**TYPESCRIPT OOPS**

Object-oriented programming is a programming paradigm based on classes and objects rather than functions and logic. The software programs in object-oriented programming are structured into reusable pieces of code known as classes.

Although TypeScript is a superset of JavaScript, object-oriented programming in TypeScript differs from Object-oriented programming in JavaScript because, unlike JavaScript, TypeScript has full class support, has access modifiers, and type annotations like most object-oriented programming languages.

In this article, you will explore object-oriented programming in TypeScript and have an overview of its implementation with the three principles of object-oriented programming: Inheritance, Encapsulation, and Polymorphism.

## Classes

A class is a blueprint used to create an instance of an object. It is made up of variables (called instance variables) and methods.

Every object instantiated from a class will have all the properties of the class that instantiated it.

For example:

class Car {

model: string;

year: number;

p rice: string;

drive () {

console.log('The Car has Started driving');

}

stop() {

console.log('The car has stopped');

}

}

Objects are instantiated from a class using the new keyword, and any object instantiated from a class will contain all the class properties.

For example:

const tesla = new Car();

The object tesla is an instance of the Car class. Thus, it contains all the properties present in the Car class. The properties can be accessed and set using the dot (.) Operator.

For example:

tesla.model= "Model X"

tesla.year= 2022

tesla.price="$114,990"

tesla.drive()

tesla.stop()

Classes are used in object-oriented programming to avoid code duplication, create and manage new objects, and support inheritance.

## Constructor Functions

A constructor function is a class function responsible for initializing a class’s instance variables.

Constructors in TypeScript are defined using the constructor keyword. The constructor function takes all class’s instance variables as parameters, initializes them, and assigns their values in its body.

class A {

variable: string;

constructor(variable: string){

this.variable = variable

}

}

const object = new A('value')

When a new object is instantiated from A, the values of the class’s instance variables are specified as arguments.

## Inheritance

Inheritance in object-oriented programming refers to a mechanism where a class (subclass) inherits properties from an existing class (superclass).

The subclass can also extend functionality by adding new properties or methods.

For example, consider the class below as the superclass:

class Person {

name: string;

age: number;

constructor(name:string, age:number){

this.name = name;

this.age = age;

}

eat() {

console.log(`What's for dinner?`);

}

speak() {

console.log(`My name is ${this.name}, I am ${this.age} years old`);

}

}

To inherit from a class (superclass), the extends keyword is used by affixing it after the sub class's name followed by the superclass's name.

Note that if the superclass has properties defined in its constructor, the subclass has to initialize these properties in its constructor using the super keyword. The super keyword is used to reference a superclass properties in a subclass.

For example:

class Chef extends Person {

occupation: string;

constructor(name:string, age:number, occupation: string){

super(name, age)

this.occupation = occupation;

}

speak(): void {

console.log(`I a m a ${this.occupation}`);

}

cook() {

console.log(`I am cooking`);

}

}

The Chef class inherits all the properties present in the Person, like the name and age instance variables and the eat and speak methods.

Sometimes, a subclass needs to follow a super class's implementation but not inherit its properties. These cases require the implements keyword instead of the extends keyword.

## Extends vs Implements

The extends keyword enables the subclass to benefit from inheritance, giving it access to the properties and methods of its superclass.  
The implements keyword, however, treats the superclass as an interface, ensuring that the subclass conforms to the shape of its superclass.

An interface is a TypeScript structure that acts as a contract by enforcing a particular shape on a class or a specific type on a function or variable.  
Classes that “implement” another class must declare all the properties present in the class they implement.

For example:

class Human {

name: string;

gender: string;

constructor(name:string, gender:string){

this.name = name;

this.gender = gender;

}

speak() {

return `I am speaking`;

}

}

class Doctor implements Human {

name: string;

gender: string;

constructor(name:string, gender:string){

this.name = name;

this.gender = gender;

}

speak() {

return 'I am a doctor';

}

}

If the subclass that implements a superclass doesn’t completely mirror its superclass, TypeScript will throw an error.

## Overriding and Extending Inherited Properties

When sub classes inherit properties and methods from their superclass, the inherited properties can be modified or extended. This process of modifying an inherited property is known as overriding.

Overriding is implemented if a subclass has to execute logic that differs from that of its superclass when the same method is invoked.

A superclass's property can be overridden by re-declaring the same property in a subclass.

For example:

class A {

print() {

console.log('I am class A');

}

}

class B extends A {

print() {

console.log('I am class B');

}

}

Other scenarios might exist where the functionality needs to be “extended,” not completely overridden. In these scenarios, you must call the method with the super keyword first, then implement its new functionality.

