CSM
breaking thought barriers

Angular Training

Session -9

Outlines

- TypeScript

TypeScript

Agenda

- What is TypeScript?
- Installation
- Hello World
- Why TypeScript?
- Basic Type
- Function & Class
- Interface
- Generic
- Enum
- Who Use TypeScript?
- Conclusion
- Q&A
- References

What is TypeScript?

Anders Hejlsberg

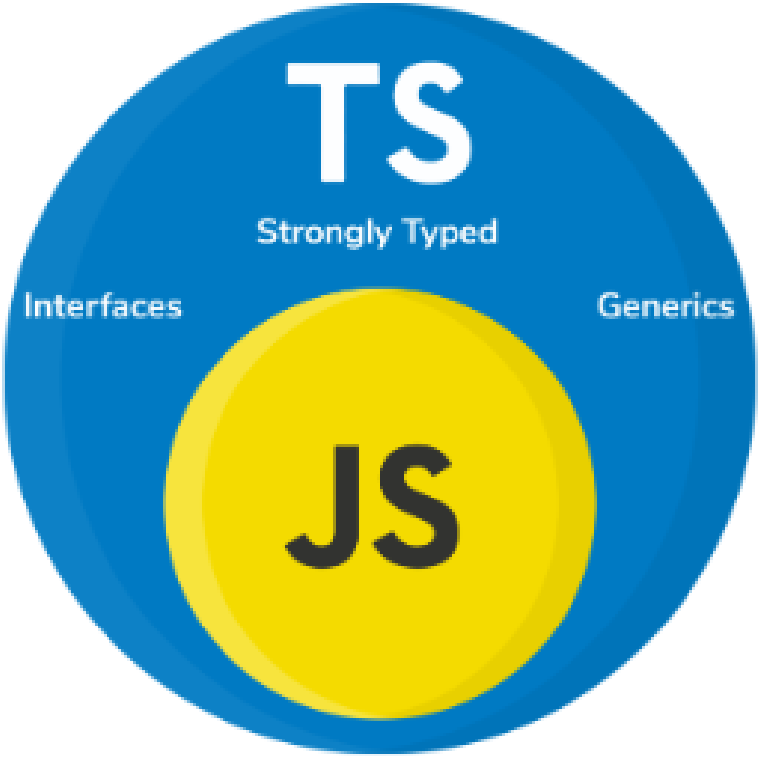
2012 , Microsoft

JavaScript Superset

Add new features & advantages to JavaScript

Language building up on JavaScript

Browser CAN’T execute it!


TS
Strongly Typed
Interfaces
Generics
JS

Typescript Vs Javascript

- TypesScript is an Object oriented programming language whereas JavaScript is a scripting language (with support for object oriented programming).
- TypeScript has static typing whereas JavaScript does not.
- TypeScript uses types and interfaces to describe how data is being used.
- TypeScript has interfaces which are a powerful way to define contracts within your code.
- TypeScript supports optional parameters for functions where JavaScript does not.

Why TypeScript?

- ✓ Transpiling allows you to generate ECMAScript
- ✓ Typescript supports JS libraries and API documentation
- ✓ Typescript introduces static typing
- ✓ Typescript uses NPM
- ✓ Typescript is easier to maintain
- ✓ Typescript makes it easier to use React, Angular, and Vue.

Installation

The following tools you need to setup to start with TypeScript:
- Node js
- TypeScript compiler
- IDE (VsCode)

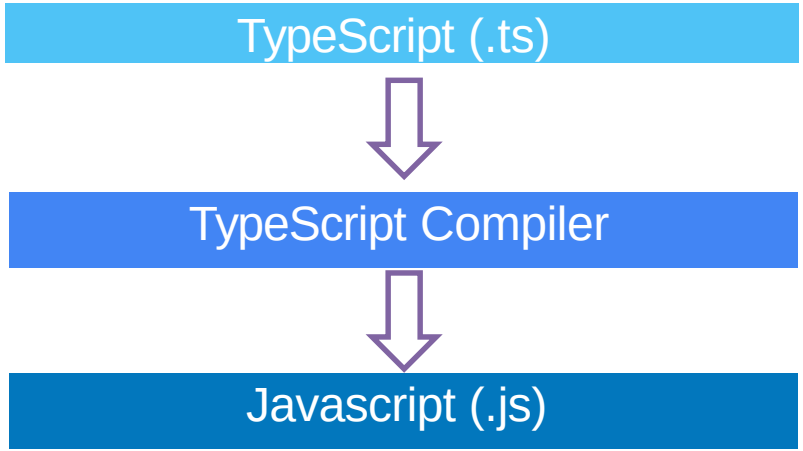
```
npm install -g typescript
```

```
tsc -v
```

```
Version 4.0.2
```

