

# JavaScript / Typescript

# Table of Contents

1. JavaScript Review
2. New Features in ECMA Script 2015
3. TypeScript
4. Module

# Tools

1. Vscode
2. Nodejs
3. Git

# JavaScript Review

# What is JavaScript?

- JavaScript was initially created to make pages *"dynamic"*
- They can be written directly into an HTML page and executed automatically when pages are loaded by browsers

# 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

# Variables

# Variables

- To create a variable in JavaScript, simply use the `let` keyword.

## Before ES6+

```
var message;  
message = "Hello"; // stores the string 'Hello'  
var firstName = "Toto"; // Declaration and assignment
```

## After ES6+

```
//Variable After ES6+  
let message;  
message = "Hello"; // stores the string 'Hello'  
let firstName = "Toto"; // Declaration and assignment
```



# 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.

The `const` keyword was introduced in ES6.

# 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.

# Data Types

# JavaScript Data Types

- There are 8 basic data types in JavaScript.

Type	Description	Example
number	integer or floating point	<code>let n = 123;</code>
bigint	arbitrary-length integers	
string	character strings	<code>let str = "Hello";</code>
boolean	true/false	<code>let nameFieldChecked = true;</code>
null	unknown values	<code>let age = null;</code>
undefined	unassigned values	<code>let age; alert(age); // shows "undefined"</code>
symbol	unique identifiers	
object	complex data structures	

# 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 **Not a Number**.
- 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`, added in ES6
- 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"
```

# 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 $++x$
Decrement	$x--$ or $--x$

# Comparison Operators

Operator	Examples
Equality	<code>3 == var1</code>
Inequality	<code>var1 != 4</code>
Strict equality	<code>3 === var1</code>
Strict inequality	<code>var1 !== "3"</code>
Greater than	<code>var2 &gt; var1</code>
Greater or equal	<code>var2 &gt;= var1</code>
Less than	<code>var1 &lt; var2</code>
Less or equal	<code>var1 &lt;= var2</code>

# 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

# 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";
```

# 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 and 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.

# 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: No text provided
```

# 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;
```

# 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));
```

# 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 myCar = {  
  maker: "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";  
  },  
};
```

# 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;  
    }  
}
```

# 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 with \*

- 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";
```

# Lab 1 - Building a Media Library Application in JavaScript Using OOP

## Objective:

Participants will develop a media library management application that supports different types of media such as books, films, and magazines. The application will allow adding, deleting, and searching for media in the library using the concepts of Object-Oriented Programming.

### 1. Class Modeling:

- Create a base class `Media` with common properties such as `title`, `author`, and `year`.
- Derive specific subclasses like `Book`, `Film`, and `Magazine` by adding properties unique to each media type, for example, page count for books and duration for films.

### 2. Library Functionality:

- Implement a `Library` class that manages a collection of media.
- Add methods to `addMedia`, `removeMedia`, and `findMedia` by title or author.

### 3. Interaction Management:

- Use `console.log` to interact with the user: display options, receive commands, and show the results of actions.
- Add functionality to load and save the library's state in a JSON file (if the environment allows, otherwise simulate this feature).

# Promises, Async Await

# Asynchronous

- JavaScript is **synchronous** by default and is run on a **single thread**.
- Computers are **asynchronous** by design.
- Asynchronous means that things can happen **independently of the main program flow**.

# 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

JavaScript introduced several features that help us write asynchronous code without 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));
```


# 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();  
  // ...  
}
```

# 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  
}
```



## **Lab 2 Topic: Enhancing a Media Library Application in JavaScript with Asynchronous Programming**

Building upon the initial media library management application created using Object-Oriented Programming (OOP), participants will now integrate asynchronous programming techniques to manage data operations more effectively. This enhancement will focus on adding, updating, and retrieving media entries asynchronously.

# TypeScript

# Table of Contents

- |                        |                      |
|------------------------|----------------------|
| 1. Introduction        | 6. Classes           |
| 2. tsconfig.json       | 7. Type manipulation |
| 3. Types               | 8. Utility types     |
| 4. Functions in detail | 9. Decorators        |
| 5. Object type         |                      |

# Introduction

# TypeScript history

TypeScript is a **superset** of JavaScript developed by Microsoft in 2012

TypeScript adds static typing, classes, interfaces, and other object-oriented features

It became popular thanks to its adoption by the Angular framework, which made it its core programming language.

# TypeScript goals

The language is designed to improve productivity by providing better error checking and IDE support

It is used both on the client and server sides

Most JS frameworks support it by default

# Installation

