Plan

2023-10-20

OOP Review

1. General Concepts

- General idea
- Object
- Class
- Field/Attribute
- Class methods
- Constructor
  - Default constructors
  - Parameterized constructors
- super
- this
- getters
- setters
- Special features of a class with a single private constructor

1. Encapsulation

- General idea
- Access modifiers (private, protected, public)

1. Polymorphism

- General idea
- Constructor overloading
- Static (method and operator overloading)
- Dynamic (method overriding)
  - toString
  - equals vs ==

1. Inheritance

- General idea
- Inheritance restrictions (use of final methods)
- instanceof operator
- Relation of any class to the Object class
- final method
- Extending modifiers in inheritance, overriding, and method hiding

1. Abstraction

- General idea
- Abstract classes
- Abstract methods
- Interfaces
- Multiple inheritance of interfaces

Повторение ООП

1. Общие понятия
    - общее понятие
    - объект
    - класс
    - поле/атрибут
    - методы класса
    - конструктор
    - конструкторы по умолчанию
    - конструктор с параметрами
    - super
    - this
    - getters
    - setters
    - особенноси класса с единственным закрытым (private) конструктором
2. Инкапсуляция
    - общее понятие
    - модификаторы доступа (private, protected, public)
3. Полиморфизм
    - общее понятие
    - перегрузка конструктора
    - Статический (перегрузка методов и операторов)
    - Динамический (переопределение методов)
        - toString
        - equals против ==
4. Наследование
    - общее понятие
    - ограничения наследования (использование final методов)
    - оператор instanceof
    - связь любого класса с классом Object
    - метод final
    - Расширение модификаторов при наследовании, переопределении и сокрытии методов
5. Абстракция
    - общее понятие

- абстрактные классы
- абстрактные методы
- интерфейсы
- Множественное наследование интерфейсов

Theory

▶ **English**

▼ **На русском**

1. Общие понятия

Общее понятие

ООП (Объектно-Ориентированное Программирование) — это парадигма программирования, основанная на концепциях объектов и классов.

Объект

Объект — это экземпляр класса.

```
Car myCar=new Car();
```

Класс

Класс — это шаблон для создания объектов. Он описывает состояние и поведение, которое будут иметь объекты.

```
public class Car {
    String model;
    int speed;

    void go() {
        System.out.println("This car is going");
    }
}
```

Поле/Атрибут

Поля класса хранят состояние объекта.

```
public class Car {
```

```
    public int speed; // поле класса

    // other code...
}
```

Методы класса

Методы определяют поведение класса.

```
public class Car {
    void go() { // поведение класса, метод класса
        System.out.println("This car is going");
    }
}
```

Конструктор

Конструктор — это специальный метод, вызывается при создании объекта.

Конструкторы по умолчанию

Java автоматически создаёт конструктор без параметров, если в классе не определены другие конструкторы.

```
public class Car {
    public Car() {
    }
}
```

Конструктор с параметрами

```
public class Car {

    public Car(String model) { //Конструктор с параметром
        this.model = model;
    }
}
```

`super` и `this`

`super` используется для вызова конструктора родительского класса, `this` — для доступа к полям и методам текущего класса.

```
public class Car extends Transport {
```

```
    private String model; // поле класса

    public Car(String model) {
        super(); // вызов конструктора родительского класса
        this.model = model; // обращение к полю текущего класса
    }
}
```

Getters и Setters

Методы для доступа к приватным полям.

```
public class Car extends Transport {

    private String model;

    public Car(String model) {
        super();
        this.model = model;
    }

    public String getModel() { // getter для поля класса model
        return this.model; // обращение к полю текущего класса
    }

    public void setModel(String model) { // setter для поля класса model
        this.model = model; // обращение к полю текущего класса
    }
}
```

Особенности класса с единственным закрытым (private) конструктором

Закрытый конструктор не позволяет создать экземпляр класса извне.

```
public class Singleton {

    private Singleton() {
    }
}
```

2. Инкапсуляция

Общее понятие

Инкапсуляция — это одна из ключевых концепций ООП, позволяющая скрыть детали реализации от внешнего мира.

Модификаторы доступа (private, protected, public)

Инкапсуляция реализуется с помощью модификаторов доступа.

private: поля и методы доступны только внутри класса. **protected**: поля и методы доступны внутри класса и его наследников. **public**: поля и методы доступны из любого места.

```
public class Car {
    private String model; // доступно только внутри этого класса
    protected int speed; // доступно в этом классе и его наследниках

    public void go() {
    } // доступно из любого места
}
```

3. Полиморфизм

Общее понятие

Полиморфизм — это способность объектов разных типов обрабатываться как объекты одного типа.

