



# **TypeScript Review**

**CS572 Modern Web Application**

**Maharishi International University**

**Department of Computer Science**

**Associate Professor Asaad Saad**

# Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

# Why TypeScript?

One of the great things about type-checking is that:

1. It helps write safe code because it can **prevent bugs at compile time**.
2. Compilers can improve and run the code **faster**.

Types are **optional** in TypeScript (*because they can be inferred*).

# TypeScript Compiler

TypeScript compiles into vanilla JavaScript.

A TypeScript code is written in a file with **.ts** extension and then compiled into JavaScript using the TypeScript compiler.

Note: if you desire to transpile the ts files to js, you may use the following command: **npx tsc**

# TypeScript Development Workspace

Go to [nodejs.org](https://nodejs.org) and download the **stable LTS version** of node.

**Note:** to transpile and run TypeScript files in node on the fly, you will need a few dependencies, in a blank folder, run the following commands:

```
// create package.json
```

```
npm init --y
```

```
// install Node types as development dependency
```

```
npm i @types/node -D
```

```
// run your script
```

```
npm run tsx watch file.ts
```

```
// create .gitignore and exclude node_modules from being tracked by Git
```

# Modules

Any file is considered a module, and everything inside the file is private, until we explicitly export it, we have two kinds of exports:

- **export default** (can be used once) *default export*
- **export** (can be used multiple times) *named export*

To import what is explicitly exported we use:

**import** defaultExport, {namedExport1, namedExport2} **from** './file.js'

# Type Annotations

We can specify the type using **:type** after the name of the variable, parameter or property.

TypeScript includes all the primitive types of JavaScript- number, string and boolean.

```
const grade: number = 90; // number variable
const name: string = "Asaad"; // string variable
const isFun: boolean = true; // Boolean variable
```

# Type Inference

It is not mandatory to annotate types in TypeScript, as it **infers types** of variables when there is no explicit information available in the form of type annotations.

```
let text = "some text";  
text = 123; // Type '123' is not assignable to type 'string'
```



# Any

Never ever use the type **any** especially to silence an error or when you do not have prior knowledge about the type of some variables.

```
let something: any = 'Asaad';  
something = 569;  
something = true;
```

```
let data: any[] = ["Asaad", 569, true];
```

# Union Type

Union type allows us to combine more than one data type.

`(type1 | type2 | type3 | .. | typeN)`

```
let course: (string | number);  
let data: string | number;
```

# Enum

Enums allow us to declare **a set of named Constants**, a collection of related values that can be numeric or string values.

Enum values start from zero and increment by 1 for each member.

Enum in TypeScript supports **reverse mapping**.

```
enum Technologies {  
    Angular,  
    React,  
    ReactNative  
}  
  
// Technologies.React; returns 1  
// Technologies["React"]; returns 1  
// Technologies[0]; returns Angular
```

```
console.log(Technologies);  
  
{  
    '0': 'Angular',  
    '1': 'React',  
    '2': 'ReactNative',  
    Angular: 0,  
    React: 1,  
    ReactNative: 2  
}
```

# Array

There are two ways to declare an array:

## 1. Using **square brackets**

```
let values: number[] = [12, 24, 48];
```

## 2. Using a **generic array type**, `Array<elementType>`

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

You can always initialize an array with many data types elements, but you will not get the advantage of TypeScript's type system.

# There are two ways to do Type Assertion

## 1. Using the angular bracket <> syntax

```
let code: any = 123;  
let courseCode = <number> code;
```

## 2. Using **as** keyword

```
let code: any = 123;  
let courseCode = code as number;
```

**Note:** The assertion from type S to T succeeds if either S is a subtype of T or T is a subtype of S. Asserting to unknown (or any) is compatible with all types.