TypeScript Compilers

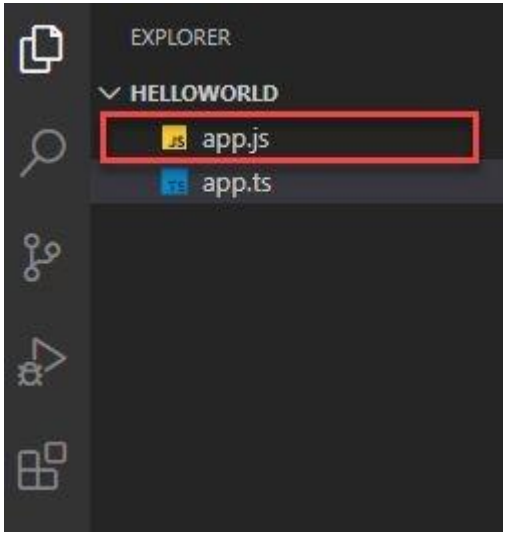
To compile TS code, we run the command tsc filename.ts. This will create a JS file of the same name, so we can eventually use it on the browser.



```ts
let message: string = 'Hello, World!';
console.log(message);
```

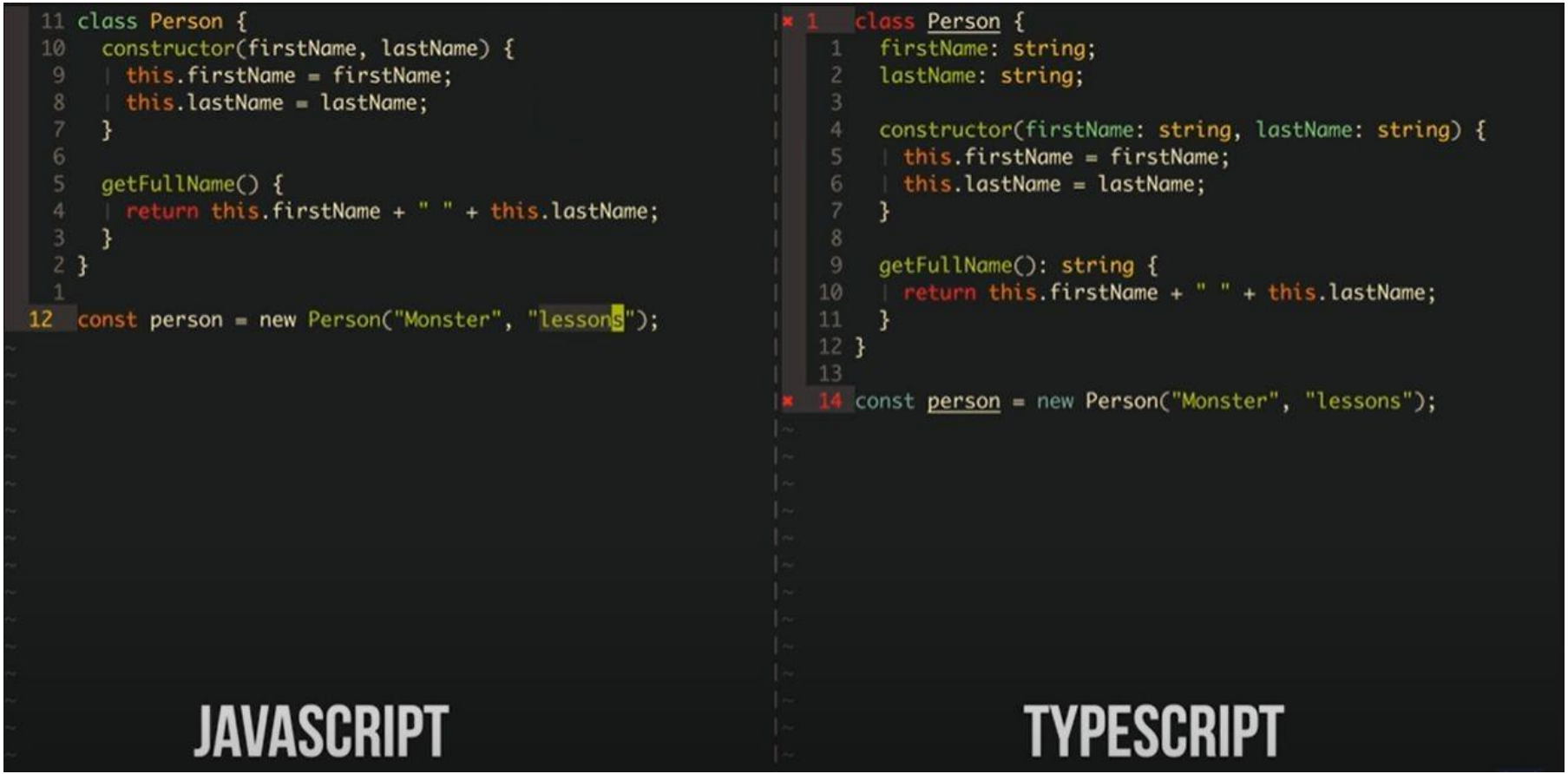
compile the app.ts file

```
tsc app.ts
```




```
node app.js
```

