UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

JavaScript

Table of Contents

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction to JavaScript

What is JavaScript?

- JavaScript was initially created to make pages *"dynamic"*
- Programs in this language are called **scripts**
- They can be written directly into an HTML page and executed automatically when pages are loaded by browsers
- ⚠️ JavaScript and Java are two very different languages

How JavaScript is Executed

- Browsers incorporate engines also called the JavaScript virtual machine
  - **V8**: Chrome, Edge, Opera
  - **SpiderMonkey**: Firefox
  - **Chakra**: (deceased) Internet Explorer
- JavaScript is a language used both server-side and browser-side

Interpreted Language

- In computing, we talk about **interpreted** or **compiled** code
- JavaScript is an **interpreted** language: the code is executed from top to bottom and the result of the executed code is sent immediately
- Compiled languages, on the other hand, are transformed (compiled) into another form before being executed by the computer (C, C++)

Advantages of JavaScript

- Full integration with HTML and CSS
- Simple and clear syntax
- Supported by all browsers and enabled by default

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

JavaScript Basics

The script tag

- JavaScript programs can be inserted into any part of an HTML document using the <script> tag
- The script tag contains code that is automatically executed when the browser encounters the tag

```
...
<body>
  <script>
    alert( 'Hello, world!' );
  </script>
</body>
...
```

External Scripts

- It is generally advised to separate the JavaScript from the HTML page.
- JavaScript files have a `.js` extension.
- To import it, just add the `src` attribute to the script tag.
- The files are downloaded and then stored in the cache.

```
<script src="/path/to/script.js"></script>
```

Instruction

- Instructions are syntax constructs and commands that perform actions.
- Each instruction is typically written on a separate line.
- Instructions end with `;` even though it is *generally* optional when followed by a newline.

```
alert("Hello");
alert("World");
```

Comments

- Single-line comments start with two slashes `//`.

```
alert("World"); // This is a comment
```

- Multi-line comments start with a slash and an asterisk `/*` and end with an asterisk and a slash `*/`.

```
/* An example with two messages.
This is a multi-line comment.
*/
alert("Hello");
```

About "use strict"

- The directive `"use strict";` or `'use strict';` at the beginning of a script allows for the use of what's called **modern** JavaScript.
- Modern JavaScript supports **classes** and **modules** which automatically enable use strict.

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Variables

Variables

- A variable is a named storage for data.
- It behaves like a box where we store information.
- To create a variable in JavaScript, simply use the `let` keyword.

```
let message;

message = "Hello"; // stores the string 'Hello'

let firstName = "Toto"; // Declaration and assignment
```

How Variables Work

- When the value is changed, the old data is removed from the variable.

Variable Names

1. The name should contain only **letters**, **numbers**, symbols **$** and **_**.
2. The first character should not be a number.
3. When the name contains multiple words, **camelCase** is commonly used: `myVeryLongName`.
4. Variable names are case-sensitive.
5. Do not use reserved words: **return**, **class**, **let**...

Constants

- To declare a constant (unchanging value), one can use `const` instead of `let`.
- They cannot be reassigned. An attempt to do so would cause an error.
- Constants are typically named in uppercase for aliases of hard-coded values in the code.

```
const MY_BIRTHDAY = "18.04.1982";
const COLOR_ORANGE = "#FF7F00";
```

Naming Rules

A variable name should have a clear and obvious meaning, describing the data it stores.

- Use human-readable names like `userName` or `shoppingCart`.
- Avoid abbreviations or short names.
- Ensure the name is as descriptive and concise as possible.
- Agree with your team (and yourself) on the terms used.

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Data Types

JavaScript Data Types

- There are 8 basic data types in JavaScript.

Type	Description	Example
number	integer or floating point	`let n = 123;`
bigint	arbitrary-length integers	
string	character strings	`let str = "Hello";`
boolean	true/false	`let nameFieldChecked = true;`

Type number

- The number type is used for both integers and floating-point numbers.
- Apart from regular numbers, there are specific values: `Infinity`, `-Infinity`, and `NaN`.
- Mathematical operations never cause errors in JavaScript.