```
npm install -g typescript
```

```
tsc main.ts
```

# tsconfig.json



# Overview

The presence of a `tsconfig.json` file in a directory indicates that this directory is the root of a TypeScript project

The `tsconfig.json` file specifies the root files and the options needed to compile the project

# Configuration file

- `compilerOptions`: configuration of TypeScript behavior
- `include`: Location of .ts files to compile
- `exclude`: Location of .ts files to skip

```
{  
  "compilerOptions": {  
    "module": "system"  
  },  
  "include": [ "src/**/*.ts" ],  
  "exclude": [ "**/*.spec.ts" ]  
}
```

# Types

# Primitive types

- `string`
- `number`
- `boolean`

```
let firstName: string = "Toto";  
const age: number = 20;  
let isAdult: boolean = age >= 18;
```

# Arrays

- `string[]`
- `Array<number>`

```
const firstNames: string[] = ["Toto", "Titi"];  
const ages = (Array<number> = [1, 5, 10]);
```

# any

A variable of type `any` can be assigned with anything.

The `any` type is the most permissive type in TypeScript. It bypasses type checking, and can be convenient to avoid writing a long, specific type to convince TypeScript that a line of code is valid.

Because this comes at the cost of losing **type safety**, you should try to avoid using `any` unless necessary

# unknown

`unknown` also represents any value, but it is safer than `any`: type checking or type assertions remains necessary to operate on an unknown variable

```
let anyVar: any;  
let unknownVar: unknown;  
  
anyVar.someMethod();           // OK, skips type checking  
unknownVar.someMethod();       // not OK! We don't know that this method exists
```

# Functions

It is possible to add types to function parameters

```
function sayHello(name: string) {  
  console.log(`Hello, ${name.toUpperCase()}`);  
}
```

Function returns also have their own types

```
function getRandomInt(min: number, max: number): number {  
  return Math.floor(Math.random() * (max - min)) + min;  
}
```



# Promises

The `Promise` type is a **generic** type: we can pass the expected return type

```
async function getFavoriteNumber(): Promise<number> {  
    return 26;  
}
```

# Object types

To type an object, add a type to each of its properties

```
function printCoord(pt: { x: number, y: number }) {}  
function printCoord(user: { first: string, last?: string }) {}
```

# Union types

It is possible to **combine** types using the `union` type

The union type consists of at least two types, and the passed value can be any of those types

To create a union type, the `|` character is used

```
let id: number | string;  
const printId = (id: number | string) => {};
```

# Working with union types

TypeScript will only allow an operation if it is valid for **every** member of the union. The methods that are specific to only some of the union types will only be accessible after type checking: this process is called **narrowing**

```
function printId(id: number | string) {  
  if (typeof id === "string") {  
    // id is of type string  
    console.log(id.toUpperCase());  
  } else {  
    // id is of type number  
    console.log(id);  
  }  
}
```

# Type aliases

You can use a **type** alias to create a new name to refer to a type

```
type Name = string;  
type ID = number | string;
```

# Interfaces

Interfaces are another way to name an object type

TypeScript's type checking focuses on the **shape** that values have (**duck typing**)

```
interface Point {  
  x: number;  
  y: number;  
}  
function printCoord(pt: Point) {  
  console.log("The coordinate's x value is " + pt.x);  
  console.log("The coordinate's y value is " + pt.y);  
}  
printCoord({ x: 100, y: 100 });
```

# Differences between type and interface

Type aliases and interfaces are very similar

The main distinction is that a type cannot be reopened to add new properties, whereas an interface is always extendable

- Primitive Type: to create an alias for a primitive type, `type` is preferred
- Union: for a union, `type` is preferred
- Object: an `interface` is more suitable for representing an object
- Type Extension: an `interface` is more suitable for extending a type

# Type assertion

Type assertion allows you to specify a type that TypeScript cannot infer by itself with the keyword `as`, thus enabling you to use the methods and properties of this type

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

This type is removed during compilation: no exception will be thrown at runtime if the specified type is incorrect



# Literal types

Literal types allow you to specify the **exact value** that a string, number, or boolean must have

String literal types can be combined with union types, type guards, and type aliases to achieve behavior similar to an `enum`