You will see the output as
Hello, World!


class Person {
constructor(firstName, lastName) {
this.firstName = firstName;
this.lastName = lastName;
}
getFullName() {
return this.firstName + " " + this.lastName;
}
}
const person = new Person("Monster", "lessons");
JAVASCRIPT
class Person {
firstName: string;
lastName: string;
constructor(firstName: string, lastName: string) {
this.firstName = firstName;
this.lastName = lastName;
}
getFullName(): string {
return this.firstName + " " + this.lastName;
}
}
const person = new Person("Monster", "lessons");
TYPESCRIPT

TypeScript project configuration :

- TypeScript uses a configuration file named tsconfig.json that holds a number of compilation options.
- We can use the tsc command-line compiler to generate a tsconfig.json file by using the --init command

tsc –init

tsc --target

TypeScript project configuration :

Option for `target` Property	Version of JavaScript
`"ES3"`	ECMAScript 3 (ES3)
`"ES5"`	ECMAScript 5 (ES5)
`"ES6"` or `"ES2015"`	ECMAScript 2015 (ES6)
`"ES2016"`	ECMAScript 2016 (ES7)
`"ES2017"`	ECMAScript 2017 (ES8)
`"ES2018"`	ECMAScript 2018 (ES9)
`"ES2019"`	ECMAScript 2019 (ES10)
`"ES2020"`	ECMAScript 2020 (ES11)
`"ESnext"`	Latest version

Watching files for changes :

- TypeScript also has a handy option that will watch an entire directory tree and if a file changes, it will automatically recompile the entire project.

```
tsc -w hello.ts
```


1 let hello: string = "world";
2 hello = [];
[tsserver 2322] [E] Type 'undefined[]' is not assignable to type 'string'.

Basic Type

TypeScript inherits the built-in types from JavaScript. TypeScript types is categorized into:

- **Primitive type**
- **Objective Type**

Primitive types

The following illustrates the primitive types in TypeScript:

Name	Description
`string`	represents text data
`number`	represents numeric values
`boolean`	has true and false values
`null`	has one value: null
`undefined`	has one value: `undefined`. It is a default value of an uninitialized variable
`symbol`	represents a unique constant value

Object types:

- Function
- Arrays
- Classes
- Objects
- Tuples
- Enum

Inferred Typing and Type Annotation

- TypeScript uses a technique called **inferred typing**, or **type inference**, to determine the type of a variable. This means that even if we do not explicitly specify the type of a variable, the compiler will determine its type based on when it was first assigned

- Inferred typing can be useful in TypeScript because it allows us to omit explicit type annotations in situations where the type can be easily determined from the context.

- NOTE : Inferred typing relies on TypeScript's type inference algorithm, which may not always make the correct type assignments.

- Example :

```
let items = [1, 2, 3, null];
```

```
let items = [0, 1, null, 'Hi'];
```

The any Type

Explicit Casting

```
var item1 = <any>{ id: 1, name: "item1" }

var item1 = { id: 1, name: "item1" } as any;
```

The let and const Keywords

```
var index: number = 0;
if (index == 0) {
  var index: number = 2;
  console.log(`index = ${index}`);
}
console.log(`index = ${index}`);
```

Union Types, Type Guards, and Aliases

TypeScript allows us to express a type as a combination of two or more other types. These types are known as union types, and they use the pipe symbol (|) to list all of the types that will make up this new type.

```
function printObject(obj: string | number) {
  // Log the value of obj
  console.log(`obj = ${obj}`);
}
printObject(1);
printObject("string value");
```

Type guards

- When working with union types, the compiler will still apply its strong typing rules to ensure type safety.

```
function add( a: string | number, b: string | number): number |string {
    return arg1 + arg2;
}
```

- A type guard is an expression that performs a check on our type and then guarantees that type within its scope.

```
function add(a: number | string, b: number | string): number | string {
    if (typeof a === 'number' && typeof b === 'number')
        return a + b;

    if (typeof a === 'string' && typeof b === 'string')
        return a + b;
}
```

Type aliases

- TypeScript introduces the concept of a type alias, where we can create a named type that can be used as a substitute for a type union.