Details on the Number Type

- **NaN**: The global NaN property represents a value that is **N**ot **a** **N**umber.
- Dividing by 0 returns Infinity: `1/0 => Infinity`.
- A division that causes a calculation error returns NaN: `"toto"/2 => NaN`.

Type BigInt

- BigInt was recently added to the language to represent arbitrary-length integers.
- A BigInt value is created by appending n to the end of an integer.
- Currently, BigInt is supported in Firefox/Chrome/Edge/Safari, but not in IE.

```
const bigInt = 1234567890123456789012345678901234567890n;
```

Strings

- A string in JavaScript must be in quotes.
- JavaScript accepts 3 types of quotes:
  1. Double quotes: "Hello"
  2. Single quotes: 'Hello'
  3. Backticks: `Hello`
- Backticks allow you to embed variables and expressions into a string by wrapping them in ${...}.

```
`the result is ${1 + 2}`;
```

Boolean Type

- The boolean type has only two values: `true` and `false`.
- Boolean values are generally used for comparisons or conditional structures.

```
let nameFieldChecked = true;
let isGreater = 4 > 1;
```

The "null" Value

- The special value **null** does not belong to any of the previously seen types.
- In JavaScript, null is not a "reference to a non-existing object" or a "null pointer" as in other languages.
- It's a special value that signifies **"nothing"**, **"empty"**, or **"unknown value"**.

```
let age = null;
```

The "undefined" Value

- The special value **undefined** stands out from the others. It's a type all of its own, just like null.
- The meaning of undefined is **"value is not assigned"**.
- If a variable is declared but not assigned, then its value is precisely **undefined**.

```
let age;

alert(age); // displays "undefined"
```

Objects and Symbols

- **Objects** are used to store collections of data and more complex entities.
- The **symbol** type is used to create unique identifiers for objects.

The "typeof" Operator

- The **typeof** operator returns the type of the argument.
- It's useful when you want to process values of different types differently or to quickly check.

```
typeof undefined; // "undefined"

typeof 0; // "number"

typeof 10n; // "bigint"

typeof Math; // "object"
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Operators

Assignment Operators

Name	Operator	Meaning
Assignment	x = f()	x = f()
Assignment after addition	x += f()	x = x + f()
Assignment after subtraction	x -= f()	x = x - f()
Assignment after multiplication	x *= f()	x = x * f()
Assignment after division	x /= f()	x = x / f()
Assignment of remainder	x %= f()	x = x % f()

Mathematical Operators

Operator	Examples
Addition	1 + 1
Subtraction	1 - 1
Division	1 / 1
Multiplication	1 * 1
Remainder	4 % 2
Exponential	4 ** 2
Increment	x++ or ++x

Comparison Operators

Operator	Examples
Equality	3 == var1
Inequality	var1 != 4
Strict equality	3 === var1
Strict inequality	var1 !== "3"
Greater than	var2 > var1
Greater or equal	var2 >= var1
Less than	var1 < var2

Logical Operators

Operator	Usage	Description
Logical AND (&&)	expr1 && expr2	returns true if both operands are true otherwise false
Logical OR (||)	expr1 || expr2	returns true if one of the operands is true, and false if both are false
Logical NOT (!)	!expr	Returns false if its single operand can be converted to true, otherwise returns true

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Conditional Structures

if

- The `if` statement executes a statement if a given condition is true or equivalent to true.
- Condition: An expression that evaluates to **true** or **false**

```
if (condition) {
  statement1;
}
```

if ... else

- If the `else` clause exists, the statement is executed if the condition evaluates to false.

```
if (condition) {
  statement1;
} else {
  statement2;
}
```

if ... else if ... else

- Multiple if...else statements can be nested to create an else if structure.

```
if (condition1) {
  instruction1;
} else if (condition2) {
  instruction2;
} else if (condition3) {
  instruction3;
}
...
else {
  instructionN;
}
```

switch

- The `switch` statement evaluates an **expression**, and based on the resulting value and the matched case, executes the corresponding instructions.

