Типизация в TypeScript представляет собой одну из ключевых особенностей этого языка и особенно важна при работе с объектно-ориентированным программированием (ООП). TypeScript обеспечивает строгую статическую типизацию, что делает разработку более предсказуемой, упрощает тестирование и предотвращает большое количество ошибок на этапе написания кода. Ниже я подробно объясню, как типизация работает в рамках парадигмы ООП.

**Основные концепции ООП в TypeScript**

TypeScript поддерживает все основные принципы ООП:

Классы (Classes)

Инкапсуляция (Encapsulation)

Наследование (Inheritance)

Полиморфизм (Polymorphism)

Абстракция (Abstraction)

Также TypeScript предоставляет мощную систему типизации, включая интерфейсы, типы и другие механизмы, для работы с объектами, классами и их свойствами.

**1. Классы и типы в TypeScript**

Классы можно рассматривать как шаблоны для создания объектов. TypeScript, в отличие от JavaScript, позволяет статически описывать типы свойств и параметров методов внутри классов.

class Person {

  name: string; // тип свойства "строка"

  age: number; // тип свойства "число"

  constructor(name: string, age: number) {

    this.name = name; // инициализация имени

    this.age = age; // инициализация возраста

  }

  greet(): string {

    return Привет! Меня зовут ${this.name};

  }

}

// Пример использования

const person: Person = new Person("Иван", 25);

console.log(person.greet()); // Привет! Меня зовут Иван

Особенности:

name: string и age: number — это явное указание типов для свойств.

Тип string указывается для метода greet, чтобы TypeScript мог проверять возвращаемое значение.

Преимущества:

Благодаря статической типизации можно сразу обнаружить ошибки, например, при попытке присвоить неверный тип данных:

    person.name = 123; // Ошибка: Type 'number' is not assignable to type 'string'

**2. Инкапсуляция и модификаторы доступа**

TypeScript поддерживает модификаторы видимости для свойств и методов:

public (по умолчанию): свойство/метод доступен везде.

private: доступ только внутри класса.

protected: доступ разрешен внутри класса и в наследуемых классах.

readonly: свойство доступно только для чтения, после инициализации его изменение запрещено.

Пример:

class BankAccount {

  private balance: number;

  constructor(initialBalance: number) {

    this.balance = initialBalance;

  }

  public deposit(amount: number): void {

    this.balance += amount;

  }

  public getBalance(): number {

    return this.balance;

  }

}

const account = new BankAccount(1000);

account.deposit(500);

// console.log(account.balance); // Ошибка: Свойство "balance" недоступно из-за модификатора "private"

console.log(account.getBalance()); // 1500

Особенности:

Свойство balance скрыто (private), поэтому оно недоступно извне напрямую.

Члены класса можно сделать только для чтения с помощью readonly.

**3. Наследование и расширение классов**

TypeScript позволяет создавать наследуемые классы с помощью ключевого слова extends. При наследовании потомок получает доступ ко всем публичным и защищённым (protected) методам и полям родителя.

Пример:

class Animal {

  protected name: string;

  constructor(name: string) {

    this.name = name;

  }

  public makeSound(): void {

    console.log(${this.name} издает звук);

  }

}

class Dog extends Animal {

  public bark(): void {

    console.log(${this.name} лает);

  }

}

const dog = new Dog("Шарик");

dog.makeSound(); // Шарик издает звук

dog.bark(); // Шарик лает

**4. Абстракция**

TypeScript поддерживает абстрактные классы и методы. Абстрактные классы не могут быть инстанцированы (создавать их экземпляры), но могут быть унаследованы. Абстрактные классы часто используются для создания общих интерфейсов.

Пример:

abstract class Shape {

  abstract getArea(): number; // Абстрактный метод (должен быть реализован в наследниках)

}

class Circle extends Shape {

  private radius: number;

  constructor(radius: number) {

    super();

    this.radius = radius;

  }

  public getArea(): number {

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

  }

}

const circle = new Circle(10);

console.log(circle.getArea()); // 314.159...

**5. Интерфейсы (Interfaces)**

Интерфейсы в TypeScript часто используются для определения типов объектов или описания сигнатур, которые классы должны реализовать. Интерфейсы поддерживают множественное наследование.

Пример:

interface Animal {

  name: string;

  age: number;

  makeSound(): void;

}

class Cat implements Animal {

  name: string;

  age: number;

  constructor(name: string, age: number) {

    this.name = name;

    this.age = age;

  }

  makeSound(): void {

    console.log("Мяу!");

  }

}

const cat = new Cat("Мурка", 2);

cat.makeSound(); // Мяу!

Особенности:

Интерфейсы определяют контракт, который должен быть реализован классом.

Классы могут реализовывать сразу несколько интерфейсов.

**6. Полиморфизм**

Полиморфизм достигается с использованием наследования и реализации интерфейсов. Родительский класс или интерфейс можно использовать для создания объектов разных типов.

Пример:

abstract class Shape {

  abstract getArea(): number;

}

class Rectangle extends Shape {

  constructor(private width: number, private height: number) {

    super();

  }

  public getArea(): number {

    return this.width \* this.height;

  }

}

class Circle extends Shape {

  constructor(private radius: number) {

    super();

  }

  public getArea(): number {

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

  }

}

const shapes: Shape[] = [

  new Rectangle(10, 20),

  new Circle(5),

];

shapes.forEach(shape => {

  console.log(shape.getArea()); // Вызывает метод "getArea" с разными реализациями

});

**Расширенные возможности TypeScript для ООП**

TypeScript добавляет дополнительные возможности к классическому ООП:

**Геттеры и сеттеры:**

class Employee {

  private \_salary: number;

  constructor(salary: number) {

    this.\_salary = salary;

  }

  get salary(): number {

    return this.\_salary;

  }

  set salary(amount: number) {

    if (amount = new (...args: any[]) => T;

Статические свойства и методы:

class MathUtils {

  static readonly PI = 3.14;

  static calculateArea(radius: number): number {

    return this.PI \* radius \* radius;

  }

}

console.log(MathUtils.calculateArea(10)); // 314

**Заключение**

TypeScript предоставляет мощные инструменты для работы с ООП, значительно улучшая JavaScript в этой области за счет типизации, интерфейсов, модификаторов доступа и других механизмов. Возможности TypeScript, такие как интерфейсы, абстрактные классы, статическая типизация и строгий контроль типов, делают его отличным выбором для реализации крупных и сложных приложений в стиле ООП