- Type aliases can be used wherever normal types are used and are denoted by using the type keyword,

```
type StringOrNumber = string | number;
function addWithTypeAlias(
arg1: StringOrNumber,
arg2: StringOrNumber)
{
  return arg1.toString() + arg2.toString();
}
console.log( addWithTypeAlias(1, 2)); // Output: '12'
console.log( addWithTypeAlias('Hello', ' World')); // Output: 'Hello World'
```

Enums

```
enum DoorState {
  Open,
  Closed
}

function checkDoorState(state: DoorState) {
  console.log(`enum value is : ${state}`);

  switch (state) {
    case DoorState.Open:
      console.log(`Door is open`);
      break;
    case DoorState.Closed:
      console.log(`Door is closed`);
      break;
  }
}
```

Conditional Expressions

```
(conditional) ? ( true statement ) : ( false statement );
```

When using object properties in JavaScript, and in particular nested properties, it is important to ensure that a nested property exists before attempting to access it.

Optional chaining

```
var objectA = {
    nestedProperty: {
        name: "nestedPropertyName"
    }
}
function printNestedObject( obj) {
    console.log("obj.nestedProperty.name = " +
obj.nestedProperty.name);
}
printNestedObject(objectA);
//printNestedObject({});
```

```
function printNestedObject( obj: any) {
  // Check if the input object is defined
  if (obj != undefined
  && obj.nestedProperty != undefined
  && obj.nestedProperty.name) {
    console.log(`name = ${obj.nestedProperty.name}`)
  } else {
    console.log(`name not found or undefined`);
  }
}
printNestedObject(objectA);
```

```
function printNestedOptionalChain( obj: any) {
        if (obj?.nestedProperty?.name) {
                console.log(`name = ${obj.nestedProperty.name}`)
          } else {
                console.log(`name not found or undefined`);
          }
}
```

```
function printNestedOptionalChain(obj: any) {
        if (obj?.nestedProperty?.name) {
            console.log(`name = ${obj.nestedProperty.name}`)
        }
  } else {
     console.log(`name not found or undefined`);
  }
}
```

```
printNestedOptionalChain(undefined);
```

```
function printNestedOptionalChain(obj: any) {
       if (obj?.nestedProperty?.name) {
      console.log(`name = ${obj.nestedProperty.name}`)
     }
  } else {
    console.log(`name not found or undefined`);
  }
}
```

```
printNestedOptionalChain({
  aProperty: "another property",
});
```

```
function printNestedOptionalChain(obj: any) {
      console.log(`name = ${obj.nestedProperty.name}`)
  } else {
    console.log(`name not found or undefined`);
  }
}
```

```
printNestedOptionalChain({
  nestedProperty: {
    name: null,
  },
});
```

```
function printNestedOptionalChain(obj: any) {
        if (obj?.nestedProperty?.name) {
       console.log(`name = ${obj.nestedProperty.name}`)
    }
  } else {
     console.log(`name not found or undefined`);
  }
}
```

```
printNestedOptionalChain({
  nestedProperty: {
    name:null
  },
});
```

Nullish Coalescing operator (??) :

expr1 ?? `undefined or null`

This syntax provides an alternative value, which is provided on the right-hand side of the operator, to use if the variable on the left-hand side is either null or undefined.

```
function nullishCheck(a: number | undefined | null) {
  console.log(`a : ${a ?? `undefined or null`}`);
}
nullishCheck(1);

nullishCheck(null);

nullishCheck(undefined);
```

Null or undefined operands

```
function testNullOperands(a: number, b: number | null | undefined) {

      let addResult = a + (b ?? 0);
     console.log(addResult);
}

testNullOperands(5,undefined);
```

Object Types

- TypeScript introduces the object type to cover types that are not primitive types.
- This includes any type that is not **number, boolean, string, null, symbol, or undefined**.

```
let structuredObject: object = {
  name: "myObject",
  properties: {
    id: 1,
    type: "AnObject"
  }
};
function printObjectType(a: object) {
  console.log(`a: ${JSON.stringify(a)}`);
}
printObjectType(structuredObject);
printObjectType("this is a string");
```