```
const code = 200;
switch (code) {
  case 200:
    console.log("Ok");
    break;
    ...
  default:
    console.log(`Unknown code`);
}
```

Ternary Operator

- The ternary operator `?` is commonly used as a shortcut for if...else statements.
- `expression ? valueIfTrue : valueIfFalse;`

```
let elvisLives = Math.PI > 4 ? "Yep" : "Nope";
```

Nullish Coalescing Operator

- The logical operator `??` returns its right-hand operand when its left-hand operand is null or undefined.
- `leftExpr ?? rightExpr`

```
const valA = nullableValue ?? "default value";
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Iterative Structures

While

- The `while` statement creates a loop that executes as long as a test condition is true.
- The condition is evaluated before executing the instruction contained within the loop.

```
let n = 0;

while (n < 3) {
  n++;
}

console.log(n);
```

Do While

- The `do...while` statement creates a loop that executes an instruction until a test condition is no longer true.
- Therefore, the block of instructions defined in the loop is executed at least once.

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

For

- The `for` instruction creates a loop consisting of three optional expressions, separated by semicolons and enclosed in parentheses, followed by an instruction to be executed in the loop.

```
for (begin; condition; step) {
  // Instructions
}
```

```
for (let i = 0; i < 3; i++) {
  console.log(i);
}
```

Continue et Break

- The `continue` instruction stops the execution of instructions for the current iteration of the loop. Execution is resumed at the next iteration.
- The `break` instruction allows you to end the current loop and pass control of the program to the instruction following the finished instruction.

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Functions

Definition

- A function is a *"sub-program"* that can be called by code outside of the function.
- In JavaScript, functions are **first-class** objects.
- They can be manipulated and passed around, and they can have **properties** and **methods**, just like every other JavaScript object.

Function Declaration

- To create a function, we can use a function declaration.

```
function name(parameter1, parameter2, ...parameterN) {
  // instructions
}
```

```
function showMessage() {
  alert("Hello everyone!");
}
```

- To call a function, use the function name followed by parentheses

Default Values

- You can specify a default value for a parameter in the function declaration by using `=`.

```
function showMessage(from, text = "No text provided") {
  alert(from + ": " + text);
}

showMessage("Ihab"); // Ihab: Empty text
```

Returning a Value

- A function can return a value to the calling code as a result using the `return` keyword.
- When the `return` statement is encountered, the function exits.

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

let result = sum(1, 2);
alert(result); // 3
```

Function Expression

- In JavaScript, a function isn't a "magical language construct", but rather a **specific type of value**.
- Omitting a name is allowed for function expressions.

```
let sayHi = function() {
  alert("Hello");
};
```

Callback

- A callback is a function passed into another function as an argument, which is then invoked inside the outer function.

```
function greeting(name) {
  alert("Hello " + name);
}
function processUserInput(callback) {
  var name = prompt("Enter your name.");
  callback(name);
}
processUserInput(greeting);
```

Arrow Functions

- Arrow functions are a shorthand syntax for function expressions.

```
([param] [, param]) => {
   // statements
}
(param1, param2, …,) => expression
param => expression
```

```
let sum = (a, b) => a + b;
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Arrays

Definition

- Arrays are list-like objects whose prototype has **methods** to **traverse** and **modify** the array.
- The global **Array** object is used to create arrays.
- Arrays are high-level objects (in terms of human-machine complexity).

Declaring an Array

- There are two syntaxes for creating an empty array:

```
let fruits = [];
// OR
let fruits = new Array();
```

- The first syntax is preferable:

```
let fruits = ["Apple", "Banana"];
```

- Access an item (via its index) in the array:

```
let first = fruits[0];
```

- Add an item to the end of the array:

```
let newLength = fruits.push("Orange");
```

- Remove the first item from the array:

```
let first = fruits.shift();
```

- To remove the first item from the array:

```
let first = fruits.shift();
```

Manipulating an Array

- Add to the beginning of the array:

```
let newLength = fruits.unshift("Strawberry");
```

- Copying an array:

```
let shallowCopy = fruits.slice();
let copy = [...fruits];
```

- Iterate over an array using `.forEach`

```
fruits.forEach((item, index) => console.log(item));
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Objects

Definition

- An object is a collection of properties, and a property is an association between a name (or key) and a value.
- A property's value can be a function, in which case the property is known as a method.
- Objects are used to store collections of various data and more complex entities.
- An object can be created with figure brackets `{...}` with an optional list of properties.

Instantiate an Object

- There are 2 syntaxes to instantiate an object:

```
let user = new Object(); // "object constructor" syntax
let user = {}; // "object literal" syntax
```

Properties

- Object properties are essentially regular **variables**, but they are **attached to objects**.
- The properties of an object represent its **characteristics** and can be accessed using a dot notation **"."**.
- Property naming convention is in camelCase.