For example:

class B extends A {

print() {

super.print(); //I am class A

console.log('I am class B');

}

}

const object = new B();object.print();// I am class A, I am class B

The super.<method>() method first executes the command from the superclass and then executes the command on the subclass.

## Deadly Diamond of Death

Multiple inheritance refers to a subclass inheriting from more than one superclass; this leads to a problem known as the deadly diamond of death.

The deadly diamond of death is a problem that arises when two classes inherit from one superclass, and another class inherits from the child classes that are under the previously created superclass.

class A {};class B extends A {};class C extends A {};class D extends B, C {};// This will throw an error

For context, assume that B and C override a method inherited from A, and then the method is called on an object of D. Which method will be executed? A's method, B's method, or C’s method?

The workaround for multiple inheritance is using interfaces instead of classes, so the subclass doesn't “extend” the superclasses; rather, it “implements” them.

For example:

class A {};interface B extends A {};interface C extends A {};class D implements B, C {};

Although, this implementation only ensures that the subclass takes the shape of its superclass's which can be termed Polymorphism. It is the most viable solution.

## Encapsulation

Encapsulation in object-oriented programming refers to restricting unauthorized access and mutation of specific properties of an object.

In TypeScript, access modifiers are used to achieve encapsulation.

By default, all class properties in a class are public. This default behavior can be overridden by prefixing the properties with access modifiers.

### Access Modifiers

An access modifier is a keyword that changes the accessibility of a property or method in a class.

There are three primary access modifiers in TypeScript:

* public: This is the default visibility of every class property. A public property is accessible outside the class.
* private: A property prefixed with the private keyword can’t be accessed anywhere outside the class and cannot be inherited by a subclass.
* protected: The protected access modifier is very similar to the private access modifier with one key difference. Properties marked with the protected keyword are visible and can be inherited by a subclass.

In addition to the main three, TypeScript has two more access modifiers:

* static: Properties or methods prefixed with static can only be accessed directly on the class and not on an object instantiated from the class. They also can’t be inherited.

For example:

class A {

static index: number = 1;

};

A.index; // 1

Note that static properties and methods can't reference the This keyword unless the referenced property is static.

* read only: Properties prefixed with read only can’t be modified; their values can only be read. Since read-only properties cannot be modified, they must be set at the class declaration or inside a constructor function.

### Initializing Instance Variables with Access Modifiers

TypeScript provides a shorthand method of initializing instance variables in the constructor. The shorthand method involves declaring the variable once as a parameter in the constructor and prefixing the instance variable with an access modifier.

For example:

class A {

constructor(public variable: string){}

}

const object = new A('value')

This method is ideal for classes with a few instance variables as it can quickly get messy and hard to read with multiple instance variables.

Prefixing the properties with specific access modifiers prevents them from being accessed outside the class, which makes it impossible to read or set their values outside the class. This issue is solved using **getters** and **setters**, which allow you to read and write inaccessible properties outside the class by implementing accessible methods inside the class.

### Setters and Getters

A setter is a method inside a class that sets the value of an instance variable.

A getter is a method inside a class that returns the value of an instance variable.

Setters and getters are implemented to add some logic between the reading and writing of properties. Since they are methods, the conditions must be fulfilled before mutation occurs or the property's value can be read.

In TypeScript, setters are implemented using the set keyword, and getters are implemented using the get keyword.

For example:

class A {

private \_variable: string;

constructor(variable:string){

this.\_variable = variable;

}

get variable(): string {

return this.\_variable;

}

set variable(value: string) {

if(value === '') throw new Error("Variable cannot be an empty string");

this.\_variable = value;

}

}

const object = new A('string')

//setting the variableobject.variable = 'different string'

//getting the variableconsole.log(object.variable)

Encapsulation plays a considerable role in object-oriented programming. It prevents unauthorized access to an object's properties, giving you better control over properties and methods, thereby increasing code quality and making code easier to maintain.

## Polymorphism

Polymorphism in object-oriented programming refers to a situation where multiple classes inherit from a parent and override a particular functionality, i.e. the ability of a method or property to exist in different forms.

When you override inherited methods or properties, that's polymorphism.

For example:

class A {

name: string = "Class A"

print(){

console.log('I am class A')

}

}

class B extends A {

name: string = "Class B"

print(){

console.log('I am class B')

}

}

The name property and the print method exist in different forms in each class.

Implementing Polymorphism improves code quality and re usability by allowing you to perform the same action differently.

## Conclusion

In this tutorial, you went over the pillars of object-oriented programming:

Inheritance, Encapsulation, Polymorphism, and Abstraction while going into detail about the deadly diamond of death, setters and getters, method overriding, and the implementation of abstract classes.

The importance of object-oriented programming cannot be over-emphasized as it makes maintaining and reusing code very easy.