```
let x: "hello" = "hello";  
function printText(s: string, alignment: "left" | "right" | "center");
```

# Null and undefined

The `strictNullChecks` option will raise a type error when a value might be null or undefined where a definite value is expected. This ensures that the value exists before using methods or properties on it

```
function sayHi(name: string | null) {  
  console.log("Hello, " + name.toUpperCase());  
  // ✗ Error: 'name' is possibly 'null'.  
}
```

```
function sayHi(name: string | null) {  
  if (name === null) {  
    doSomething();  
  } else {  
    console.log("Hello, " + name.toUpperCase());  
  } // ✓ OK !  
}
```

# Lab 3 Union Type

1. Create an interface `User` with the following properties: name, age, occupation
2. Create an interface `Admin` with the following properties: name, age, role
3. Create an array containing the following people (create a `Person` type):

```
{  
  name: 'Toto Dupont',  
  age: 35,  
  occupation: 'Postman'  
}  
  
{  
  name: 'Jeanne Doe',  
  age: 25,  
  role: 'Admin'  
}...
```

4. Display the name and age of each of these people.

# Functions

# Typing functions

It is also possible to type functions. The simplest way is with a **function type expression**:

```
function greeter(fn: (a: string) => void) {  
    // ...  
}  
  
type GreetFunction = (a: string) => void;  
  
function greeter(fn: GreetFunction) {  
    // ...  
}
```

# Typing functions

Functions may also have properties. A **call signature** has a slightly different syntax, and allows for declaring function properties in an object type:

```
type GreetFunction = {  
  (a: string): void; // call signature, (a: string) => void;  
  someProperty: string;  
};
```

# Construct signature

To type **constructors**, simply add the `new` keyword in the type declaration

```
type SomeConstructor = {  
    new(s: string): SomeObject,  
};  
function fn(ctor: SomeConstructor) {  
    return new ctor("hello");  
}
```

# Generic functions

A **generic function** allows you to manipulate different types, it is a form of polymorphism in OOP

Declaring a generic function is done by adding a type parameter between `<>`

```
function firstElement<Type>(arr: Type[]): Type | undefined {  
    return arr[0];  
}
```



# Generic type constraints

**Constraints** on generic types allow you to add limitations on the passed type parameter

In the following example, the type must implement the length property:

```
function longest<Type extends { length: number }>(a: Type, b: Type) {  
    if (a.length >= b.length) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

# Function overloading

Some JavaScript functions can have the same name but different parameters (ex: `Date`)

TypeScript allows **function signature overloading**. To achieve this, you need to add the function **signatures** followed by its **implementation**

```
function makeDate(timestamp: number): Date;  
function makeDate(m: number, d: number, y: number): Date;  
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {  
    // Function details  
}
```

## Other return types

- `void`: the return type of functions that do not return a value
- `object`: represents a non-primitive type with properties, different from `Object` or `{}`
- `never`: represents the type of values that never occur. For example, indicates that a function throws an exception or never returns
- `Function`: represents a function that has the `bind`, `call`, `apply` properties and can be called

# Parameter destructuring

TypeScript parameter destructuring works the same way as in JavaScript, with the addition of specifying the types

```
function sum({ a, b, c }: { a: number, b: number, c: number }) {  
    console.log(a + b + c);  
}  
  
// OR  
type ABC = { a: number, b: number, c: number };  
function sum({ a, b, c }: ABC) {  
    console.log(a + b + c);  
}
```

## Lab 4

### 1. Creating Overloaded Functions:

- Define a function `formatDate` that accepts either a `Date` object or a string representing a date, and an optional boolean parameter `includeTime`.
  - If `includeTime` is true, the function should return the date and time formatted as "YYYY-MM-DD HH:MM". If false or omitted, it should return the date formatted as "YYYY-MM-DD".
  - Implement function overloads to correctly type-check the parameters based on the input type.

# Lab 4

## 2. Using Optional Parameters:

- Write a function greet that takes a name string and an optional age number.
  - The function should return a greeting string. If the age is provided, it should include the age in the greeting, such as "Hello, John! You are 30 years old.". If the age is not provided, it should return "Hello, John!".

## Lab 4

### 3. Implementing Generic Functions:

- Create a generic function `getFirst` that accepts an array of any type and returns the first element of the array.
- Ensure that the type of the returned element is the same as the type of the elements in the input array.

## Lab 4

### 4. Utilizing Type Guards:

- Define a function `isString` that takes an input parameter of type `any` and returns a boolean indicating whether the provided input is a string.
- Use this function in another function `processInput` that accepts a string or number. The `processInput` function should print the input directly if it is a number, or print the uppercased version if it is a string.