```
objectName.propertyName;
```

```
let maVoiture = {
  make: "Ford",
};
```

Properties

- Objects are sometimes referred to as "associative arrays".
- Each property is associated with a string that allows access to it.

```
myCar["manufacturer"] = "Ford";
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

Checking Property Existence

- In JavaScript, it's possible to access any property.
- Reading a non-existent property simply returns `undefined`.
- There's also a special operator `in` for checking the existence of a key.

```
let user = { name: "John", age: 30 };
console.log("age" in user); // true
```

For In

- The `for...in` loops allow you to iterate over all enumerable properties of an object and its prototype chain.

```
for (const property in object) {
  console.log(`${property}: ${object[property]}`);
}
```

Methods

- There are two ways to declare methods in JavaScript.
- The shorter syntax is generally preferred.

```
user = {
  sayHi: function() {
    alert("Hello");
  },
};
```

```
user = {
  sayHi() {
    alert("Hello");
  },
};
```

The `this` Keyword

- It's common for an object's method to need access to the information stored in the object to perform its task.
- To access the object, a method can use the `this` keyword.

```
let user = {
  name: "John",
  sayHi() {
    // this refers to the current object
    console.log(this.name);
  },
};
```

Specificity of `this`

- In JavaScript, `this` is "free", its value is evaluated at the time of the call and does not depend on where the method was declared, but rather on the object "before the dot".
- `this` in an arrow function corresponds to the value of the enclosing context.

Constructor Function

- Constructor functions are technically regular functions.
- They allow for the creation of similar objects.
- There are, however, two conventions:
  1. They are named with an initial capital letter.
  2. They should only be executed using the `new` operator.

Declaring a Constructor

- When a function is executed with `new`:
  1. A new empty object is created and assigned to `this`.
  2. The function body executes.
  3. The value of `this` is returned.

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");
```

Adding a Method to the Constructor

- Utilizing constructor functions to create objects provides significant flexibility.
- ⚠️ This method is not optimized; it would be better to add it to the function's prototype.

```
function User(name) {
  this.name = name;
  this.sayHi = function () {
    alert("My name is: " + this.name);
  };
}

let john = new User("John");
john.sayHi(); // My name is: John
```

Optional Chaining

- Optional chaining `?.` provides a safe way to access nested object properties.
- Optional chaining `?.` stops the evaluation if the value before `?.` is **undefined** or **null** and returns **undefined**.

```
let user = {}; // user doesn't have an address
console.log(user?.address?.street); // undefined (no error thrown)
user.admin?.(); // Execute a function if it exists
user?.[key]; // access the property
```

Object-to-Primitive Conversion

- The conversion from object to primitive is automatically called by many built-in functions.
- The conversion algorithm is:
  1. Call **obj[Symbol.toPrimitive](hint)** if the method exists.
  2. Otherwise, if the hint is "string", try **obj.toString()** then **obj.valueOf()**.
  3. Otherwise, if the hint is "number" or "default", try **obj.valueOf()** then **obj.toString()**.

`toString()` Method

- In practice, it's often sufficient to implement only obj.toString() as the method for string conversions.

```
let obj = {
  toString() {
    return "2";
  },
};
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Classes

Definition

- JavaScript classes were introduced in ECMAScript 2015 (ES6).
- They are primarily syntactic sugar over JavaScript's **prototypal inheritance**.
- In fact, classes are just **special functions**.
- As such, classes are defined in the same way functions are: by **declaration**, or by **expression**.

