САМАРСКИЙ УНИВЕРСИТЕТ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Лекция № 3

ООП. Интерфейсы и классы. Наследование. Обобщения. Примеси

2023

РУБРИКА: «ВОПРОСЫ ПО ЛЕКЦИИ № 2»

Предисловие об ООП

Что такое ООП?

Объектно-ориентированное программирование – методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Чем полезна такая методология?

Данная методология позволяет максимально приблизить процесс моделирования на языке программирования к описанию некоторого реального механизма, процесса, бизнес-логики с некоторым уровнем абстракции, наследования элементов, использования полиморфизма и правил инкапсуляции.

Что такое Инкапсуляция?

Это процесс сокрытия элементов реализации от абстракции

Зачем она нужна?

Расскажете свой PIN-код от банковской карты кому-нибудь? Правильно, нет! Потому что там ваши сбережения и деньги на каждый день, которые могут быть украдены, потрачены и так далее...
Можно сказать, что каждая банковская карта внутри себя инкапсулирует метод проверки PIN-кода, класс его защищенного хранилища и разрешает подать на вход только код, который вам известен без раскрытия действительного кода, запускающего механизм подписи при оплате покупки.

Что такое Наследование?

Это механизм наложения объекта или класса на другой объект или класс (наследование на основе классов) с сохранением аналогичной реализации.

Зачем оно нужно?

Допустим у нас есть класс Птица. Слишком абстрактно... Но у каждой птицы есть свои параметры: цвет, ареал обитания и т.д.
Не все птицы летают, поэтому нам нужен отдельный класс, реализующий логику для летающих пернатых и для нелетающих
Нелетающие в свою очередь делятся на сухопутных, водоплавающих и так далее.
Каждый класс будет реализовывать методы и свойства, присущие родительскому, таким образом достигая отсутствия нового описания для каждого класса, что и есть принцип наследования.

Что такое Полиморфизм?

Это механизм, при котором некоторые классы выглядят одинаково в абстракции, но ведут себя по разному внутри (множество реализации одного интерфейса разными классами, переопределение метода в классе-наследнике).

Зачем оно нужно?

Представим, что у нас есть интерфейс Нагревательный прибор.
Опять же, кто-то подумал про тепловентилятор, кто-то про радиатор, а кто-то про инфракрасный излучатель.
Все эти приборы греют, но реализация процесса нагревания у каждого разная.
Для простоты Вам как пользователю доступна только одна функция для каждого типа прибора - включить и выключить питание. Сам же процесс мало кого-либо интересует, да и в реализацию лазать нельзя без соответствующих знаний и квалификации во избежание травм или поломки.

Что такое Класс?

это расширяемый шаблон программного кода для создания объектов, предоставляющий начальные значения состояния (переменные-члены) и реализации поведения (функции-члены или методы)

Что такое Интерфейс?

это расширяемый шаблон программного кода для создания прототипа методов (и в некотором случае свойств), который в дальнейшем может быть реализован одним или более классом.

```
class Car {
  private _id: number;
  private name: string;
  public model: string;

  get id() {
    return this._id;
  }

   set id(id: number) {
    this._id = id;
  }

  constructor(carId: number, carModel: string, model: string) {
    this.id = carId;
    this.name = carModel;
    this.model = model;
  }

  getCarInfo(): string {
    return "Car model = " + this.name + " model= " + this.model;
  }
}
```

Модификаторы доступа: это специальные конструкции класса, позволяющие сокрыть необходимые свойства и методы от других элементов, кому эти данные знать не нужно.
Модификаторы бывают 3х типов:

- private – нельзя напрямую обратиться, недоступен к использованию извне класса
- protected - определяет поля и методы, которые из вне класса видны только в классах-наследниках
- public – определяет доступ отовсюду, где он затребован.

Геттеры (get) и Сеттеры (set) – это специальные методы класса, позволяющие считывать в нужном формате или модифицировать данные перед сохранением их в определенное поле или поля класса. Могут быть использованы для построения принципа инкапсуляции

Конструктор – это специальный метод класса, вызываемый при создании объекта для его первичной инициализации.

