## Inheritance Or IS-A Relationship:

- Inheritance is a concept of extending the parent class, by creating the child class.
- In this process, all the members (Variables and Methods) of parent class will be inherited (derived) into the child class.
- The main advantage of inheritance is **Code Reusability** and we can extend existing functionality with some more extra functionality.
- The "**extends**" keyword is used to create inheritance.

  For example,
  "Student" class extends "Person class".
  "Car" class extends "Vehicle" class.
  "SmartMobile" class extends "Mobile" class

**Note:**

1. When parent class has a constructor, the child class's constructor must call the parent class's constructor explicitly. Otherwise, it will be an error.
2. The "**super()**" method is used to call the Parent class's **constructor**

**Inheritance - Example**

```
export {};

class Mobile {
  constructor(model: number) {
    console.log("Mobile Class Constructor");
  }
  call(): void {
    console.log("call...");
  }
  msg(): void {
    console.log("msg...");
  }
}

class SmartMobile extends Mobile {
  constructor(model: number) {
    super(model);
    console.log("SmartMobile Class Constructor");
  }
  browsing(): void {
    console.log("browsing...");
```

```
  }
}

let m: Mobile = new Mobile(1234);
m.call();
m.msg();

let sm: SmartMobile = new SmartMobile(1234);
sm.call();
sm.msg();
sm.browsing();
```

## Overriding:

- Whatever the parent class has by default available to the child class.
- If the child is not satisfied with parent class implementation, then child is allowed to redefine its implementation in its own way.
- This process is called **Overriding**.

```
export{}
class Mobile{
    call():void{
        console.log("Audio Calling");
    }
}
class SmartMobile extends Mobile{
    call():void{
        console.log("Audio+Video Calling");
    }
}

let m:Mobile = new Mobile();
m.call();

let sm:SmartMobile = new SmartMobile();
sm.call();
```

```
Audio Calling
Audio+Video Calling
```

## Abstract Method:

- Sometimes we don't know the about implementation, still we can declare a method.
- Such type of methods is called abstract methods. i.e., abstract method has only declaration but not implementation.
- Abstract methods can declare using '**abstract'** keyword.
  **Syntax:**
  **abstract methodName(): return_type;**
- Abstract methods must end with semi colon (;).
- It must be taken only in **Abstract class**.

## Abstract Class:

- Sometimes implementation of a class is not complete, such type of partially implementation class is called **Abstract Class**.
- Abstract class can be created using "**abstract**" keyword.
- Abstract class can't be instantiated.

**Syntax:**

```
abstract class ClassName{
    abstract Method;
    concrete Method{ }
}
```

**Eg:**

```
export{}
abstract class CarRacingGame{
    start():void{
          console.log("Car Started");
    }
    abstract move():void;
    abstract stop():void;
}
class MyCarRacingGame extends CarRacingGame{
    move():void{
        console.log("Car moved");
    }
    stop():void{
        console.log("Car stopped");
    }
}
let game = new MyCarRacingGame();
game.start();
game.move();
game.stop();
```

## Interfaces

- Interface is considered as a **Service Requirement Specification** (SRS) (or) a **contract between a client and service provider** (or) **100% pure abstract class**.

  Interfaces act as a **mediator** between two or more developers;
  one developer implements the interface, other developer creates reference variable for the interface and invokes methods; so interface is common among them.

- **'interface'** key is used to create an interface.

- Interfaces doesn't contain actual code; contains only **list of properties** and **method declaration.**
  Eg:

```
interface Printer{
    modelNo:number;

    scan():void;
    print():void;
}
```

- Interfaces doesn't contain method implementation (method definition);

- The child class that implements the interface must implement all the methods of the interface; if not, compile-time error will be shown at child class.

- The child class can implement the interface with "**implements**" keyword.
  Eg:

```
class EpsonPrinter implements Printer {
    modelNo: number = 10101;
    scan(): void {
        console.log("scanning...");
    }
    print(): void {
        console.log("printing...");
    }
}
```

**Example**

```
export{}
interface Printer{
    modelNo:number;

    scan():void;
    print():void;
}
class EpsonPrinter implements Printer{
    modelNo: number = 10101;
    scan(): void {
        console.log("scanning...");
    }
    print(): void {
        console.log("printing...");
    }
}

let s1:Printer = new EpsonPrinter();
s1.scan();
s1.print();
console.log(s1.modelNo);

let s2:EpsonPrinter = new EpsonPrinter();
s2.scan();
s2.print();
console.log(s2.modelNo);
```

```
scanning...
printing...
10101
scanning...
printing...
10101
```

