
TS

# Object Oriented Programming

# Introduction

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain
  - **data**, in the form of fields (often known as attributes or properties)
  - **code**, in the form of procedures (often known as methods)
- In OOP, programs are designed by making them out of objects that interact with one another
- OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types
- Class-based programming, or more commonly class-orientation, is a style of Object-oriented programming (OOP) in which inheritance occurs via defining classes of objects, instead of inheritance occurring via the objects alone

# Object

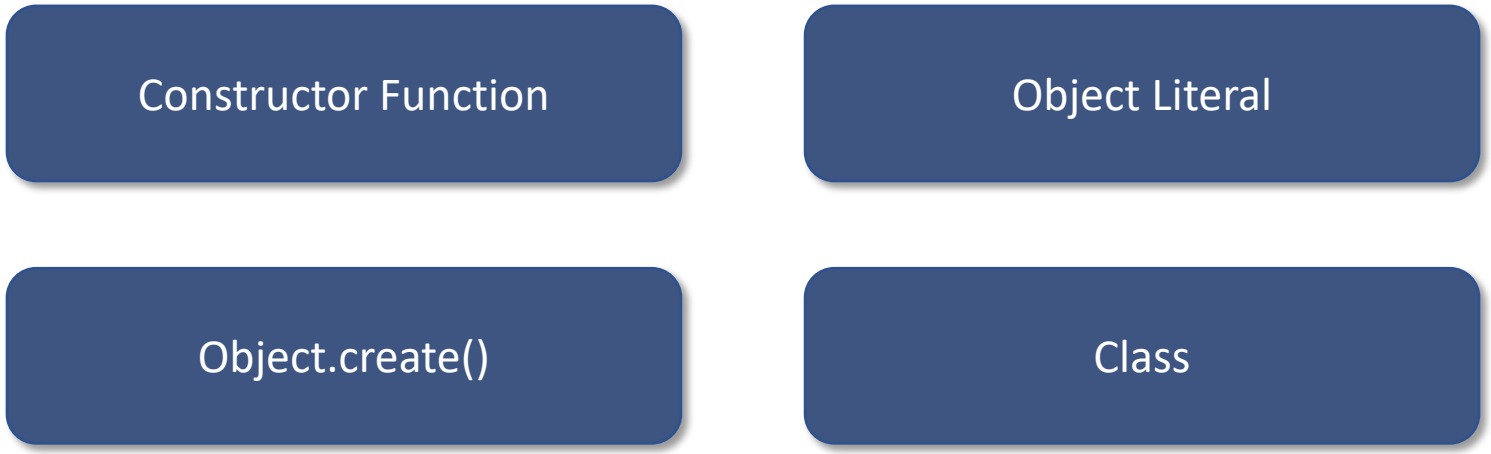
- At the heart of OOP is the concept of an object, which may refer to an abstract or concrete object in our world so this makes it easier for programmers to model actual problems with computers languages before trying to find the solutions
- An object can have data (which form the properties or attributes of the real-world object) and code in a form of methods (which represent the behavior in the equivalent real-world object)
- Think of a car for example, it has a color, weight and speed and can move forward or backward

# Object Roles

- Store state
- Pass state from one entity to another
- Group related functionality together
- Model real world objects

# Object Creation

- In TypeScript object can be created by using one of the following techniques



# Constructor Function

- TypeScript allows object creation using constructor function

```
function Car(model: string, price: number) {
  this.model = model
  this.price = price
}

const car1 = new Car('i20', 7.5)
console.log(car1)
```

# Object Literal

- The object literal may very well be one of the best and most popular features in TypeScript
- It's a superset of JSON (which has now become the de facto standard for data transport on the web, quickly replacing XML)
- The great thing about object literals is that they make it crazy easy to assemble an arbitrarily nested and dynamic set of data object with a definition syntax that is human-readable

```
const person = {
  name: 'person1',
  age: 20
}
```

```
const cat = 'Miaow'
const dog = 'Woof'
const bird = 'Peet peet'

const someObject = {
  cat,
  dog,
  bird
}
```