Class Declarations

- For a simple class declaration, you use the `class` keyword followed by the name of the class being declared.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Constructor

- The `constructor` method is a **special method** used for creating and **initializing objects** that are created with a class.
- There can only be **one method** with the name "constructor" in a class.
- Having more than one occurrence of a constructor method in a class will throw a **SyntaxError**.

Getters/Setters

- Just like literal objects, classes can include getters/setters, computed properties, etc.

```
class User {
  constructor(name) {
    // invokes the setter
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      console.log("Name is too short.");
      return;
    }
    this._name = value;
  }
}
```

Inheritance

- The `extends` keyword, used in class declarations or class expressions, creates a class as a child of another class.
- `super` is used to call the parent class's constructor and access the parent class's properties and methods.

```
class Mage extends Hero {
  constructor(name, level, spell) {
    super(name, level);
    this.spell = spell;
  }
}
```

Static Methods

- The `static` keyword defines a static method for a class.
- Static methods are called on the class itself, not on instances of the class.

```
class Calculator {
  static sum(a, b) {
    return a + b;
  }
}

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

Encapsulation

- An experimental proposal allows for the definition of private variables in a class using the `#` prefix.

```
class ClassWithPrivateField {
  #privateField;
  #privateMethod() {
    return this.#privateField;
  }
}
```

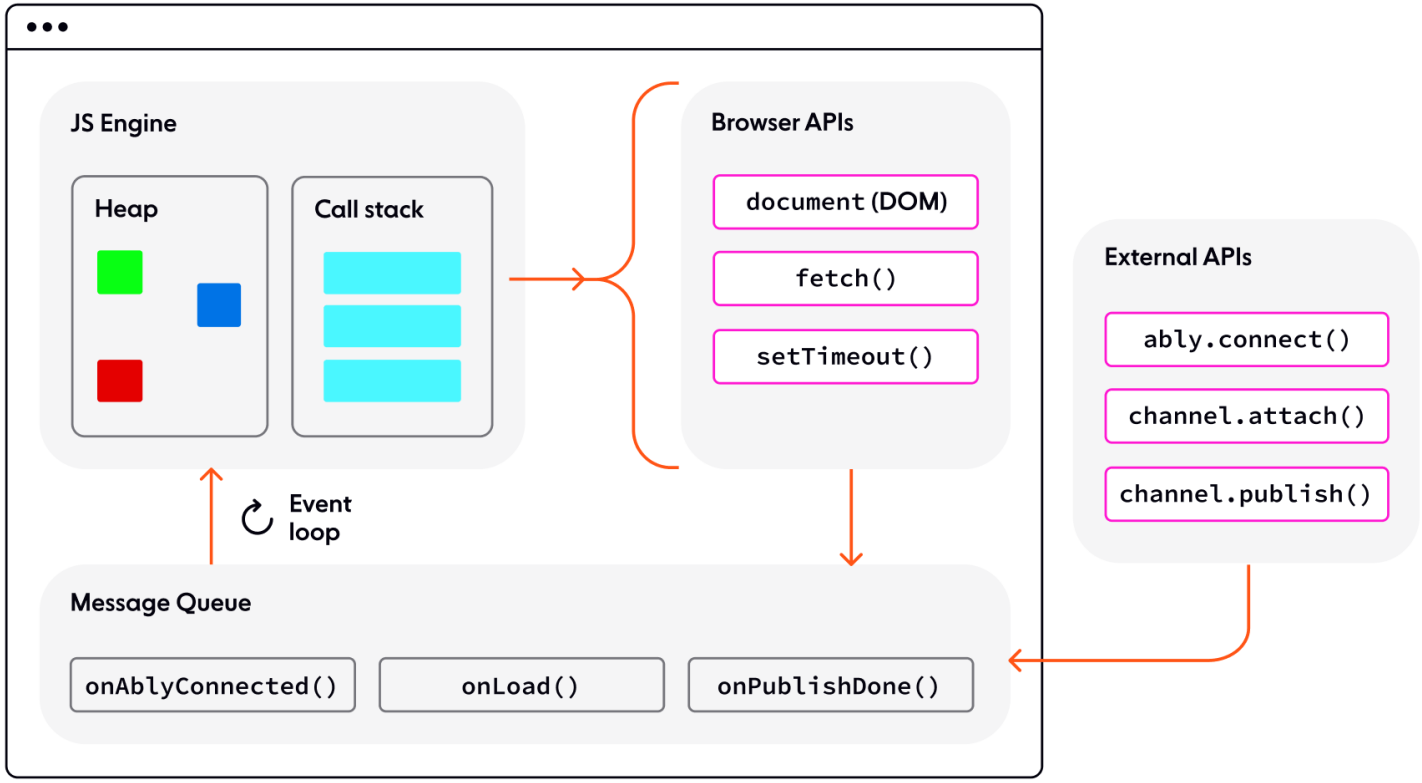
UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Promises, Async Await

Asynchronous

- JavaScript is **synchronous** by default and is run on a **single thread**.
- Code cannot spawn new threads and run in parallel.
- Computers are **asynchronous** by design.
- Asynchronous means that things can happen **independently of the main program flow**.

Asynchronous in the Browser



Callbacks

- A callback is a simple function that is **passed as a value** to another function and is only executed once the event happens (e.g., a click).
- JavaScript has first-class functions, meaning they can be assigned to variables and passed to other functions (these are called **higher-order functions**).

Using Callbacks

- Callbacks are used everywhere, not just in DOM events.