The Unknown Types

- TypeScript introduces a special type into its list of basic types, which is the type unknown
- The unknown type can be seen as a type-safe alternative to the type any.
- A variable marked as unknown can hold any type of value, similar to a variable of type any.
- A variable of type unknown can’t be assigned to a known type without explicit casting.

```
let u: unknown = "an unknown";
u = 1;
let aNumber2: number;
aNumber2 = u;
//aNumber2 = <number>u;
```

The never Type

This type is used to indicate instances where something should never occur.

```
function alwaysThrows() : never {
  // Throw an error with a specified message
  throw new Error("this will always throw");

  // Return a value, but it will never be reached because of the thrown error
  return -1;
}

alwaysThrows();
```

Object Spread

```
var firstObj = { id: 1, name: "firstObj" };

var secondObj = { ...firstObj };

console.log(`secondObj : ${JSON.stringify(secondObj)}`);

let emailObj: object = { email : " abc@example.com " };

let obj3 = { ... firstObj, ...emailObj };

console.log(`obj3 = ${JSON.stringify(obj3)}`);
```

Spread precedence

When using object spread, properties will be copied incrementally.

In other words, if two objects have a property with the same name, then the object that was specified last will take precedence.

Spread with arrays

```
let firstArray = [1, 2, 3];

let secondary = [3, 4, 5];

let thirdArray = [...firstArray, ...secondArray];

console.log(`third array = ${thirdArray}`);
```

Tuples

- Tuples are a method of defining a type that has a finite number of unnamed properties, with each property having an associated type.

- When using a tuple, all of the properties must be provided.

```
let tuple1: [string, boolean];
tuple1 = ["test", true];
tuple1 = ["test"];
```

Tuples destructuring

- As tuples use the array syntax, they can be destructured or disassembled in two ways.

```
let tuple1: [string, boolean];
console.log(`tuple1[0] : ${tuple1[0]}`);
console.log(`tuple1[1] : ${tuple1[1]}`);

let [tupleString, tupleBoolean] = tuple1;
console.log(`tupleString = ${tupleString}`);
console.log(`tupleBoolean = ${tupleBoolean}`);
```

Optional tuple elements

```
let tupleOptional: [string, boolean?];

tupleOptional = ["test", true];

tupleOptional = ["test"];

console.log(`tupleOptional[0] : ${tupleOptional[0]}`);

console.log(`tupleOptional[1] : ${tupleOptional[1]}`);
```

Tuples and spread syntax

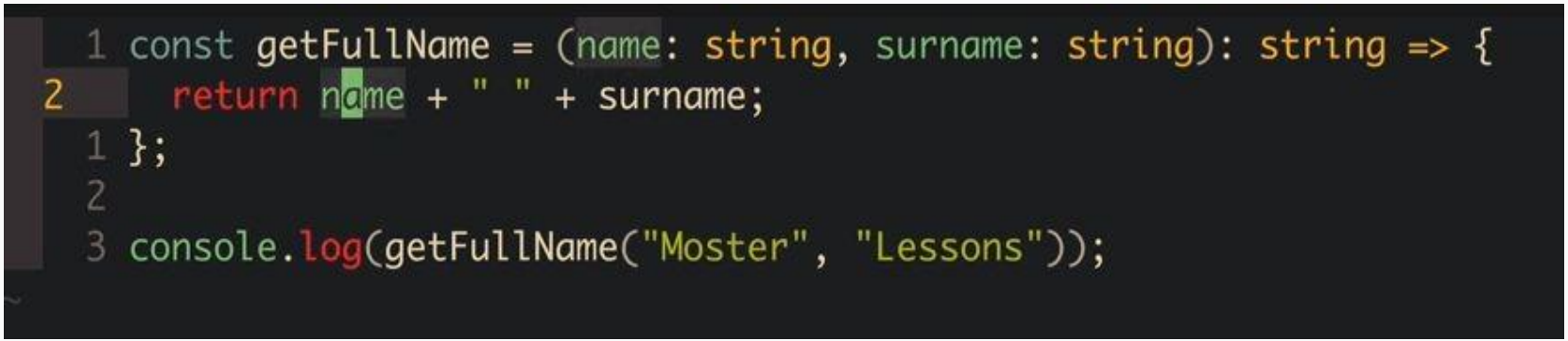
```
let tupleRest: [number, ...string[]];

tupleRest = [1];

tupleRest = [1, "string1"];

tupleRest = [1, "string1", "string2"];
```

Object destructuring

```
let complexObject = {
 aNum: 1,
 bStr: "name",
 cBool: true
}
let { aNum, bStr, cBool } = complexObject;
console.log(`aNum : ${aNum}`);
console.log(`bStr : ${bStr}`);
console.log(`cBool : ${cBool}`);
let { aNum: objId, bStr: objName, cBool: isValid } = complexObject;
```

Function

TypeScript functions are the building blocks of readable, maintainable, and reusable code.