Классы и интерфейсы :: Пример класса и интерфейса класса

```
interface IUser {
    id: number;
    name: string;
    getFullName(surname: string): string;
}

class User implements IUser{

    id: number;
    name: string;
    age: number;
    constructor(userId: number, userName: string, userAge: number) {

        this.id = userId;
        this.name = userName;
        this.age = userAge;
    }
    getFullName(surname: string): string {

        return this.name + " " + surname;
    }
}

let tom = new User(1, "Tom", 23);
console.log(tom.getFullName("Simpson"));
```

Абстрактные классы представляют классы, определенные с ключевым словом abstract. Они во многом похожи на обычные классы за тем исключением, что мы не можем создать напрямую объект абстрактного класса, используя его конструктор.

```
abstract class Figure {
    getArea(): void{
        console.log("Not Implemented")
    }
}
class Rectangle extends Figure{

    constructor(public width: number, public height: number){
        super();
    }

    getArea(): void{
        let square = this.width * this.height;
        console.log("area =", square);
    }
}

let someFigure: Figure = new Rectangle(20, 30)
someFigure.getArea();   // area = 600
```

Классы и интерфейсы :: Пример расширения и наследования интерфейсов

Расширение интерфейса

```
interface IUser {
   id: number;
   name: string;
   getFullName(surname: string): string;
}

interface IUser {
   surname: number;
}
```

Наследование интерфейса

```
interface IMovable {

   speed: number;
   move(): void;
}

interface ICar extends IMovable {

   fill(): void;
}
```

```
class User<D> implements EmployeeDepartment<D> {
  private _id: number;
  private fio: string;
  private phone: string;
  private department: D;

  /* Геттеры и сеттеры */

  constructor(carId: number, carModel: string, model: string,
department: D) {
    this.id = carId;
    this. fio = carModel;
    this.model = model;
    this.department = department;
  }

  getInfo(): string {
    return null;
  }
}
```

```
interface ITDepartment extends Department {
  teamLeader: string;
}

interface SalesDepartment extends Department {
   priceController: string;
}
interface Department {
  id: number;
  name: string;
  manager: string;
}
interface EmployeeDepartment<D extends Department>
{
  id: number;
  fio: string;
  phone: string;
  department: D
}
```

Обобщения :: Множество параметров

Таких параметров может быть много и они могут быть зависимы друг от друга

function identity<T, D> (param1: T, param2: D): T //независимые

function identity<T, D extends keyof T> (param1: T, param2: D): D //А здесь передаваемый тип D зависим от ключа типа T. т.е. что зададим тип для 1го, то повлияет на решение о допустимости 2го типа, что впоследствии будет обработано компилятором и в случае наличия несоответствия приведет к ошибке при компиляции.

Что такое Примеси?

TypeScript, как и многие объектно-ориентированные языки, как, например, Java или C#, не позволяет использовать напрямую множественное наследование. Мы можем реализовать множество интерфейсов в классе, но унаследовать его можем только от одного класса. Однако функциональность миксинов (mixins) частично позволяют унаследовать свойства и методы сразу двух и более классов.

Пусть, у нас есть класс Animal, который представляет животное, и класс Movable, который представляет транспортное средство. Оба эти класса имеют свой уникальный функционал, который позволяет выполнять заложенные в них задачи. И также пусть у нас будет класс, который представляет лошадь - с одной стороны, лошадь является животным и наследует все черты, присущие животному, а с другой стороны, лошадь также можно использовать в качестве транспортного средства. То есть для создания подобного класса было бы неплохо унаследовать его сразу и от класса Animal, и от класса Movable

Примеси :: Пример

```
class Animal {

    feed():void {
        console.log("Кормим животное");
    }
}

class Movable {

    speed: number=0;
    move(): void {
        console.log("Перемещаемся");
    }
}

class Horse {}

interface Horse extends Animal, Movable {}

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {

Object.getOwnPropertyNames(baseCtor.prototype).forEach(name =>
{
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}

applyMixins(Horse, [Animal, Movable]);

let pony: Horse = new Horse();
pony.feed();
pony.move();
```

РУБРИКА: «ВОПРОСЫ СЛУШАТЕЛЕЙ»

КОНЕЦ ЛЕКЦИИ № 3
СПАСИБО ЗА ВНИМАНИЕ