**Классы и объектно-ориентированное программирование в TypeScript**

TypeScript, как надстройка над JavaScript, поддерживает объектно-ориентированное программирование (ООП), что делает его мощным инструментом для разработки сложных и масштабируемых приложений. Одной из ключевых концепций ООП являются классы, которые позволяют создавать объекты, инкапсулировать данные и поведение, а также организовывать код таким образом, чтобы он был легко поддерживаемым и расширяемым. В этой лекции мы рассмотрим, как в TypeScript реализуются классы, какие возможности предоставляет язык для использования ООП и как эффективно применять эти концепции на практике.

**Основы классов в TypeScript**

Классы в TypeScript представляют собой шаблоны для создания объектов. Каждый класс может содержать свойства и методы, которые будут унаследованы всеми экземплярами этого класса. Классы TypeScript имеют синтаксис, аналогичный классам в других объектно-ориентированных языках программирования, таких как Java, C# или Python. В TypeScript классы поддерживают наследование, инкапсуляцию, полиморфизм и абстракцию — все основные принципы ООП.

Классы в TypeScript объявляются с использованием ключевого слова class. Внутри класса можно определять конструктор, который используется для инициализации объектов, а также методы, которые описывают поведение объекта.

**Конструктор**

Конструктор — это специальный метод, который автоматически вызывается при создании нового объекта класса. В TypeScript конструкторы объявляются с помощью ключевого слова constructor. Они могут принимать параметры, которые будут использоваться для инициализации свойств объекта.

**Свойства и методы**

Свойства класса представляют собой переменные, которые хранят данные, связанные с объектом. Эти свойства могут быть определены с разными уровнями доступа — публичными, приватными или защищенными. Публичные свойства доступны из любой части программы, приватные — только внутри самого класса, а защищенные — только внутри класса и его подклассов.

Методы класса — это функции, которые выполняют действия с объектом. Методы могут быть обычными функциями или геттерами и сеттерами, которые позволяют контролировать доступ к свойствам.

**Наследование**

Наследование — это механизм, позволяющий создавать новые классы на основе существующих. В TypeScript наследование реализуется с помощью ключевого слова extends. Классы-потомки наследуют все свойства и методы родительского класса, но могут добавлять свои собственные свойства и методы или изменять поведение унаследованных.

При наследовании в TypeScript важно использовать ключевое слово super для вызова конструктора родительского класса и доступа к его методам. Это позволяет подклассу не только унаследовать, но и расширять функциональность родительского класса.

**Модификаторы доступа**

Типичный принцип инкапсуляции ООП предполагает скрытие внутреннего состояния объекта и предоставление к нему доступа только через методы. В TypeScript это можно реализовать с помощью модификаторов доступа. Есть три основных модификатора доступа:

1. **public** — по умолчанию все свойства и методы класса в TypeScript являются публичными, что означает, что они доступны извне. Это позволяет легко обращаться к объекту и изменять его свойства.
2. **private** — свойства и методы, объявленные с этим модификатором, доступны только внутри самого класса. Это позволяет скрыть детали реализации и предотвратить нежелательное изменение состояния объекта.
3. **protected** — свойства и методы, доступные только внутри класса и его подклассов. Этот модификатор полезен, если нужно ограничить доступ к определенным частям кода, но все же предоставить возможность изменять поведение в наследующих классах.

**Абстракция и абстрактные классы**

Абстракция — это принцип ООП, при котором детали реализации скрыты от пользователя, а предоставляется только необходимый интерфейс для работы. В TypeScript абстракцию можно реализовать с помощью абстрактных классов.

Абстрактный класс — это класс, который нельзя инстанциировать напрямую, но который может служить основой для других классов. В абстрактных классах можно определять абстрактные методы, которые должны быть реализованы в подклассах. Эти методы не имеют реализации в абстрактном классе, но обязательно должны быть реализованы в классах, наследующих этот абстрактный класс.

**Полиморфизм**

Полиморфизм — это принцип ООП, который позволяет объектам разных классов иметь одинаковые интерфейсы, но реализовывать их по-разному. В TypeScript полиморфизм проявляется в способности объектов разных типов отвечать на одинаковые вызовы методов, но с разным поведением.

Полиморфизм реализуется через переопределение методов в подклассах. Когда подкласс переопределяет метод родительского класса, он может изменить поведение метода, сохраняя тот же интерфейс.

**Интерфейсы и классы**

В TypeScript интерфейсы играют важную роль в объектно-ориентированном программировании. Интерфейсы могут использоваться для задания контракта, который должны соблюдать классы. Классы могут реализовывать интерфейсы, что означает, что они обязаны реализовать все свойства и методы, определенные в этом интерфейсе. Это позволяет достичь дополнительной гибкости и модульности в проектировании.

**Пример**

Для того чтобы продемонстрировать основные концепции объектно-ориентированного программирования в TypeScript, рассмотрим пример, в котором создадим класс, используем наследование, инкапсуляцию и полиморфизм.

// Абстрактный класс

abstract class Animal {

  constructor(public name: string) {}

  // Абстрактный метод

  abstract speak(): void;

}

// Класс Dog наследует Animal

class Dog extends Animal {

  constructor(name: string, public breed: string) {

    super(name); // Вызов конструктора родительского класса

  }

  // Реализация абстрактного метода

  speak(): void {

    console.log(`${this.name} says Woof!`);

  }

}

// Класс Cat наследует Animal

class Cat extends Animal {

  constructor(name: string, public color: string) {

    super(name); // Вызов конструктора родительского класса

  }

  // Реализация абстрактного метода

  speak(): void {

    console.log(`${this.name} says Meow!`);

  }

}

// Использование классов

const dog = new Dog('Buddy', 'Golden Retriever');

const cat = new Cat('Whiskers', 'Black');

dog.speak(); // Выводит: Buddy says Woof!

cat.speak(); // Выводит: Whiskers says Meow!