**JavaScript Object-oriented programming**

TypeScript is an object-oriented superset of JavaScript, which means it incorporates many Object-Oriented Programming (OOP) features. Here are some of the key OOP features in TypeScript along with examples:

**Classes:**

TypeScript supports the definition of classes, which can be used to create objects with properties and methods.

```typescript

class Person {

constructor(public name: string, public age: number) {}

sayHello() {

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

}

}

const person = new Person("Alice", 30);

person.sayHello(); // Output: Hello, my name is Alice and I am 30 years old.

```

**Inheritance:**

TypeScript allows classes to inherit properties and methods from other classes.

```typescript

class Student extends Person {

constructor(name: string, age: number, public studentId: string) {

super(name, age);

}

study() {

console.log(`${this.name} is studying.`);

}

}

const student = new Student("Bob", 20, "S12345");

student.sayHello(); // Output: Hello, my name is Bob and I am 20 years old.

student.study(); // Output: Bob is studying.

```

**Encapsulation:**

TypeScript supports access modifiers like `public`, `private`, and `protected` to control the visibility of class members.

```typescript

class BankAccount {

private balance: number = 0;

constructor() {}

deposit(amount: number) {

if (amount > 0) {

this.balance += amount;

}

}

withdraw(amount: number) {

if (amount > 0 && amount <= this.balance) {

this.balance -= amount;

}

}

getBalance() {

return this.balance;

}

}

const account = new BankAccount();

account.deposit(100);

account.withdraw(50);

console.log(account.getBalance()); // Output: 50

```

**Abstraction:**

TypeScript allows you to define abstract classes and methods that must be implemented by derived classes.

```typescript

abstract class Shape {

abstract calculateArea(): number;

}

class Circle extends Shape {

constructor(private radius: number) {

super();

}

calculateArea(): number {

return Math.PI \* this.radius \* this.radius;

}

}

const circle = new Circle(5);

console.log(circle.calculateArea()); // Output: 78.53981633974483

```

**Polymorphism:**

TypeScript supports polymorphism, which allows objects of different classes to be treated as objects of a common superclass.

```typescript

function printArea(shape: Shape) {

console.log(`Area: ${shape.calculateArea()}`);

}

const circle = new Circle(5);

const square = new Square(4);

printArea(circle); // Output: Area: 78.53981633974483

printArea(square); // Output: Area: 16

```

These are some of the core Object-Oriented Programming features in TypeScript. TypeScript allows you to write more structured and maintainable code by leveraging these features to create reusable and extensible classes and objects.

In JavaScript, there are several ways to create classes, each with its own syntax and features. Here are the primary ways to create classes in JavaScript:

1. \*\*Constructor Functions:\*\*

You can create classes using constructor functions. These functions are typically named with an initial capital letter by convention.

```javascript

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.sayHello = function () {

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

};

const person = new Person("Alice", 30);

person.sayHello();

```

2. \*\*ES5 Prototype-based Classes:\*\*

You can define classes using the prototype-based approach, where you attach methods to the prototype of a constructor function.

```javascript

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.sayHello = function () {

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

};

const person = new Person("Alice", 30);

person.sayHello();

```

**ES6 (ES2015) Classes:**

ECMAScript 2015 introduced a class syntax that provides a more concise and familiar way to create classes.

```javascript

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

sayHello() {

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

}

}

const person = new Person("Alice", 30);

person.sayHello();

```

**Factory Functions:**

You can create classes using factory functions that return objects with methods.

```javascript

function createPerson(name, age) {

return {

name,

age,

sayHello() {

console.log(`Hello, my name is ${name} and I am ${age} years old.`);

}

};

}

const person = createPerson("Alice", 30);

person.sayHello();

```

**Object.create():**

You can create objects using `Object.create()` to set the prototype of an object explicitly.

```javascript

const personPrototype = {

sayHello() {

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

}

};

const person = Object.create(personPrototype);

person.name = "Alice";

person.age = 30;

person.sayHello();

```

**Class Expressions:**

You can also use class expressions to create classes, which are similar to class declarations but can be unnamed.

```javascript

const Person = class {

constructor(name, age) {

this.name = name;

this.age = age;

}

sayHello() {

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

}

};

const person = new Person("Alice", 30);

person.sayHello();

```

Each of these approaches has its own advantages and use cases, and the choice of which one to use depends on your project requirements and coding style preferences. ES6 classes are the most modern and widely adopted way to create classes in JavaScript, but the other methods are still relevant in certain situations, especially when working with legacy code or specific design patterns.