# Object.create()

- Object.create() creates a new object using existing object as prototype of the newly created one
- By default it copies all the properties of existing object
- New properties can be added to the newly created object without affecting the older object

```
const person = Object.create({})
person.name = 'person1'
person.age = 30
person.address = 'pune'
```

# Class

- Classes are the fundamental entities used to create reusable components
- In ECMAScript 6, object-oriented class based approach was introduced
- TypeScript introduced classes to avail the benefit of object-oriented techniques like encapsulation and abstraction
- The class in TypeScript is compiled to plain JavaScript functions by the TypeScript compiler to work across platforms and browsers
- A class can include
  - Constructor
  - Properties
  - Methods

```
class Person {
  name: string
  age: number
  address: string

  printInfo() {
    console.log(`name: ${this.name}`)
    console.log(`address: ${this.address}`)
    console.log(`age: ${this.age}`)
  }
}

const person = new Person()
person.name = 'person1'
person.age = 20
person.address = 'pune'
```

# Properties

# Properties

- Properties are the data members used to store the object state
- Properties can be added to the object
  - At the time of declaring a class
  - Dynamically

```
class Person {
  name: string
  age: number
}

const person = new Person()
person.name = 'person1'
person.age = 20
person['address'] = 'pune'
```

# Modifiers

- In object-oriented programming, the concept of 'Encapsulation' is used to make class members public or private i.e. a class can control the visibility of its data members
- There are three types of access modifiers in TypeScript
  - public
  - private
  - protected
- TypeScript has other modifiers
  - readonly

# Public Properties

- By default properties of class are public
- These properties can be freely accessed outside of the class

```
class Person {
  public name: string
  public age: number
}

const person = new Person()
person.name = 'person1'
person.age = 20
```

# Private Properties

- The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class

```
class Person {
  private name: string
  public age: number
}

const person = new Person()
// can not access name as its private
// person.name = 'person1'

// can access age as its public
person.age = 20
```

# Protected Properties

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

```
class Person {
  protected name: string
}

class Employee extends Person {
  private id: number

  printInfo() {
    console.log(`id: ${this.id}`)

    // can access name here as its protected
    console.log(`name: ${this.name}`)
  }
}

const emp = new Employee()

// can not access name as its protected
// emp.name = 'emp1'
```

# Readonly Properties

- TypeScript includes the readonly keyword that makes a property as read-only in the class
- Prefix readonly is used to make a property as read-only
- Read-only members can be accessed outside the class, but their value cannot be changed
- Since read-only members cannot be changed outside the class, they either need to be initialized at declaration or initialized inside the class constructor

```
class Employee {
  name: string
  readonly id: number
}
```

# Static Properties

- These properties are visible on the class, rather than on the object
- Every object of the class shares the static properties

```
class Maths {
  static readonly pi = 10
}

console.log(`pi = ${Maths.pi}`)
```

# Methods

# Methods

- A class provides methods to add perform operations on properties
- A class can have following types of methods
  - Constructor
  - Setters
  - Getters
  - Faciliters

# Constructor

- A constructor method allows object initialization
- In TypeScript, “constructor” is the name of the constructor method

```
class Person {
  name: string
  address: string

  constructor(name: string, address: string) {
    this.name = name
    this.address = address
  }
}
```

# Setters and Getters

- A class provides
  - setter (also known as mutator) to set/change the property value
  - getter (also known as inspector) to get the current value of a property

```
class Person {
  private name: string

  constructor(name: string) {
    this.name = name
  }

  setName(name: string) { this.name = name }
  getName() { return this.name }
}
```