Перегрузка конструктора

Это возможность создавать несколько конструкторов в одном классе с разным набором параметров.

```
public class Car {
    private String model;

    public Car() {
    }

    public Car(String model) {
        this.model = model;
    }
}
```

Статический полиморфизм (перегрузка методов и операторов)

Статический: Статический полиморфизм достигается через перегрузку методов.

```
        public void go(){}
public void go(String place){}
public void go(String formPlace,String toPlace){}
```

Динамический полиморфизм (переопределение методов)

Динамический: Динамический полиморфизм достигается через переопределение методов.

```
public class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

public class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Использование:

```
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.sound();  // Output: Dog barks
    }
}
```

`equals`

- Метод equals используется для сравнения двух объектов на равенство.
- По умолчанию, этот метод наследуется от класса Object и сравнивает объекты по ссылке. То есть, два объекта считаются равными только в том случае, если они указывают на одно и то же место в памяти, а не по содержимому

Переопределение метода `equals` Часто требуется более тонкое сравнение объектов. В этом случае метод equals переопределяется. При переопределении этого метода, важно соблюдать контракт, который включает в себя:

- если a.equals(b) возвращает true, то и b.equals(a) должен возвращать true.
- если a.equals(b) и b.equals(c), то a.equals(c) также должен возвращать true.

```
@Override
public boolean equals(Object obj) {
        if (this == obj) {
        return true;
        }

        if (obj == null || getClass() != obj.getClass()) {
        return false;
        }

        MyObject myObject = (MyObject) obj;

        return field1 == myObject.field1 && field2.equals(myObject.field2);
        }
```

`toString`

Метод `toString` используется для получения строкового представления объекта. По умолчанию, этот метод возвращает строку в формате "имя*класса*@*хеш*кода". В большинстве случаев, такое представление не является информативным, и метод toString часто переопределяется для возврата более подробной информации о состоянии объекта.

```
@Override
public String toString() {
    return "MyObject{" +
        "field1=" + field1 +
        ", field2='" + field2 + '\'' +
        '}';
}
```

В этом примере, метод toString переопределен для класса MyObject так, чтобы он возвращал значения полей field1 и field2.

Оба метода equals и toString являются часто переопределяемыми и очень полезными при разработке Java-приложений для обеспечения корректного сравнения объектов и удобного их отображения.

4. Наследование

Общее понятие

Наследование — это механизм, который позволяет одному классу наследовать поля и методы другого класса.

Ограничения наследования (использование `final` методов)

Если метод объявлен как **final**, он не может быть переопределен.

```
public final void stop() { // этот метод не может быть переопределен
   // code...
}
```

Оператор `instanceof`

Оператор instanceof в Java используется для проверки, является ли объект экземпляром конкретного класса или его подкласса, либо имплементирует ли он определенный интерфейс. Это может быть полезно, когда у вас есть ссылка на объект, тип которого является суперклассом или интерфейсом для разных классов, и вам нужно выполнить разные действия в зависимости от его фактического типа.

```
if(myCar instanceof Car){
    // some code
    }
```

Особенности и ограничения

- null не является экземпляром никакого класса. Поэтому null instanceof ClassName всегда возвращает false.
- instanceof не может быть использован для проверки примитивных типов данных (int, float, char и т.д.).

Связь любого класса с классом `Object`

В Java все классы неявно наследуются от класса `Object`.

Связь любого класса с классом `Object`

В Java, класс `Object` является корневым классом в иерархии всех классов. Это значит, что любой класс в Java является подклассом класса `Object`, явно или неявно. Если класс не наследует от какого-либо другого класса, он автоматически наследует от класса `Object`.

Примеры

Явное наследование

```
class MyCustomClass extends Object {
    // Класс явно наследует от Object
}
```

Неявное наследование

```
class MyCustomClass {
    // Класс неявно наследует от Object
}
```

В обоих случаях `MyCustomClass` будет иметь доступ к методам, определенным в классе `Object`.

Основные методы класса `Object`

1. **`public String toString()`**: Возвращает строковое представление объекта. По умолчанию, это имя класса и хеш-код объекта.

   ```
   Object obj = new Object();
   System.out.println(obj.toString());  // Вывод: java.lang.Object@<hashcode>
   ```

2. **`public boolean equals(Object obj)`**: Сравнивает объекты на равенство. По умолчанию, метод сравнивает ссылки, а не содержимое.

   ```
   Object obj1 = new Object();
   Object obj2 = new Object();
   System.out.println(obj1.equals(obj2));  // Вывод: false
   ```

3. **`public final Class<?> getClass()`**: Возвращает объект `Class`, который представляет класс данного объекта.

   ```
   Object obj = new Object();
   System.out.println(obj.getClass());  // Вывод: class java.lang.Object
   ```

4. **`public int hashCode()`**: Возвращает хеш-код объекта. По умолчанию, хеш-код зависит от физического адреса в памяти.

Эти методы можно переопределить в своих классах для создания специфической функциональности.

Например, методы `equals()` и `hashCode()` часто переопределяются для обеспечения корректного сравнения объектов.

Метод `final`

Метод, объявленный как `final`, не может быть переопределен в подклассах.

Расширение модификаторов при наследовании, переопределении и сокрытии методов

При переопределении методов модификатор доступа может быть расширен, но не может быть сужен.

5. Абстракция

Общее понятие

Абстракция означает выделение ключевых характеристик объекта, игнорируя незначимые детали.

В контексте ООП, это означает создание класса, который содержит общую структуру и функциональность для группы подклассов.

Пример Представьте, что у вас есть классы Car, Boat и Airplane. Все эти транспортные средства могут перемещаться, поэтому вы можете создать абстрактный класс Vehicle с методом move.

Абстрактные классы

Абстрактный класс — это класс, который не может быть инстанциирован (не может быть создан объект этого класса). Он служит базой для других классов. В абстрактном классе можно определить абстрактные методы без реализации, а также обычные методы с реализацией.

