Интерфейсы и абстрактные классы

Лекция 4: Продвинутые концепции ООП

Преподаватель: [Ваше имя]

Группа: 203

Семестр: Осенний 2024

План лекции

1. **Интерфейсы в Java**
2. **Абстрактные классы vs Интерфейсы**
3. **Множественное наследование**
4. **Default методы**
5. **Функциональные интерфейсы**
6. **Практический пример: Игровые интерфейсы**

Интерфейсы в Java

Определение:

Интерфейс — это абстрактный тип, который определяет контракт для классов.

Особенности:

- **Все методы абстрактные** (до Java 8)
- **Все поля статические и финальные**
- **Класс может реализовывать несколько интерфейсов**
- **Интерфейс может наследовать от других интерфейсов**

Синтаксис интерфейса

```
public interface ИмяИнтерфейса {
    // Константы (public static final по умолчанию)
    int MAX_VALUE = 100;

    // Абстрактные методы (public abstract по умолчанию)
    void methodName();
    int calculate(int value);

    // Default методы (Java 8+)
    default void defaultMethod() {
        // реализация по умолчанию
    }
}
```

Пример интерфейса для игры

```
public interface Movable {
    // Константы
    int MAX_MOVEMENT = 3;

    // Абстрактные методы
    void moveTo(Position newPosition);
    boolean canMoveTo(Position position);
    int getMovementRange();

    // Default метод
    default void moveTowards(Position target) {
        // Простая реализация движения к цели
        if (canMoveTo(target)) {
            moveTo(target);
        }
    }
}
```

Реализация интерфейса

```
public class Warrior implements Movable {
    private Position position;
    private int movementRange;

    @Override
    public void moveTo(Position newPosition) {
        if (canMoveTo(newPosition)) {
            this.position = newPosition;
            System.out.println("Warrior moved to " + newPosition);
        }
    }

    @Override
    public boolean canMoveTo(Position position) {
        int distance = this.position.getDistanceTo(position);
        return distance <= movementRange;
    }

    @Override
    public int getMovementRange() {
        return movementRange;
    }
}
```

Множественная реализация интерфейсов

```
public interface Combatable {
    void attack(Unit target);
    void takeDamage(int damage);
    boolean isAlive();
}

public interface Healable {
    void heal(int amount);
    int getMaxHealth();
}

// Класс реализует несколько интерфейсов
public class Paladin implements Movable, Combatable, Healable {
    // Реализация всех методов из всех интерфейсов
    @Override
    public void moveTo(Position newPosition) { /* ... */ }

    @Override
    public void attack(Unit target) { /* ... */ }

    @Override
    public void heal(int amount) { /* ... */ }
}
```

Абстрактные классы vs Интерфейсы

Абстрактные классы:

- Может содержать реализованные методы
- Может иметь конструкторы
- Может иметь поля состояния
- Одиночное наследование

Интерфейсы:

- Только абстрактные методы (до Java 8)
- Не может иметь конструкторов
- Только константы
- Множественная реализация

Когда использовать что?

Используйте абстрактный класс когда:

- Нужна общая реализация для нескольких классов
- Классы имеют общее состояние
- Нужны конструкторы
- Логически связанные классы

Используйте интерфейс когда:

- Нужно определить контракт
- Класс должен реализовывать несколько типов
- Нужна гибкость в иерархии
- Определение поведения без состояния

Default методы (Java 8+)

Особенности:

- **Реализация по умолчанию** в интерфейсе
- **Не нужно переопределять** в реализующих классах
- **Можно переопределить** при необходимости
- **Обратная совместимость**

```
public interface GameEntity {
    String getName();
    Position getPosition();

    // Default метод с реализацией
    default boolean isAt(Position position) {
        return getPosition().equals(position);
    }
```

Наследование интерфейсов

```
public interface Entity {
    String getName();
    Position getPosition();
}

public interface Movable extends Entity {
    void moveTo(Position newPosition);
    boolean canMoveTo(Position position);
}

public interface Combatable extends Entity {
    void attack(Unit target);
    void takeDamage(int damage);
}

// Комбинированный интерфейс
public interface CombatUnit extends Movable, Combatable {
    int getAttackPower();
    int getDefense();
}
```

Функциональные интерфейсы (Java 8+)

Определение:

Функциональный интерфейс — интерфейс с одним абстрактным методом.

Использование:

- **Lambda выражения**
- **Method references**
- **Stream API**

```
@FunctionalInterface
public interface Action {
    void execute();
}
```

Практический пример: Игровые интерфейсы

```
// Базовый интерфейс для всех игровых объектов
public interface GameEntity {
    String getName();
    Position getPosition();
    void update();
    void render();
}

// Интерфейс для объектов, которые могут двигаться
public interface Movable extends GameEntity {
    void moveTo(Position newPosition);
    boolean canMoveTo(Position position);
    int getMovementRange();
}

// Интерфейс для объектов, которые могут сражаться
public interface Combatable extends GameEntity {
    void attack(GameEntity target);
    void takeDamage(int damage);
    boolean isAlive();
    int getAttackRange();
}
```

Реализация игровых интерфейсов

```
public class Warrior implements Movable, Combatable {
    private String name;
    private Position position;
    private int health;
    private int attackPower;

    @Override
    public String getName() { return name; }

    @Override
    public Position getPosition() { return position; }

    @Override
    public void update() {
        // Логика обновления воина
    }

    @Override
    public void render() {
        // Отрисовка воина
    }

    @Override
    public void moveTo(Position newPosition) {
        if (canMoveTo(newPosition)) {
            this.position = newPosition;
        }
    }

    @Override
    public void attack(GameEntity target) {
        if (target instanceof Combatable) {
            Combatable enemy = (Combatable) target;
            enemy.takeDamage(attackPower);
        }
    }
}
```

Использование интерфейсов в коллекциях

```
public class GameEngine {
    private List<GameEntity> entities = new ArrayList<>();
    private List<Movable> movableEntities = new ArrayList<>();
    private List<Combatable> combatEntities = new ArrayList<>();

    public void addEntity(GameEntity entity) {
        entities.add(entity);

        if (entity instanceof Movable) {
            movableEntities.add((Movable) entity);
        }

        if (entity instanceof Combatable) {
            combatEntities.add((Combatable) entity);
        }
    }

    public void updateAll() {
        entities.forEach(GameEntity::update);
    }

    public void moveAll() {
        movableEntities.forEach(movable -> {
            // Логика движения
        });
    }
}
```

Преимущества использования интерфейсов

Гибкость:

✅ **Множественная реализация**
✅ **Слабая связанность**
✅ **Легкое тестирование**

Расширяемость:

✅ **Новые реализации** без изменения кода
✅ **Полиморфизм**
✅ **Инверсия зависимостей**

Поддержка:

Домашнее задание

Задача 1:

Создать интерфейс `ResourceProducer` с методами:

- `produce()` — производство ресурса
- `getProductionRate()` — скорость производства

Задача 2:

Реализовать классы `Farm` и `Mine`, реализующие `ResourceProducer`

Задача 3:

Создать интерфейс `Upgradeable` с методами для улучшения зданий

Что дальше?

На следующей лекции:

- **Коллекции и generics**
- **ArrayList, LinkedList**
- **HashMap, HashSet**
- **Stream API**

Подготовка:

- Изучить главу 7-8 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

Вопросы?

Контакты:

- **Email:** [ваш.email@university.edu]
- **Telegram:** [@username]
- **Офис:** [номер кабинета]

Следующая лекция: Коллекции и generics