```
window.addEventListener("load", () => {
  // Window is loaded
  // Do whatever you want here
});

setTimeout(() => {
  // executes after 2 seconds
}, 2000);
```

Error Handling in Callbacks

- A common strategy is to make the first parameter an error object: **error-first callback**.
- If there's no error, the object is **null**.

```
fs.readFile("/file.json", (err, data) => {
  if (err) {
    console.log(err); // handle error
    return;
  }

  console.log(data); // process data
});
```

The Callback Problem

- Each callback introduces a level of nesting which can add to the complexity.
- This is commonly referred to as **callback hell**.

```
window.addEventListener("load", () => {
  document.getElementById("button").addEventListener("click", () => {
    setTimeout(() => {
      items.forEach((item) => {
        // your code here
      });
    }, 2000);
  });
});
```

Promises

With ES6, JavaScript introduced several features that help us with asynchronous code that doesn't involve using callbacks: **Promises** (ES6) and **Async/Await** (ES2017).

- A promise is an object representing the eventual **completion** or **failure** of an asynchronous operation.
- A promise is an **object** to which we attach callbacks, instead of passing callbacks into a function.

Promise States

- A promise can be in one of 3 states:
  - Operation in progress (*pending*)
  - Operation completed successfully (*fulfilled*)
  - Operation terminated/stopped after a failure (*rejected*)
- Most of JavaScript's asynchronous operations are already built using promises.

Creating a Promise

- JavaScript's Promise constructor requires a function which takes two arguments: **resolve** and **reject**

```
let myPromise = new Promise((resolve, reject) => {
  // Asynchronous task
  // Call resolve() if the promise is fulfilled
  // Call reject() if it's rejected
});
```

Promise Example

```
function loadScript(src) {
  return new Promise((resolve, reject) => {
    let script = document.createElement("script");
    script.src = src;
    document.head.append(script);
    script.onload = () => resolve("File " + src + " loaded successfully");
    script.onerror = () => reject(new Error("Failed to load " + src));
  });
}

const promise1 = loadScript("loop.js");
const promise2 = loadScript("script2.js");
```

Promise Chaining

- The `.then` handler creates a new promise with the result obtained from the resolve function.
- When a handler returns a value, it becomes the result of that promise.

```
myPromise
  .then((result1) => {
    // Handle the case where the promise is fulfilled successfully
  })
  .then((result2) => {
    // Handle the result returned by the previous then
  });
```

Error Handling

- Promise chains are excellent for error handling.
- When a promise is rejected, the control jumps to the nearest `.catch` error handler.
- Typically, an Error object is returned.

