Plan

2023-10-17

1. Interfaces

Theory

▶ **English**

▼ **На русском**

1. Введение в интерфейсы:

- **Что такое интерфейсы в Java?**
  - В Java интерфейс является абстрактным типом данных, представляющим собой набор абстрактных методов (без реализации) и, возможно, констант (статических и финальных переменных). Основное назначение интерфейсов - определение контрактов для классов.
- **Зачем нужны интерфейсы?**
  - Интерфейсы предоставляют способ реализации полиморфизма и установления связей между классами, не зависимо от их иерархии наследования. Они позволяют определить, какие методы должны быть реализованы в классе, не предоставляя саму реализацию. Это облегчает разработку и поддержку кода.

2. Основные концепции интерфейсов:

- **Объявление интерфейса:**
  - Для объявления интерфейса используется ключевое слово `interface`. Пример объявления интерфейса:

```
interface MyInterface {
    void doSomething(); // Абстрактный метод без реализации
    int MY_CONSTANT = 42; // Константа
}
```

- **Методы в интерфейсе:**
  - Методы, объявленные в интерфейсе, являются абстрактными, то есть они не имеют реализации и не содержат тела метода. Они определяют сигнатуру метода

и его возвращаемый тип.

- **Переменные в интерфейсе:**
  - Интерфейсы также могут содержать переменные, которые автоматически считаются `public`, `static` и `final`. Эти переменные обычно используются для определения констант.
- **Реализация интерфейса в классах:**
  - Чтобы класс мог использовать интерфейс, он должен реализовать (implement) этот интерфейс с помощью ключевого слова `implements`. Класс должен предоставить реализацию всех абстрактных методов, объявленных в интерфейсе.

3. Интерфейсы и наследование:

- **Реализация интерфейсов в классах:**
  - Класс может реализовать (implement) несколько интерфейсов. Для этого используется ключевое слово `implements`. Например:

```
class MyClass implements MyInterface1, MyInterface2 {
    // Реализация методов интерфейсов
}
```

- **Множественная реализация интерфейсов:**
  - Java позволяет классам реализовывать несколько интерфейсов, что открывает возможности для создания гибких иерархий классов и использования разнообразных контрактов.

Практика:

Техническое задание (ТЗ) для приложения управления работниками в IT-фирме:

Общее описание

Наша IT-фирма ищет решение для учета и управления информацией о сотрудниках различных специализаций: разработчики, менеджеры и продажники. Необходимо создать приложение, которое позволит добавлять, удалять, искать и выводить информацию о работниках. Приложение должно быть реализовано на языке Java и иметь пользовательский интерфейс в виде командной строки.

Функциональные требования

1. **Добавление работника**
   - Пользователь должен иметь возможность добавить нового работника с указанием имени, уникального идентификатора, года найма и других

характеристик в зависимости от специализации (например, почасовую ставку для разработчиков, базовую зарплату для менеджеров и продажников).
    - Должны быть проверки на корректность ввода данных (например, неотрицательные значения для зарплаты, корректный год найма и т. д.).
2. **Удаление работника**
    - Пользователь должен иметь возможность удалить работника по его уникальному идентификатору.
    - Если работник с указанным идентификатором не найден, должно быть выведено сообщение об ошибке.
3. **Поиск работника по идентификатору**
    - Пользователь должен иметь возможность найти работника по его уникальному идентификатору и вывести информацию о нем.
4. **Вывод списка всех работников**
    - Пользователь должен иметь возможность вывести список всех работников с их основной информацией (имя, уникальный идентификатор, специализация, зарплата и т. д.).
5. **Расчет зарплаты**
    - Для каждой специализации работников (разработчики, менеджеры и продажники) должен быть реализован расчет зарплаты в соответствии с их характеристиками.
    - Расчет зарплаты для разработчиков: почасовая ставка * количество отработанных часов.
    - Расчет зарплаты для менеджеров: базовая зарплата + бонус за количество проектов.
    - Расчет зарплаты для продажников: базовая зарплата + бонус за количество сделок.
6. **Повышение зарплаты по стажу**
    - Пользователь должен иметь возможность повысить зарплату для работников с опытом от X до Y лет на Z%.
    - Опыт работы вычисляется как разница текущего года и года найма.

Для реализации приложения управления данными о работниках в IT-фирме, нам потребуются следующие классы и интерфейсы:

Классы:

1. **Employee (Интерфейс)**:
    - Описывает общие характеристики для всех работников.
    - Содержит методы, такие как `getName()`, `getId()`, `calculateSalary()`, которые будут реализованы в подклассах.