Function Signatures and the void Keyword

```
// This function calculates the result of (a * b) + c
function calculate(a, b, c) {
  // Return the result of (a * b) + c
  return (a * b) + c;
  }
// Log the result of calling the calculate function with arguments 3, 2 and 1
console.log("calculate() = " + calculate(3, 2, 1));
console.log("calculate() = " + calculate("2", "3", "1"));
```

Function with Optional parameters

```
function concatValues(a: string, b?: string) {
  console.log(`a + b = ${a + b}`);
}

// Call the function with two arguments
concatValues("first", "second");

// Call the function with only one argument
concatValues("third");
```

Note: Any optional parameters must be listed last in the parameter list of the function definition. We can have as many optional parameters as we like as long as nonoptional parameters precede the optional parameters.

Function with Default parameters

```
function concatWithDefault(a: string, b: string = "default") {

  console.log(`a + b = ${a + b}`);

}

concatWithDefault("first", "second");

concatWithDefault("third");
```

Note: Any optional parameters must be listed last in the parameter list of the function definition. We can have as many optional parameters as we like as long as nonoptional parameters precede the optional parameters.

Function with Rest parameters

If we do not specify any parameters in a function definition, we can still access the values that were provided when the function was invoked.

```
function testArguments() {

  for (var i = 0; i < arguments.length; i++) {

    console.log(`argument[${i}] = ${arguments[i]}`);

  }

}

testArguments(1, 2);

testArguments("first", "second", "third");
```

Function with Rest parameters

```
function testArguments(...args: (string[] | number[])) {
  for (let i in args) {
    // Log each argument in the format `args[i] = argument_value`
    console.log(`args[${i}] = ${args[i]}`);
  }
}
// Call the `testArguments` function with different arguements
testArguments("1");
testArguments(10, 20);
```

Function overrides

TypeScript provides an alternative to union types when defining a function and allows a function signature to provide different parameter types.

```
function add(a: string, b: string): string;
function add(a: number, b: number): number;

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

add("first", "second");
add(1, 2);
add(true, false);
```

Interfaces

Interfaces provide us with a mechanism to define what properties an object must implement and is, therefore, a way for us to define a custom type.

By defining an interface, we are describing the properties and functions that an object is expected to have in order to be used by our code.

```
interface IIdName {
  id: number;
  name: string;
}
let idObject: IIdName = {
  id: 2,
  name: "this is a name",
};
```

Interfaces do not generate any JavaScript code. This means that interfaces are constructs only used in the TypeScript compilation step and language services and are there to ensure type safety.

```ts
interface Person {
    firstName: string;
    lastName: string;
}
```

```ts
function getFullName(person: Person) {
    return `${person.firstName} ${person.lastName}`;
}

let john = {
    firstName: 'John',
    lastName: 'Doe'
};

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

Optional Properties and Prefixing Interface Names

Interface definitions may also include optional properties similar to functions that specify optional parameters using the question mark (?) syntax.

```
interface IOptional {
  id: number;
  name?: string;
}
let optionalId: IOptional = {
  id: 1,
};

let optionalIdName: IOptional = {
  id: 2,
  name: "optional name",
};
```

Weak Types in Interfaces

When we define an interface where all of its properties are optional, this is considered to be a weak type. In other words, we have defined an interface, but none of the properties of the interface are mandatory.

```
interface IWeakType {
  id?: number;
  name?: string;
}

let weakTypeNoOverlap: IWeakType = {
  description: "a description",
};
```

TypeScript is strongly type even weak types.

The in Operators

```
interface IIdName {
  id: number;
  name: string;
}
interface IDescrValue {
  descr: string;
  value: number;
}
function printNameOrValue(obj: IIdName | IDescrValue): void {
        if ('id' in obj) {
           console.log(`obj.name : ${obj.name}`);
          }
        if ('descr' in obj) {
           console.log(`obj.value : ${obj.value}`);
          }
}
```

The Usage in Operators

```
interface IIdName {
  id: number;
  name: string;
}
interface IDescrValue {
  descr: string;
  value: number;
}

function printNameOrValue(obj: IIdName | IDescrValue): void {
  if ("id" in obj) {
    console.log(`obj.name : ${obj.name}`);
  }
  if ("descr" in obj) {
    console.log(`obj.value : ${obj.value}`);
  }
}
```

```
printNameOrValue({
  id: 1,
  name: "nameValue",
});
printNameOrValue({
  descr: "description",
  value: 2,
});
```

The keyof Operators

TypeScript allows us to iterate through the properties of a type and extract the names of its properties through the keyof keyword, which we can use as a string literal type.