# Object Types

We may represent object types as an **interface** or a **type alias**:

```
interface Person {  
  name: string;  
  age: number;  
}
```

OR

```
type Person = {  
  name: string;  
  age: number;  
};
```



```
function greet(person: Person) {  
  return "Hello " + person.name;  
}
```

# Record

```
interface City {  
  name: string;  
  population: number;  
}
```

```
type County = "Jefferson" | "Johnson" | "Keokuk";
```

```
const state: Record<County, City[]> = {  
  Jefferson: [{ name: "Fairfield", population: 9464 }],  
  Johnson: [{ name: "Iowa City", population: 75233 },  
            { name: "Cedar Rapids", population: 136429 }],  
  Keokuk: [{ name: "Richland", population: 531 }],  
};
```

```
state.Jefferson;
```

# Combine Types

```
type Person = {  
    name: string;  
    age: number;  
};
```

```
type Tech = {  
    phone: string;  
    laptop: string;  
};
```

```
type TechPerson = Person & Tech;
```

```
const asaad: TechPerson = { name: "Asaad", age: 0, phone: "Pixel", laptop: "MacBook Pro" };
```



# Combine Interfaces

```
interface Person {  
    name: string;  
    age: number;  
}
```

```
interface Tech {  
    phone: string;  
    laptop: string;  
}
```

```
const asaad: Person & Tech = { name: "Asaad", age: 0, phone: "Pixel", laptop: "MacBook Pro" };
```

# Class with Constructor

```
class Person {  
  name: string = '';  
  constructor(name: string) {  
    this.name = name;  
  }  
  greet() { console.log(`hi ` + this.name); }  
}
```

```
const asaad = new Person("Asaad"); // asaad = {name: "Asaad", greet: Function }  
asaad.greet(); // hi Asaad
```

# TypeScript: Access Modifier in Constructor

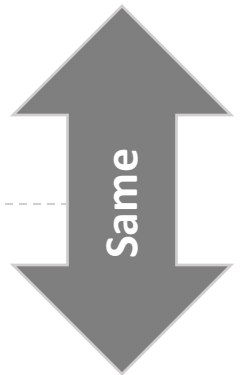
```
class Person {  
    name: string = '';  
    constructor(name: string) { this.name = name }  
}
```

```
const asaad = new Person("Asaad"); // asaad = { name: "Asaad" }
```

---

```
class Person {  
    constructor(public name: string) {}  
}
```

```
const asaad = new Person("Asaad"); // asaad = { name: "Asaad" }
```



# Access Modifiers

Access modifiers are set for encapsulation and are used to control class members visibility.

There are two access modifiers in Javascript: **public**, **private**

There are three access modifiers in Typescript: **public**, **private** and **protected**.

# public

By default, all members of a class in TypeScript are public. All the public members can be accessed without any restrictions.

JS does not have **public** keyword, all members are considered public. TS has **public** keyword

# Public Properties

```
class Person {  
    public first: string = 'Asaad'; // TS  
    last: string = 'Saad'; // JS and TS  
}
```

```
const asaad = new Person();
```

```
console.log(asaad.first); // Asaad  
console.log(asaad.last); // Saad
```

# private

The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class (the instance or extended classes).

JS has a `#` native accessor to mark private properties, while TS has the **private** keyword.

# Private Properties

```
class Person {  
    private first: string = 'Asaad'; // TS  
    #last: string = 'Saad'; // JS and TS  
}
```

```
const asaad = new Person();
```

```
console.log(asaad.first); // Error  
console.log(asaad.last); // Error
```



# protected

The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes by inheritance.

```
class Person {  
    protected name: string = 'Asaad'; // TS  
}
```

```
const asaad = new Person();
```

```
console.log(asaad.name); // Error
```

# Generics

Generics offer a way to create **reusable components**. Generics provide a way to make components work with any data type and not be restricted to one data type.

Generic types can also be used with other non-generic types.

The implementation of generics gives us the ability to pass a range of types to a component, adding an extra layer of abstraction and re-usability to your code.

Generics can be applied to **functions**, **interfaces**, and **classes** in Typescript.

# Generic Function

```
function last_element<T>(arr: T[]): T {  
    return arr.at(-1) as T  
}
```

You may need: `npm i @types/node -D`

The generic type parameter is specified in **angle brackets** after the name of the function. T can be replaced with any valid name.

```
const last = last_element<number>([1, 2, 3])  
const last = last_element<string>(['a', 'b', 'c'])
```

# Generic Interface

```
interface Todo<TCompleted, TPending> {  
    title: string;  
    completed: TCompleted;  
    pending: TPending;  
}
```

```
const todo1: Todo<boolean, number> = { title: "hello", completed: true, pending: 0 };
```

```
const todo2: Todo<number, number> = { title: "hello", completed: 1, pending: 0 };
```