```
fetch("https://no-such-server.blabla")
  .then((response) => response.json())
  .catch((err) => alert(err));
```

Static Methods of Promise

- There are 6 static methods of the Promise class:
  1. `Promise.all(promises)` ★ : waits for all promises to resolve and returns an array of their results.
  2. `Promise.allSettled(promises)` : waits for all promises to settle and returns their results as an array of objects.
  3. `Promise.race(promises)` : waits for the first promise to settle, and its result/error becomes the outcome.

Async

- There's a special syntax to work with promises in a more comfortable fashion, called **"async/await"**.
- The word `async` before a function means it always returns a promise.
- async ensures that the function returns a promise, and wraps non-promises in it.

```
async function f() {
  return 1;
}
f().then(alert); // 1
```

Await

- The keyword `await` ensures that JavaScript waits until a promise settles and returns its result.
- `await` offers a more elegant syntax for obtaining the result from a promise.
- ⚠️ `await` only works inside asynchronous functions.

```
async function showAvatar() {
  let response = await fetch("/article/promise-chaining/user.json");
  let user = await response.json();
  // ...
}
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Modules

Introduction

- As applications grow, it's often useful to **split them into multiple files**, called "modules".
- A module typically contains **a class** or a **library of functions** for a specific task.
- For a long time, JavaScript lacked native support for modules.

History

Back when JavaScript lacked a built-in script import system, several libraries were implemented to address this issue:

- **AMD**: Implemented by require.js.
- **CommonJS**: A module system created for Node.js.
- **UMD**: A universal system compatible with both AMD and CommonJS.

The module system was introduced in ES6 using the `import` and `export` keywords.

Import and Export

- The `export` keyword tags variables and functions that should be accessible from outside the current module.
- `import` allows the importing of functionalities from other modules.

```
// 📁 sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

```
import { sayHi } from "./sayHi.js";
sayHi("John");
```

Export Before Declarations

- It's possible to export any declaration by prefixing it with the `export` keyword.

Examples:

```
// Export a constant
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// Export a class
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

Export After Declarations

- You can export declarations after they have been defined.

Examples:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export { sayHi, sayBye };
```

Import avec *

- Generally, we place a list of what we want to import inside curly braces `import {...}`.
- But if there's a lot to import, we can import everything as an object using `import * as <obj>`.

Examples:

```
import * as say from "./say.js";

say.sayHi("John");
say.sayBye("John");
```

Import with Aliases

- For simplicity or to avoid naming conflicts, aliases can be used during imports.
- The keyword `as` is used to rename both imports and exports.

Examples:

```
import { sayHi as hi, sayBye as bye } from "./say.js";

hi("John"); // Hello, John!
bye("John"); // Bye, John!
```

```
export { sayHi as hi, sayBye as bye };
```

Default Export

- Typically, there are two common types of modules:
  1. Modules that contain multiple declarations.
  2. Modules that declare a single entity, e.g., a class or function.
- For the second type of module, the `default` keyword provides a more straightforward way to export and later import.

Examples:

```
// 📁 user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

// In another file
import User from "./user.js";

export default API_KEY = "AOFKVZP120501KFK205";
```

```
import API_KEY from "./user.js";
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Generators

Definition

- In JavaScript, a generator is a **special function** that can produce multiple values **lazily**, meaning it doesn't compute all values upfront.
- Generators are defined using the `function*` keyword, followed by the function name and parentheses.
- A generator can have one or more `yield` expressions which produce the current value of the iteration.

Declaration

- When a generator is called, it doesn’t run its code. Instead, it returns a special object, termed the "generator object."

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();
alert(generator); // [object Generator]
```

How Generators Work

- The main method of a generator is `next()`.
- When it's called, the function execution runs until the nearest yield statement.
- The result of `next()` is always an object with two properties:
  - **value**: The yielded value.
  - **done**: true if the function code is finished, otherwise false.

Example

- Here's an example of how to use a generator:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}
```

Iterating Over a Generator

- Iterating through the values from a generator can be done using `for..of`.
- The `for..of` loop will ignore the final value when `done: true`.

Example

- Here's how you can iterate over the values from a generator:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for (let value of generator) {
  alert(value); // alerts 1, then 2
}
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Thank You for Your Attention

Thank you for listening and being an engaged audience. I appreciate your time and interest in this topic.

Questions?

If there are any questions or clarifications needed, please don't hesitate to ask. I'm here to help!