```
interface IPerson {
  id: number;
  name: string;
}
type PersonPropertyName = keyof IPerson;
function getProperty(key: PersonPropertyName, value: IPerson) {
  console.log(`${key} = ${value[key]}`);
}
getProperty("id", { id: 1, name: "firstName" });
getProperty("name", { id: 2, name: "secondName" });
getProperty("telephone", { id: 3, name: "thirdName" });
```

Classes in Typescripts

ES5

```
function Person(ssn, firstName, lastName) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
}
```

```
Person.prototype.getFullName = function () {
    return `${this.firstName} ${this.lastName}`;
}
```

```
let person = new Person('171-28-0926','John','Doe');
console.log(person.getFullName());
```

ES6

```
class Person {
    ssn;
    firstName;
    lastName;

    constructor(ssn, firstName, lastName) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName() {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

```
let person = new Person('171-28-0926','John','Doe');
console.log(person.getFullName());
```

TypeScript

```typescript
class Person {
    ssn: string;
    firstName: string;
    lastName: string;

    constructor(ssn: string, firstName: string, lastName: string) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

```typescript
let person = new Person(171280926, 'John', 'Doe');
```

Class Constructors and Modifiers

Constructor Parameter Properties and Property Accessors

TypeScript also introduces a shorthand version for access modifiers that can be applied to parameters in a constructor function.

```
class ClassWithCtorMods {
        constructor(public id: number, private name: string)
        {

         }
}
let myClassMod = new ClassWithCtorMods(1, "test");
console.log(`myClassMod.id = ${myClassMod.id}`);
console.log(`myClassMod.name = ${myClassMod.name}`);
```

The readonly property in TypeScript

This is similar to the concept of the const keyword and means that once a value has been assigned to a readonly property, it is not allowed to be modified.

```
class ClassWithReadonly {
readonly name: string;
   constructor(_name: string) {
                this.name = _name;
   }
  setNameValue(_name: string) {
                this.name = _name;
   }
}
```

property accessors

```
class ClassWithAccessors {
        private _id: number = 0;
        get id(): number {
          console.log(`get id property`);
          return this._id;
        }
        set id(value: number) {
          console.log(`set id property`);
          this._id = value;
        }
}
let classWithAccessors = new ClassWithAccessors();
classWithAccessors.id = 10;
console.log(`classWithAccessors.id = ${classWithAccessors.id}`);
```

Static Functions and Namespaces

A class can mark a function with the static keyword, meaning that there will only be a single instance of this function available throughout the code base.

When using a static function, we do not need to create an instance of the class in order to invoke this function.

```
class StaticFunction {
        static printTwo() {
                console.log(`2`);
          }
}
StaticFunction.printTwo();
```

Static properties

- In a similar manner to static functions, classes can also have static properties.

- If a class property has been marked as static, then there will only be a single instance of this property throughout the code base.

```
class StaticProperty {
    static count = 0;
    updateCount() {
        StaticProperty.count++;
    }
}
let firstInstance = new StaticProperty();
let secondInstance = new StaticProperty();
firstInstance.updateCount();
console.log(`StaticProperty.count = ${StaticProperty.count}`);
secondInstance.updateCount();
console.log(`StaticProperty.count = ${StaticProperty.count}`);
```

Namespaces in TypeScript

- When working within large projects, and particularly when working with large numbers of external libraries, there may come a time when two classes or interfaces share the same name. To handle such situations, TypeScript employs namespaces.
- A namespace in TypeScript is a way to organize code and ensure that class names remain unique.
- To create a namespace in TypeScript, we use the namespace keyword followed by the desired name of the namespace.
- Within the namespace block, we can define one or more classes or interfaces.