# Generic Class

```
class Item<T> {  
    constructor(public data: T) {}  
}
```

```
const item1 = new Item<boolean>(true); // item1 = { data: true }  
const item2 = new Item<string>("Hello"); // item1 = { data: "Hello" }
```

# Utility Types

```
interface Person {  
  firstname: string;  
  lastname: string;  
  age: number;  
  height: number;  
  weight: number;  
}
```

```
type NamedPerson = Pick<Person, "firstname" | "lastname">;  
const asaad: NamedPerson = { firstname: "Asaad", lastname: "Saad" };
```

```
type SomePerson = Partial<Person>;  
const mike: SomePerson = { firstname: "Asaad", height: 180 };
```

```
type JustName = Omit<Person, "age" | "height" | "weight">;  
const theo: JustName = { firstname: "Theo", lastname: "Saad" };
```

# **Promise Review**

# Create Promise Instance

A **Promise** represents a value which may be available now, or in the future, or never.

```
const promiseInstance = new Promise(function(resolve, reject){  
    resolve(); // return value  
    reject(); // return error  
});
```

Promise object states:  
Pending, Fulfilled, Rejected

# Consume Promise Instance

```
promiseInstance.then(fn1) // when resolve(), fn1 is invoked with value  
    .then(fn2) // receives the value returned from the first callback  
    .catch(fn3) // when reject(), fn2 is invoked with error  
    .finally(fn4) // always invoke fn3
```

Both catch and finally handlers are optional, we may also use multiple .then() functions.



# Is Promise Synchronous or Asynchronous?

The promise executor function is the function you pass to the `new Promise()`. It is executed **synchronously**.

The callbacks you attach with `then`, `catch`, and `finally` are always called **asynchronously**, whether the promise is already settled or not.

# Microtask Queue

All **then** and **catch** and **finally** callback functions are pushed into the **microtask** queue which has a higher priority than the **macrotask** queue.

# async

`async` is placed before any function, it does NOT make the function run asynchronously, and it still **runs synchronously**.

`async` **returns an asynchronous value**, as it implicitly changes the returned value of the function to become a **Promise**:

- When no return statement is defined, it returns `Promise.resolve(undefined)`
- When you return a value, it returns `Promise.resolve(value)`
- When an error is thrown, it returns `Promise.reject(error)`

# async Example

```
console.log('start');

const foo = async () => {
  console.log(`hi`);
  return `foo`; // return Promise.resolve(`foo`)
};

foo().then(console.log);

console.log('end');
```

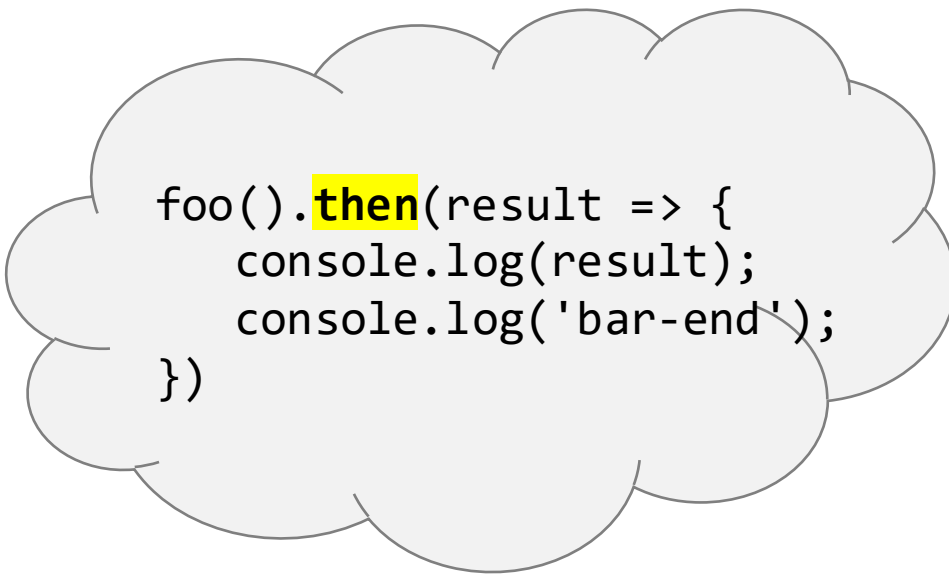
# **await**

The keyword **await** is syntactic sugar for **.then()**, it converts the execution of all lines of code that come after to become asynchronous, (within the same scope).