2. **BaseEmployee (Абстрактный класс)**:
    - Реализует интерфейс `Employee` и предоставляет общую функциональность для всех работников.
    - Содержит общие поля и методы, такие как `name`, `id`, `hireYear`, `salary`, `adjustSalaryByExperience()`.
    - Метод `calculateSalary()` оставлен для реализации в подклассах.
3. **Developer (Класс)**:
    - Подкласс `BaseEmployee`, представляющий разработчика.
    - Содержит специфические поля, такие как `hourlyRate` и `hoursWorked`, и реализует метод `calculateSalary()`.
4. **Manager (Класс)**:
    - Подкласс `BaseEmployee`, представляющий менеджера.
    - Содержит специфические поля, такие как `baseSalary` и `numberOfProjects`, и реализует метод `calculateSalary()`.
5. **Salesperson (Класс)**:
    - Подкласс `BaseEmployee`, представляющий продажника.
    - Содержит специфические поля, такие как `baseSalary` и `numberOfDeals`, и реализует метод `calculateSalary()`.
6. **EmployeeRepository (Класс)**:
    - Реализует интерфейс `EmployeeRepositoryInterface`.
    - Отвечает за хранение, добавление, удаление и поиск работников.
    - Содержит массив работников и методы для управления ими.

Интерфейсы:

1. **EmployeeRepositoryInterface (Интерфейс)**:
    - Определяет методы, которые должны быть реализованы в классе `EmployeeRepository` для управления данными о работниках.
    - Включает методы для добавления, удаления, поиска, получения списка всех работников и получения количества работников.

Такое разделение классов и интерфейсов позволяет соблюдать принципы объектно-ориентированного программирования, такие как инкапсуляция, наследование и полиморфизм. Каждый класс отвечает за конкретную специализацию работников, а интерфейс `EmployeeRepositoryInterface` определяет общий контракт для управления данными о работниках.

Homework

▶ **English**

▼ На русском

1. Реализовать интерфейс EmployeeRepositoryInterface из проекта с урока 30
2.

Code

code/Classwork_30/src/garage/entity/Car.java

```
package garage.entity;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class Car {

    private int id;
    private String modelName;

    public Car(int id, String modelName) {
        this.id = id;
        this.modelName = modelName;
    }
}
```

code/Classwork_30/src/garage/entity/GarageService2.java

```
package garage.entity;

import garage.IGarageService;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class GarageService2 implements IGarageService {

    private final Car[] CARS = new Car[5];

    @Override
    public Car addCar(Car car) {
```

```
            boolean isCarAdded = false;
            for (int i = CARS.length - 1; i >=  0; i--) {
                // установить машину в пустую ячейку (пустая ячепйка это ячейка
                if (CARS[i] == null) {
                    CARS[i] = car; // установка машины в ячейку, значение котор
                    isCarAdded = true; // изменяем значение на true, если машин
                    break; // выход из цикла
                }
            }
            // вывести сообщение если машина не добавлена
            if (!isCarAdded) {
                System.out.println("No place in garage");
            }

            return car;
        }

        @Override
        public Car removeCar(Car car) {
            return null;
        }

        @Override
        public Car findCarById(int id) {
            return null;
        }

        @Override
        public Car[] findByModel(String modelName) {
            return new Car[0];
        }
    }
```

code/Classwork_30/src/garage/GarageApp.java

```
package garage;

import garage.entity.Car;
import garage.entity.GarageService2;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
```

```
public class GarageApp {

    public static void main(String[] args) {
        IGarageService garageService = new GarageService();

        Car carM5 = new Car(1, "M5");
        Car carTT = new Car(2, "TT");
        Car carC200 = new Car(3, "C200");
        Car carS500 = new Car(4, "S500");

        garageService.addCar(carM5);
        garageService.addCar(carTT);
        garageService.addCar(carC200);
        garageService.addCar(carS500);

        IGarageService garageService2 = new GarageService2();
        garageService2.addCar(carM5);
        garageService2.addCar(carTT);
        garageService2.addCar(carC200);
        garageService2.addCar(carS500);
        garageService2.addCar(new Car(5, "A4"));
        garageService2.addCar(new Car(6, "A4"));
    }
}
```

