مثل Sass إلى CSS.
- إزالة الكود الغير مستخدم (Dead code elimination).
- تقوم بعمل تصغير الكود (Minification)، وتقوم بتغيير أسماء المتغيرات (Variables) الي اسماء أصغر وهذه العملية تسمى Uglify.
- تقليل حجم الصور لكي تناسب الموقع.

من أشهر ال Bundlers هي:

- Webpack
- Parcel
- Rollup
- Vite.js

HTTP vs. HTTPS

ال HTTP و HTTPS هما اختصاران لبروتوكولي انتقال البيانات عبر الشبكة (Hypertext Transfer Protocol)، والفرق الرئيسي بينهما يتعلق بالأمان:

HTTP (Hypertext Transfer Protocol)

هو بروتوكول نقل البيانات القديم وغير المشفر. يتم استخدامه لنقل المعلومات بين متصفح الويب وخادم الويب بطريقة غير مشفرة. البيانات المرسلة عبر HTTP يمكن أن تكون عرضة للاختراق والاستيلاء عليها بسهولة من قبل مهاجمين محتملين.

HTTPS (Hypertext Transfer Protocol Secure)

هو نسخة محمية ومشفرة من HTTP.

يستخدم تشفير SSL/TLS لحماية البيانات المرسلة بين المتصفح والخادم.

يتميز HTTPS بالأمان الأفضل ويجعل من الصعب على المهاجمين استغلال الاتصال والاستيلاء على البيانات المنقولة.

بشكل عام، يُفضل استخدام HTTPS دائمًا على HTTP للحفاظ على أمان الاتصال وحماية البيانات الشخصية والحساسة التي يمكن أن تنتقل عبر الإنترنت.