# Accessors

- TypeScript supports getters/setters as a way of intercepting accesses to a member of an object
- This gives you a way of having finer-grained control over how a member is accessed on each object
- Its provides easy property access using dot (.) syntax

```
class Person {
  private _name: string

  public get name() { return this.name }

  public set name(name: string) {
    this._name = name
  }
}

const person = new Person()
person.name = 'person1'
console.log(`name: ${person.name}`)
```

# Facilitators

- These methods provide facilities in the class

```
class Person {
  private age: number
  private name: string

  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }

  canVote(): boolean {
    return this.age >= 18
  }
}
```

# Static Methods

- Static methods are visible on the class rather than object of a class
- Static methods can not access non-static methods or non-static properties

```
class Maths {
  static readonly pi = 10

  static add(p1: number, p2: number) {
    console.log(`addition = ${p1 + p2}`)
  }
}

Maths.add(10, 20)
```

# Advanced OOP

# Inheritance

- Represents is-a relationship
- Used to reuse the code from one class to another
- TypeScript uses extends keyword for inheritance
- Multiple inheritance is not supported in TypeScript

```
class Person {
  protected name: string
}

class Employee extends Person {
  private id: number
}
```

# Method Overriding

- It is a way subclass can provide different implementation of a method than what super class has provided
- In TypeScript use the same method signature as that of super class to override the method

```
class Person {
  protected name: string

  printInfo() {
    console.log(`name: ${this.name}`)
  }
}

class Employee extends Person {
  private id: number

  printInfo() {
    console.log(`id: ${this.id}`)
    super.printInfo()
  }
}
```

# Abstract Class

- Define an abstract class in Typescript using the abstract keyword
- Abstract classes are mainly for inheritance where other classes may derive from them
- Abstract class can not be instantiated
- An abstract class typically includes one or more abstract methods or property declarations
- The class which extends the abstract class must define all the abstract methods

```
abstract class Animal {
  abstract makeSound()

  move() {
    console.log("roaming the earth...")
  }
}

class Lion extends Animal {

  makeSound() {
    console.log('roar...')
  }

}
```

# Generics

- Generics offer a way to create reusable components
- Generics provide a way to make components work with any data type and not restrict to one
- With Generics, components can be called or used with a variety of data types

```
class Stack<T> {
  items: T[] = []

  push(item: T) { this.items.push(item) }
  pop() { return this.items.splice(this.items.length - 1, 1) }
}

const stack = new Stack<number>()
stack.push(10)
stack.push(20)
stack.pop()
```

# Namespace

- The namespace is used for logical grouping of functionalities
- A namespace can include interfaces, classes, functions and variables to support a single or a group of related functionalities
- A namespace can be created using the namespace keyword followed by the namespace name
- All the interfaces, classes etc. can be defined in the curly brackets { }

```
namespace UtilityFunctions {

  function toJSONString(object: Object) {
    return JSON.stringify(object)
  }

}
```

# Interface

# Interface

- Contract between service provider and servicer consumer
- Provides barrier between provider and consumer
- It is a collection of abstract methods
- Class uses a keyword implements to implement interface
- The class must provide the implementation of every method declared in the interface

```
interface IShape {
  erase()
  draw()
}

class Shape implements IShape {
  erase() { console.log('shape is erasing') }
  draw() { console.log('shape is drawing') }
}
```

# Interface as Type

- Interface in TypeScript can be used to define a type and also to implement it in the class

```
interface KeyPair {
  key: number;
  value: string;
}

let kv1: KeyPair = { key: 1, value: "Steve" }

// Compiler Error: value is missing
// let kv2: KeyPair = { key:1}

// Compiler Error: value type is not matching
// let kv3: KeyPair = { key:1, value: 100 }
```

# Interface extension

- Interfaces can extend one or more interfaces
- This makes writing interfaces flexible and reusable

```
interface IPerson {
  name: string
  gender: string
}

interface IEmployee extends IPerson {
  empCode: number
}

let empObj: IEmployee = {
  empCode: 1,
  name: "Bill",
  gender: "Male"
}
```