# Object type

# Object types

In TypeScript, there are different ways to group data via objects:

- Anonymous objects
- Interfaces
- Type aliases

# Optional properties

Most objects can have optional properties. In that case, the ? character is added after their type

```
interface PaintOptions {  
  shape: Shape;  
  xPos?: number;  
  yPos?: number;  
}
```

# Readonly properties

Read-only properties can only be assigned during object creation

```
interface Person {  
  readonly name: string;  
}  
  
let bob: Person = {  
  name: "Bob"  
}  
  
bob.name = "Mike"; // Cannot assign to 'name' because it is a read-only property.
```

# Type extension

Type and interface extension is a way to create a new, more general type that inherit properties and methods from the base type

```
interface BasicAddress {  
    name?: string;  
    street: string;  
    city: string;  
    country: string;  
    postalCode: string;  
}  
  
interface AddressWithUnit extends BasicAddress {  
    unit: string;  
}
```

# Multiple extensions

You can extend multiple interfaces:

```
interface Colorful {  
  color: string;  
}  
  
interface Circle {  
  radius: number;  
}  
  
interface ColorfulCircle extends Colorful, Circle {}
```

# Type intersection

The `&` character allows to create a combination of multiple types, similar to the union with `|`

Thus, the code from the previous slide can also be written as:

```
interface Colorful {  
  color: string;  
}  
interface Circle {  
  radius: number;  
}  
  
type ColorfulCircle = Colorful & Circle;
```

# Generic object types

In TypeScript, a generic object is an object that can work with different types

Like generic functions, a generic object is declared with 

```
interface Box<Type> {  
  contents: Type;  
}  
  
let box: Box<string>;
```



# Tuple type

The tuple type is a kind of array that has a predefined length, and predefined types specified in each index location

```
type StringNumberPair = [string, number];  
const myStringNumber: StringNumberPair = ["user", 12];
```

`StringNumberPair` is a tuple type of two elements, with the first element being a `string`, and the second one a `number`, in that order.

# Classes

# Classes

TypeScript supports JavaScript classes, introduced in 2015 with ES6.

This support allows typing members, methods, and return types while adding new behaviors.

# Visibility

There are 3 visibility modifiers for class members:

- `public`: members are accessible outside the class (default)
- `protected`: members are only accessible within the class and its subclasses
- `private`: members are only accessible within the class

```
class User {  
    private id: number;  
    protected username: string;  
    public lastConnection: Date;  
}
```

# Members

Class members work the same way as other types in TypeScript

```
class User {  
  id = 0; // the type will be number  
  readonly username; // the member is initialized in the constructor  
  age; // type any  
}
```

# Constructor

Just like functions, it is possible to overload the constructor to have multiple signatures

```
class Point {  
    // Overloads  
    constructor(x: number, y: string);  
    constructor(s: string);  
    constructor(xs: any, y?: any) {  
        // To be implemented  
    }  
}
```

# Methods

Methods work the same way as functions in TypeScript

```
class Point {  
  x = 10;  
  y = 10;  
  
  scale(n: number): void {  
    this.x *= n;  
    this.y *= n;  
  }  
}
```

# Generic classes

Classes, like interfaces, can be generic

The constraint system is the same as for interfaces

```
class Box<Type> {  
  contents: Type;  
  constructor(value: Type) {  
    this.contents = value;  
  }  
}  
  
const b = new Box("hello!");
```



## Lab 5 Generic Exercise

Create a generic stack class that allows:

- Pushing elements onto the stack.
- Popping the last element off the stack.
- Retrieving an element by its ID.

The elements of the stack will be stored in an array.

The stack will only work with elements that have an ID.

# Big Lab 1

- This lab aims to enhance participants' understanding of TypeScript by applying advanced features to build a robust and scalable data management system. The focus is on utilizing generics, interfaces, and classes to create a flexible system that can handle various data types and operations efficiently.

## 1. Defining Types and Generic Interfaces

- **ID and Category:**

- Define `ID` as a type alias for `number`, ensuring that all identifiers in the system are numbers.
- Define `Category` as a union type to restrict item categorization to 'Electronics', 'Clothing', or 'Food', thus enforcing consistency.