Eg:2

```
export {};

interface UPIPaymentSystem {
  upiId: string;

  payment(): void;
}

class GooglePay implements UPIPaymentSystem {
  upiId: string = "abc@123";
  payment(): void {
    console.log("done payment using GooglePay");
  }
}
class PhonePay implements UPIPaymentSystem {
  upiId: string = "abc@123";
  payment(): void {
    console.log("done payment using PhonePay");
  }
}

class PayTM implements UPIPaymentSystem {
  upiId: string = "abc@123";
  payment(): void {
    console.log("done payment using PayTM");
  }
}

let upi = new GooglePay();
upi.payment();

let phonePay = new PhonePay();
phonePay.payment();

let payTM = new PayTM();
payTM.payment();
```

## Working with Access Modifiers:

- Access Modifiers specify where the member of a class can be accessible.
- That means it specifies whether the member of a class is accessible outside the class or not.
- These are used to implement "security" in OOP.
- For each member (property / method), we can specify the access modifier separately.
- "**public**" is the access modifier for all the members (property / method) in Typescript class.
- Typescript supports three access modifiers:

1. **public (default):**
   The public members are accessible **anywhere** in the program (in the same class and outside the class also).

2. **private:**
   The private members are accessible within the same class only; If you try to access them outside the class, you will get **compile-time error.**

3. **protected:**
   The protected members are accessible within the same class and also in the corresponding child classes; If you try to access them outside the same class or child class, you will get compile-time error.

**Access Modifiers - Example**

```
export{}
class Student {
    public studentId: number = 101;
    private studentName:string = "Sai";
    protected studentMarks: number = 95;

    public displayStudentDetails():void{
        console.log("Student Details");
        console.log(this.studentId);
        console.log(this.studentName);
        console.log(this.studentMarks);
    }
}
class EngineeringStudent extends Student{
    public displayEngineeringStudentDetails():void{
        console.log("EngineerStudent Details");
        console.log(this.studentId);
        //console.log(this.studentName);//not accessible
        console.log(this.studentMarks);
    }
}

var s = new Student();
s.displayStudentDetails();

var s2 = new EngineeringStudent();
s2.displayEngineeringStudentDetails();

console.log(s.studentId);
//console.log(s.studentName);//not accessible
//console.log(s.studentMarks);//not accessible
```

```
Student Details
101
Sai
95
EngineerStudent Details
101
95
101
```

# Working with Arrays in Typescript

- Typescript supports arrays concept to represent a group of homogeneous elements.
  Eg:

**Creating Array:**

Syntax 1:

let arrayVariable: DataType[] = [ element1, element2, element3, ….. ];

Syntax 2:

let arrayVariable: Array<DataType> = [ element1, element2, element3, ….. ];

Eg:

```
//traditional Syntax
let myArray:number[] = [90,20,30,70,50];
console.log(myArray);

//Generic Syntax
let myArray2: Array<number> = [90,20,30,70,50];
console.log(myArray2)
```

**Array Traversal:**

1. **Using for of loop:**

```
//traditional Syntax
let myArray:number[] = [90,20,30,70,50];

console.log("Tradition For loop");
for(let x=0; x < myArray.length; x++){
    console.log(myArray[x])
}

console.log("Using For in loop")
for(let y in myArray){
    console.log(myArray[y])
}

console.log("Using For of loop");
for(let a of myArray){
    console.log(a)
}
```

## Important Functions of Arrays:

**forEach():** This method executes a provided function once for each array element.
Eg

```
let myArray:number[] = [90,20,30,70,50];
myArray.forEach(value => console.log(value))
```

**map():** This method **creates a new array** populated with the results of calling a provided function on every element in the calling array.

**Eg: Multiply each array value by 2**

```
let myArray:number[] = [90,20,30,70,50];
var result = myArray.map((value)=> value * 2);
console.log(result);
```

```
Output:
D:\TypeScriptDemo>tsc test.ts
D:\TypeScriptDemo>node test
[ 180, 40, 60, 140, 100 ]
```

**filter():** This method creates a new array with all elements that pass the test implemented by the provided function.

**Eg: generate an array with elements which are greater than 50**

```
let myArray:number[] = [90,20,30,70,50];
var result = myArray.filter(value => value > 50);
console.log(result);
```

```
Output:
D:\TypeScriptDemo>tsc test.ts
D:\TypeScriptDemo>node test
[ 90, 70 ]
```

**reduce():** This method executes a reducer function (that you provide) on each element of the array, resulting in a **single output** value.