**await** does not pause or suspend the code execution.

# await Example

```
console.log('start');  
function foo() {  
    return new Promise(resolve => resolve(`foo`));  
}  
async function bar() {  
    console.log('bar-start');  
    let result = await foo();  
    console.log(result); // async  
    console.log('bar-end'); // async  
}  
bar();  
console.log('end');
```



```
foo().then(result => {  
    console.log(result);  
    console.log('bar-end');  
})
```

# Defensive coding

- While error handling is crucial, practicing defensive coding techniques and prevention is equally important.
- This involves validating user input, checking for null or undefined values, and implementing error checks to handle potential edge cases.
- By incorporating defensive coding practices, you can minimize the occurrence of errors and enhance the overall stability of your code.

# Errors

- **Syntax Error:** These errors occur when there is a mistake in the code's grammar or structure.
- **Reference Error:** These errors occur when you try to access a variable or function that has not been declared or is out of scope.
- **Range Error:** When a value falls outside the allowable range.
- **Type Error:** It shows up when you perform an operation on incompatible data types.
- **Custom Error:** JavaScript also allows you to create your own custom errors.



# The Try-Catch Statement

Wrap a block of code in a try block and catch potential errors in the catch block, you can prevent your program from crashing when an error occurs. Inside catch blocks: Access message, name, stack trace, etc.

```
try {  
    // Code that may throw an error  
    const result = undefinedVar + 10;  
} catch (error) {  
    // Output: An error occurred: undefinedVar is not defined  
    console.log("An error occurred:", error.message);  
}
```

# The Finally Block

The finally block is incredibly useful as it gets executed regardless of whether an error occurs or not. It's commonly used to perform cleanup operations or release resources.

```
try {  
    // Code that may throw an error  
    console.log("Inside the try block");  
} catch (error) {  
    console.log("An error occurred:", error.message);  
} finally {  
    console.log("The finally block is executed.");  
}
```

```
// Output: Inside the try block  
// Output: The finally block is executed.
```

# Error Object

The **Error** object in JavaScript plays a crucial role in error handling.

- It is an object created when an error occurs during runtime.
- We can also create it manually using the **Error** constructor:

```
const error = new Error("Something went wrong!");
```

```
console.log(error.message); // Something went wrong!
```

It contains information about the error, including its message, name, stack trace, and other properties.

# Throwing an Error

```
try {  
    // Throw an error  
    throw new Error("Something went wrong")  
} catch (error) {  
    // Output: An error occurred: Something went wrong  
    console.log("An error occurred:", error.message);  
}
```

# Creating Custom Errors

Extend **Error** to create specific error types with informative messages. This empowers you to define your own error types and provide more meaningful error messages to aid in debugging.

```
class CustomError extends Error {  
    constructor(message: string, public details: string) {  
        super(message);  
    }  
}
```

# Catching Specific Errors

Use multiple if-statement blocks with **instanceof** operator.

```
try {  
    // Code that may throw an error  
    throw new CustomError("Width error", "Width must not be 0");  
} catch (error) {  
    if (error instanceof CustomError) {  
        console.log("A custom error occurred:", error.details);  
        // Output: A custom error occurred: Width must not be 0  
    } else {  
        console.log("A generic error occurred:", error.message);  
    }  
}
```

# Asynchronous Errors

```
// Promise.reject() is equivalent to throw  
const calculate = () => Promise.reject(new Error("Something went wrong"));
```

```
try {
```

```
  ✖ calculate(); // Does not work
```

```
  } catch (error) {  
    console.log("An error occurred:", error.message);  
  }
```

# Asynchronous Errors

```
const calculate = () => Promise.reject(new Error("Something went wrong"));

try {
  calculate().catch((error) => console.log(error.message));
} catch (error) {
  console.log("An error occurred:", error.message);
}
```



# await for Asynchronous Calls

```
const calculate = () => Promise.reject(new Error("Something went wrong"));

const run = async () => {
  try {
    await calculate();
  } catch (error) {
    console.log("An error occurred:", error.message);
  }
};

run();
```

# Best Practices for Error Handling

- Use try...catch blocks.
- Log errors to the console.
- Throw **Error** objects.
- Validate user input.
- Test your code thoroughly.