- **Item, Product, and InventoryItem Interfaces:**

- `Item<T>`: A base interface that includes an `id`, `category`, and a generic `metadata` field which can contain any type of additional data.
- `Product<T>`: Extends `Item<T>` by adding `name` and `price` fields. This interface can handle products with diverse additional data (metadata).
- `InventoryItem<T>`: Further extends `Product<T>` with a `quantity` field, tailored for inventory management.

# Big Lab 1

## 2. Implementing Generic Classes

- **Inventory Class:**

- This class manages a collection of items of type `Item<T>`, providing methods to add, update, and find items within an inventory.
- Features methods such as `addItem(item: Item<T>)`, `updateItem(id: ID, updateData: Partial<Item<T>>)` for updates, and `findItemsByCategory(category: Category)` for retrieval based on category.

## 3. Advanced Interface Utilization and Grouping Techniques

- **Groupable Interface and GroupManager Class:**

- The `Groupable` interface requires a `getKey()` method that specifies how items are grouped.
- `GroupManager<T>` uses this interface to sort items into a `Map<string, T[]>`, grouping them by keys obtained from `getKey()`.

# Big Lab 1

- **Flexible Data Adjustments:**

- Define a generic type `Adjustment<T>` that allows adjustments either as a direct value (number) or a percentage (string).
- Implement `adjustPrices(adjustment: Adjustment<number>)` in the `Inventory<T>` class to update prices based on the type of adjustment provided, demonstrating type safety and flexibility.

# Type manipulation

# keyof operator

The `keyof` operator takes an object type and produces a string or numeric literal union of its keys

This operator is usually used with mapped types

```
type Point = {  
  x: number;  
  y: number  
};  
  
type P = keyof Point; // type P = "x" | "y":
```

# typeof operator

`typeof` is a JavaScript operator that returns the type of a variable as a string.

In TypeScript, the `typeof` operator has additional features, and allows you to work with types : extending and merging types, type-checking, assigning the type of a variable to another variable...

```
let user1 = {  
  name: "Bob",  
  email: "bob@email.com"  
};  
  
// Get the type of 'user1'  
// and call it 'User'  
type User = typeof user1;  
  
// type User = {  
//   name: string;  
//   email: string;  
// }
```

# Conditional types

It is possible to assign a type based on a condition using a ternary as follows:

```
SomeType extends OtherType ? TrueType : FalseType;
```

```
interface Animal {  
    live(): void;  
}  
interface Dog extends Animal {  
    woof(): void;  
}  
  
type Example1 = Dog extends Animal ? number : string;
```



# Mapped types

A mapped type is a generic type that uses a union of PropertyKeys (often created through a `keyof`) to iterate through keys to create a type

# Mapped types

Here the `OptionsFlags` type will transform all passed properties into boolean values:

```
type OptionsFlags<Type> = {  
  [Property in keyof Type]: boolean;  
};  
  
type Features = {  
  darkMode: () => void,  
  newUserProfile: () => void,  
};  
  
type FeatureOptions = OptionsFlags<Features>;
```

# Utility types

# Utility types

TypeScript provides several utility types to facilitate common type transformations

# Partial<Type>

Creates a type whose properties are all optional

```
interface Todo {  
  title: string;  
  description: string;  
}  
  
function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {  
  return { ...todo, ...fieldsToUpdate };  
}
```

# Required<Type>

Creates a type whose properties are all required. The opposite of `Partial`.

```
interface Props {  
  a?: number;  
  b?: string;  
}  
  
const obj: Props = { a: 5 };  
  
const obj2: Required<Props> = { a: 5 };
```

# Pick<Type>

Creates a type by selecting some properties from another existing type

```
interface Todo {  
  title: string;  
  description: string;  
  completed: boolean;  
  createdAt: number;  
}  
  
type TodoPreview = Pick<Todo , "title" | "completed" >;  
  
const todo: TodoPreview = {  
  title: "Clean room",  
  completed: false,  
};
```

# Omit<Type>

Constructs a type by selecting all properties from another type and then removing some of them. The opposite of `Pick`.

```
interface Todo {  
  title: string;  
  description: string;  
  completed: boolean;  
  createdAt: number;  
}
```

```
type TodoPreview = Omit<Todo, "description">;  
  
const todo: TodoPreview = {  
  title: "Clean room",  
  completed: false,  
  createdAt: 1615544252770,  
};
```



# Decorators

# Definition

Decorators provide a way to add new behaviors to classes, methods, and properties by placing them in wrappers that implement these behaviors.