```
namespace FirstNameSpace {
   export class NameSpaceClass {
   }
   class NotExported {
   }
}
```

Exporting a class from a namespace

```
let notExported = new FirstNameSpace.NotExported();
```

```
let nameSpaceClass = new FirstNameSpace.NameSpaceClass();
```

Inheritance in TypeScript

- In TypeScript, inheritance can be used by classes and interfaces
- TypeScript uses the extends keyword to implement inheritance

Interface inheritance

```
interface IBase {
  id: number;
}

interface IDerivedFromBase  extends IBase {
   name: string;
}

class IdNameClass implements IDerivedFromBase {
     id: number = 0;
     name: string = "nameString";
}
```

Multiple inheritance in interfaces

```
interface IBase {
  id: number;
}
interface IDerivedFromBase  extends IBase {
   name: string;
}
interface IMultiple extends IDerivedFromBase, IBase {
     description: string;
}

let multipleObject: IMultiple = {
        id: 1,
        name: "myName",
        description: "myDescription",
};
```

Class inheritance

```
class BaseClass implements IBase {
  id: number = 0;
}

class DerivedFromBaseClass
extends BaseClass
implements IDerivedFromBase {
      name: string = "nameString";
}
```

A class with multiple interfaces

```
interface IFirstInterface {
    id: number;
}
interface ISecondInterface {
    name: string;
}
class MultipleInterfaces implements IFirstInterface, ISecondInterface {
        id: number = 0;
        name: string = "nameString";
}
```

The super Function

If a derived class has a constructor, then this constructor must call the base class constructor using the super keyword, or TypeScript will generate an error.

```
class BaseClassWithCtor {
    private id: number;
    constructor(id: number) {
        this.id = id;
    }
}
class DerivedClassWithCtor extends BaseClassWithCtor {
    private name: string;
    constructor(id: number, name: string) {
      super(id);
      this.name = name;
    }
}
```

Function Overriding

- Function overriding is when a derived class defines a method that has the same name as a base class method.
- The derived class can determine whether or not to call the implementation of the function in the base class.

```
class BaseClassWithFn {
  print(text: string) {
      console.log(`BaseClassWithFn.print() : ${text}`)
  }
}
class DerivedClassFnOverride extends BaseClassWithFn {
    print(text: string) {
        console.log(`DerivedClassFnOverride.print(${text})`);
    }
}

let derivedClassFnOverride = new DerivedClassFnOverride();
derivedClassFnOverride.print("test");
```

Protected Member

- Classes can mark both properties and functions with the protected keyword.
- If a property is marked as protected, then it is not accessible outside of the class itself, similar to the behavior of the private keyword.
- It is, however, accessible to derived classes, which is different from private variables that are not accessible to derived classes

```
class BaseClassProtected {
        protected id: number;
        private name: string = "";
        constructor(id: number) {
                this.id = id;
          }
}
class AccessProtected extends BaseClassProtected {
     constructor(id: number) {
        super(id);
        console.log(`base.id = ${this.id}`);
        console.log(`base.name = ${this.name}`);
   }
}
```

Abstract Classes

- An abstract class is a class that can’t be instantiated. In other words, it is a class that is designed to be derived from.
- The purpose of abstract classes is generally to provide a set of basic properties or functions that are shared across a group of similar classes.
- Abstract classes are marked with the abstract keyword.

```
abstract class EmployeeBase {
  public id: number;
  public name: string;
  constructor(id: number, name: string) {
    this.id = id;
    this.name = name;
  }
}
class OfficeWorker extends EmployeeBase {}
class OfficeManager extends OfficeWorker {
      public employees: OfficeWorker[] = [];
}
let joeBlogg = new OfficeWorker(1, "Joe");
let jillBlogg = new OfficeWorker(2, "Jill");
let jackManager = new OfficeManager(3, "Jack");
```

Abstract Class Methods

- An abstract class method is similar to an abstract class in that it is designed to be overridden.
- In other words, declaring a class method as abstract means that a derived class must provide an implementation of this method. For this reason, abstract class methods are not allowed to provide a function implementation.

```
abstract class EmployeeBase {
  public id: number;
  public name: string;
  abstract doWork(): void;
  constructor(id: number, name: string) {
    this.id = id;
    this.name = name;
  }
}
```

```
abstract class EmployeeBase {
  public id: number;
  public name: string;
  abstract doWork(): void;
  constructor(id: number, name: string) {
    this.id = id;
    this.name = name;
  }
}
class OfficeWorker extends EmployeeBase {
  doWork() {
    console.log(`${this.name} : doing work`);
  }
}
```

The instanceof Operator

```
class A {}
class BfromA extends A {}
class CfromA extends A {}
class DfromC extends CfromA {}
console.log(`A instance of A :
 ${new A() instanceof A}`);
console.log(`BfromA instance of A :
 ${new BfromA() instanceof A}`);
console.log(`BfromA instance of BfromA :
 ${new BfromA() instanceof BfromA}`);
```

Any Questions?

References

- prototype inheritance
- ES6 allowed you to define a class
- https://www.typescripttutorial.net/
- https://www.typescriptlang.org/
- TypeScript type annotations