**Eg: Find the sum of elements of the given array.**

```
let myArray:number[] = [90,20,30,70,50];
var sumOfArrayElement = myArray.reduce((total, value) => total + value);
console.log(sumOfArrayElement);
```

```
D:\TypeScriptDemo>tsc test.ts
D:\TypeScriptDemo>node test
260
```

# Working with Modules

**Without Modules**

test.ts

```
class Product{
    productId:number
    productName:string
    productPrice:number
    productBrand:string
  constructor(
    productId: number,
    productName: string,
    productPrice: number,
    productBrand: string
) {
    this.productId = productId
    this.productName = productName
    this.productPrice = productPrice
    this.productBrand = productBrand
  }
}
class ProductService{
    products: Product[];

    constructor(){
        this.products = [
            new Product(101,"iPhone 10", 5000, 'Apple'),
            new Product(102, 'iPhone 12', 6000, 'Apple'),
            new Product(103,'Samsung Note 10', 4000, 'Samsung'),
            new Product(104,'Samsung Note 11', 4000, 'Samsung'),
            new Product(105,'Pixel 5', 4000, 'Google')
        ]
    }

    getProducts():Product[]{
        return this.products;
    }
}
let productService = new ProductService();
let products = productService.getProducts();
console.log(products)
```

- It is not recommended or not possible to write entire code (variables, classes, interface, etc) in a single file by a single developer.
  Example:

- In large scale applications, it is recommended to write each class in a separate file.

- To **separate code concerns** and **responsibilities** we need to go for **Modules** concept in Typescript.

- Module is a simple typescript file(.ts file) which contain at top-level "**import"** or "**export**" keywords

- We can export the code (variables, class, interface, etc) from one module to other modules by using "**export**" keyword.

- We can use the exported the code (variables, class, interface, etc) from one module to other modules by using "**import**" keyword.

**Note**:

1. We can't import the code (variables, class, interface, etc) that are not exported from the module.
2. module names are case insensitive.

**Loading Modules:**

1. Current folder, we use "**./module1**"
2. Sub folder in Current folder, we use **"./sub_folder_name/module1**"
3. Parent folder, we use "**../module1**"

**Steps for development of modules:**

**Step 1: Export a class of module1.ts**

**Syntax 1:**
**module1.ts**

```
export class Class1{}
export class Class2{}
export class Class3{}
```

**Syntax 2:**
**module1.ts**

```
class Class1{}
class Class2{}
class Class3{}

export {Class1, Class2, Class3}
```

**Step 2: Import the exported code in module.ts**

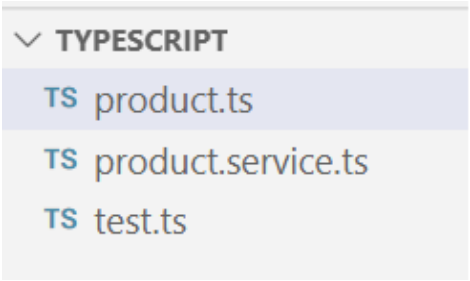
Importing single class.

```
import { Class1 } from "./module1";
```

Importing multiple classes.

```
import { Class1,Class2,Class3 } from "./module1";
```

**Example: Using Modules**




**product.ts**

```
export class Product{
    productId:number
    productName:string
    productPrice:number
    productBrand:string


  constructor(
    productId: number,
    productName: string,
    productPrice: number,
    productBrand: string
) {
    this.productId = productId
    this.productName = productName
    this.productPrice = productPrice
    this.productBrand = productBrand
  }
}
```

**product.service.ts**

```
import { Product } from './product'

export class ProductService{
    products: Product[];

    constructor(){
        this.products = [
            new Product(101,"iPhone 10", 5000, 'Apple'),
            new Product(102, 'iPhone 12', 6000, 'Apple'),
            new Product(103,'Samsung Note 10', 4000, 'Samsung'),
            new Product(104,'Samsung Note 11', 4500, 'Samsung'),
            new Product(105,'Pixel 5', 7000, 'Google')
        ]
    }

    getProducts():Product[]{
        return this.products;
    }

    //More Methods
    getProductsByBrand(brand:string):Product[]{
        return this.products.filter((p:Product) => p.productBrand === brand)
    }

    getTotalProductsPrice():number{
        return this.products.reduce( (total,value) => total + value.productPrice,0)
    }
}
```

**test.ts**

```
import { ProductService } from './product.service'

let productService = new ProductService();
let products = productService.getProducts();
console.log(products)

let appleBrandProducts = productService.getProductsByBrand('Apple')
//console.log(appleBrandProducts)

let totalProductsPrice = productService.getTotalProductsPrice()
//console.log(totalProductsPrice)
```