Decorators use the form `@expression`, where the expression must evaluate a function that will be called at runtime with information about the decorated element.

This function takes two arguments: the `target` it is applied to, and the `context` provided to the decorator.

```
function verySimpleDecorator(target: Function, context: DecoratorContext): void {  
    console.log('Hi! I am a decorator');  
}  
  
@verySimpleDecorator  
class SomeClass {}
```

# Decorator example

Here is an example of a decorator that adds logging to a method

```
function log(  
    originalMethod: Function,  
    context: ClassMethodDecoratorContext,  
) : any {  
    const methodName = context.name.toString();  
    return function newMethod(this: any, ...args: any[]) {  
        console.log(`${new Date().toLocaleString()}: Method '${methodName}' was called with arguments `, args);  
        return originalMethod.call(this, args);  
    };  
};
```

# Experimental decorators (legacy)

Decorators were introduced in Typescript 5.0. In anterior versions, only experimental decorators were supported.

To enable them, set the `--experimentalDecorators` flag, or enable the `experimentalDecorators` compiler option in `tsconfig.json`:

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

## Experimental decorators have a different syntax

```
function log(  
  target: any,  
  name: string,  
  descriptor: PropertyDescriptor  
) {  
  const original = descriptor.value;  
  descriptor.value = function (...args: any[]) {  
    console.log(`${new Date().toLocaleString()}: Method '${name}' was called with arguments `, args);  
    const result = original.apply(this, args);  
    return result;  
  };  
  return descriptor;  
}
```

This implementation is still used, but can now be considered legacy

# Decorator factory

To customize how a decorator is applied to a declaration, you can write a **decorator factory**. It is a function that returns a decorator.

```
function decoratorFactory(arg: any) {  
    return function decorator(targetFuction: Function) {  
        doSomething()  
    };  
}  
  
@decoratorFactory(param)  
class C {}
```

# Decorator factory example

Here is an example of a validation decorator factory:

```
function lengthValidator(min: number, max: number) { // validation decorator factory
  return (originalMethod: Function): any => { // validation decorator
    return function (this: any, arg: string) { // decorated target
      if (arg.length < min || arg.length > max)
        throw new Error(`Argument must be between ${min} and ${max} characters in length`);
      return originalMethod.call(this, arg);
    };
  };
}
```



# Error handling

- In TypeScript, as in JavaScript, exceptions are used to handle errors or unexpected behaviors in the code. Here's a concrete example to illustrate how you can use exceptions in TypeScript to handle errors when running functions that may encounter problems

```
function convertToNumber(input: string): number {  
    let number = parseFloat(input);  
    if (isNaN(number)) {  
        throw new Error("Input must be a valid number");  
    }  
    return number;  
}  
  
function processUserData(input: string): void {  
    try {  
        let number = convertToNumber(input);  
        console.log(`The number is: ${number}.`);  
    } catch (error) {  
        console.error(`An error occurred: ${error.message}`);  
    }  
}  
  
processUserData("42");  
processUserData("abc");
```

# Lab 6

Implement a `CatchAndLog` decorator for the `PaymentProcessor` class that manages errors in the `processPayment` method.

## Specifications:

### 1. Decorator Options:

- Accepts `logLevel` ('debug' or 'error') and `notifyOnError` (boolean).
- Adds error handling and logging to methods.

### 2. Usage:

- Apply to `processPayment`, which throws an error for non-positive payment amounts.

```
class PaymentProcessor {
  processPayment(amount: number, accountID: string) {
    if (amount <= 0) {
      throw new Error("Invalid amount for payment processing");
    }
    console.log(`Processing payment of ${amount} for account ${accountID}`);
  }
}

const processor = new PaymentProcessor();
processor.processPayment(0, "12345");
```

# .d.ts File

The `.d.ts` files in TypeScript are called "declaration files." Their primary role is to describe the shape and the type information of JavaScript code without providing implementations. This is particularly useful when working with JavaScript libraries in a TypeScript project.

1. **Type Checking:** Declaration files provide type information for JavaScript libraries so that TypeScript can perform type checking against the library usage in your code.
2. **Autocompletion:** When using IDEs or editors that support TypeScript, `.d.ts` files enable intelligent code completion (IntelliSense) for JavaScript libraries.

```
export function greet(name: string): string;
```

- **Manual Inclusion:** Include the `.d.ts` file in your project and ensure it is referenced by your TypeScript configuration (`tsconfig.json`) or imported directly in your TypeScript files using a reference directive.
- **DefinitelyTyped Repository:** Use types from the DefinitelyTyped repository (`@types` packages), which is a community-led collection of high-quality declaration files for thousands of JavaScript libraries.