code/Classwork_30/src/garage/GarageService.java

```
package garage;

import garage.entity.Car;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class GarageService implements IGarageService {

    private final Car[] CARS = new Car[3];

    @Override
    public Car addCar(Car car) {
        boolean isCarAdded = false;
        for (int i = 0; i < CARS.length; i++) {
            // установить машину в пустую ячейку (пустая ячепйка это ячейка
```

```
                if (CARS[i] == null) {
                    CARS[i] = car; // установка машины в ячейку, значение котор
                    isCarAdded = true; // изменяем значение на true, если машин
                    break; // выход из цикла
                }
            }
            // вывести сообщение если машина не добавлена
            if (!isCarAdded) {
                System.out.println("Garage is full");
            }
            // если машина не добавленна в гараж (isCarAdded == false), тогда в
            return isCarAdded ? car : null;
        }

        @Override
        public Car removeCar(Car car) {
            return null;
        }

        @Override
        public Car findCarById(int id) {
            return null;
        }

        @Override
        public Car[] findByModel(String modelName) {
            return new Car[0];
        }
    }
```

code/Classwork_30/src/garage/IGarageService.java

```
    package garage;

    import garage.entity.Car;

    /**
     * @author Andrej Reutow
     * created on 17.10.2023
     */
    public interface IGarageService {

        Car addCar(Car car);
```

```
    Car removeCar(Car car);

    Car findCarById(int id);

    Car[] findByModel(String modelName);
}
```

code/Classwork_30/src/interface_lesson/CalculatorAbstract.java

```
package interface_lesson;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public abstract class CalculatorAbstract {

    abstract int add(int a, int b);

    abstract int multiply(int a, int b);

}
```

code/Classwork_30/src/interface_lesson/CalculatorImpl.java

```
package interface_lesson;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class CalculatorImpl implements CalculatorInterface {


    @Override
    public int add(int a, int b) {
        return a + b;
    }

    @Override
    public int multiply(int a, int b) {
        return a * b;
```

```
    }
}
```

code/Classwork_30/src/interface_lesson/CalculatorInterface.java

```
package interface_lesson;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public interface CalculatorInterface {

    public static final int id = 1;

    int add(int a, int b);

    int multiply(int a, int b);

}
```

code/Classwork_30/src/interface_lesson/Main.java

```
package interface_lesson;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class Main {
    public static void main(String[] args) {
        CalculatorInterface calculator = new CalculatorImpl();
    }
}
```

code/Messaging/src/service/EmilMessageService.java

```
package service;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class EmilMessageService implements IMessageService2, IKontaktServic
```

```
    @Override
    public boolean sendMessage(String message) {
        System.out.println("Send message via email: " + message);
        return false;
    }


    @Override
    public String addKontakt(String name) {
        return null;
    }

    @Override
    public String[] getAllKontakts() {
        return new String[0];
    }
}
```

code/Messaging/src/service/IKontaktService.java

```
package service;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public interface IKontaktService {

    String addKontakt(String name);

    String[] getAllKontakts();
}
```

code/Messaging/src/service/IMessageService.java

```
package service;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public interface IMessageService extends IKontaktService { // расширили инт

    // теперь в этом интерфейсе есть так же все методы интерфейса IKontaktS
```

```
    // все классы которые реализуют интерфейс IMessageService обязаны так ж

    boolean sendMessage(String message);

}
```

code/Messaging/src/service/IMessageService2.java

```
package service;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public interface IMessageService2 { // расширили интерфейс IMessageService

    // теперь в этом интерфейсе есть так же все методы интерфейса IKontaktS
    // все классы которые реализуют интерфейс IMessageService обязаны так ж

    boolean sendMessage(String message);

}
```

code/Messaging/src/service/SmsMessageService.java

```
package service;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class SmsMessageService implements IMessageService {

    @Override
    public boolean sendMessage(String message) {
        System.out.println("Send Message via sms: " + message);
        return true;
    }

    @Override
    public String addKontakt(String name) {
        return null;
```

```
    }

    @Override
    public String[] getAllKontakts() {
        return new String[0];
    }
}
```

code/Messaging/src/MessageApp.java

```
import service.EmilMessageService;
import service.IMessageService;
import service.IMessageService2;
import service.SmsMessageService;

/**
 * @author Andrej Reutow
 * created on 17.10.2023
 */
public class MessageApp {

    public static void main(String[] args) {
        IMessageService smsMessageService = new SmsMessageService();
        IMessageService2 emialMessageService = new EmilMessageService();

        smsMessageService.sendMessage("Hello bro!");
        emialMessageService.sendMessage("Sher geehrter Herr Pupkin");


        StringBuilder stringBuilder = new StringBuilder();
        String someStr = "";

        stringBuilder.length();
        someStr.length();

//        CharSequence charSequence = new StringBuilder();
//
//        if (charSequence instanceof StringBuilder) {
//            StringBuilder sb = (StringBuilder) charSequence;
//            sb.append();
//            ((StringBuilder) charSequence).append();
//
//        }
```

```
        }
    }
```