# Test typescript code

To test TypeScript code, you can use various tools and frameworks that facilitate unit testing, integration testing, and other types of testing.

## 1. Setup the Testing Environment

```
npm install --save-dev jest ts-jest @types/jest
```

## 2. Configure Jest for TypeScript

jest.config.js

```
module.exports = {  
  preset: 'ts-jest',  
  testEnvironment: 'node',  
  testMatch: ['**/?(*.)(spec|test).[tj]s?(x)'], // Matches test files  
};
```

# Test typescript code

## 3. Write Tests

- `.test.ts` or `.spec.ts` extensions.

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

- (`sum.test.ts`)

```
import { sum } from './sum';  
  
test('adds 1 + 2 to equal 3', () => {  
    expect(sum(1, 2)).toBe(3);  
});  
  
test('adds 5 + 5 to equal 10', () => {  
    expect(sum(5, 5)).toBe(10);  
});
```

# Test typescript code

## 4. Running Tests

```
"scripts": {  
  "test": "jest"  
}
```

```
npm test
```

# Test typescript code

## 5. Advanced Configurations and Mocking

- **Mocking:** Jest provides powerful mocking capabilities. You can use `jest.mock()` to mock modules or `jest.fn()` to mock individual functions.
- **Setup and Teardown:** Use `beforeEach`, `afterEach`, `beforeAll`, and `afterAll` hooks to set up preconditions and clean up after tests.

# Final Lab

- The objective of this assignment is to migrate an event management application from JavaScript to TypeScript. The trainee will convert the existing code, improve modularity and structure, and implement decorators for logging and exception handling.

## Context:

You are provided with an event management application that allows adding, listing, deleting events, and displaying event details. Currently, the application is written in JavaScript with a monolithic code structure. Your task is to refactor this application using TypeScript while enhancing code modularity and quality. Additionally, you will implement decorators to handle logging and exceptions.



# Final Lab

## Requirements:

### 1. Type Definitions:

- Identify the various objects and data structures used in the existing code.
- Define appropriate TypeScript interfaces to represent these objects, such as creating an interface for events (`Event`).

### 2. Conversion to TypeScript:

- Convert all existing JavaScript files to TypeScript.
- Add proper types for all variables, functions, and objects.
- Ensure that the application compiles without TypeScript errors.

### 3. Modularity:

- Separate different parts of the application into distinct modules, such as data models, services, and entry points.
- Create an interface for the event management service (`IEventService`) and implement this interface in a concrete class (`EventService`).

# Final Lab

## Requirements:

### 4. Code Structuring:

- Create a `models` folder for data model definitions (e.g., `Event`).
- Create a `services` folder for business services, including interfaces and implementations.
- Ensure a clear separation of concerns: data models, business logic, and CRUD operations should be distinctly separated.

### 5. Logging and Exception Handling with Decorators:

- Implement decorators for logging method calls and exceptions.
- The logging decorator should log method entry, exit, and parameters.
- The exception handling decorator should catch and log exceptions, ensuring the application does not crash unexpectedly.

### 6. Testing:

- Implement unit tests to verify the correct functioning of various features of the application (adding, deleting, listing events, etc.).
- Use a testing framework like Jest to automate the tests.

# Final Lab

## Instructions:

### 1. Step 1: Type Definitions

- Identify the data structures for events and services.
- Define the necessary types and interfaces.

### 2. Step 2: Conversion to TypeScript

- Set up a TypeScript project and configure the TypeScript compiler (tsconfig.json).
- Migrate the existing JavaScript code to TypeScript by adding types and refactoring as needed.

### 3. Step 3: Modularity and Structuring

- Organize the project into modules: models, services, entry point.
- Implement the `IService` interface and the `Service` class.

### 4. Step 4: Logging and Exception Handling with Decorators

- Implement a logging decorator that logs method names, parameters, and execution results.
- Implement an exception handling decorator that catches exceptions and logs them without stopping the application.

# Final Lab

## Instructions:

### 5. Step 5: Testing

- Set up a testing environment with Jest.
- Write unit tests for all key functionalities of the application.
- Run the tests and ensure all tests pass successfully.

**Thank you for your attention**

**Any questions?**