```
public abstract class Vehicle {
    // Абстрактный метод
    abstract void move();

    // Обычный метод
    public void stop() {
        System.out.println("The vehicle has stopped.");
    }
}
```

Абстрактные методы

Абстрактные методы — это методы, которые объявлены, но не реализованы в абстрактном классе. Подклассы обязаны предоставить реализацию для всех абстрактных методов базового класса, если они не являются абстрактными.

```
public class Car extends Vehicle {
    @Override
    public void move() {
        System.out.println("The car is moving.");
    }
}
```

В этом примере класс Car предоставляет реализацию для абстрактного метода move базового класса Vehicle.

Теперь можно создать объект класса Car и вызвать его методы:

```
        Vehicle myCar=new Car();
        myCar.move();  // Output: "The car is moving."
```

```
        myCar.stop();  // Output: "The vehicle has stopped."
```

Важно отметить, что создать объект абстрактного класса Vehicle напрямую **невозможно**:

```
        Vehicle vehicle=new Vehicle();  // Ошибка компиляции
```

Абстрактные классы и методы — это мощный инструмент в Java для реализации абстракции и организации кода. Они позволяют создавать общие шаблоны, которые можно переиспользовать в различных ситуациях

Интерфейсы

Интерфейс - это схема, определяющая контракт для классов, которые его реализуют. Интерфейс не может содержать логику реализации, он содержит только объявления методов, которые должны быть переопределены в классе, реализующем данный интерфейс. В этом смысле интерфейс действует как набор правил и протоколов, которым должен следовать класс.

```
public interface Movable {
    void move();
}
```

Множественное наследование интерфейсов

Интерфейсы могут расширять другие интерфейсы с помощью ключевого слова extends

```
public class Car implements Movable, Stoppable {
    // Implementation here
}
```

Еще пример:

```
public interface Drawable {
    void draw();
}

public interface Rotatable extends Drawable {
    void rotate();
}

public class Circle implements Rotatable {
    public void draw() {
        System.out.println("Drawing a circle");
```

```
    }

    public void rotate() {
        System.out.println("Rotating the circle");
    }
}
```

В этом примере, интерфейс Rotatable расширяет интерфейс Drawable, добавляя метод rotate(). Класс Circle реализует Rotatable, и поэтому он должен реализовать все методы из Drawable и Rotatable.

Таким образом, ключевое слово extends позволяет формировать иерархии интерфейсов,

6. Static, non-static методы

Static методы

Методы, которые принадлежат самому классу, а не его экземплярам.

```
public static void info(){
        System.out.println("This is a car class");
        }
```

Non-static методы

Методы, которые принадлежат экземплярам класса.

```
public void go(){
        System.out.println("This car is going");
        }
```

Вопросы по ООП

ссылка на статью

Homework

▶ **English**

▼ **На русском**

Задача 1.

По методам - должны быть CRUD-операции: C - create, add R - read, find U - update (не обязательно) D - delete, remove

- Book - Library, найти несколько книг по автору
- Computer (Laptop, SmartPhone) - Shop, найти компьютеры со скидкой на BlackFriday (отбор цене)
- Product (Food, MeatFood, MilkFood) - Supermarket, искать продукты по сроку годности (алфавитный порядок)
- Pets (Cat, Dog) - Hotel, стоимость пребывания, выручка от отеля, найти всех по породе
- Student, Aspirant, Professor - High school

Во всех реализуемых классах должен быть некий id (штрих-код, isbn, id и т.д.)

Главное - не функциональность, а последовательность от классов через интерфейс к тестам и имплементации.

Задача 2.(*) Заполните массив целых чисел числами по порядку от 1 до 100. Задумайте случайное число в интервале от 1 до 100 и добавьте его в массив на произвольную (случайную) позицию. Найдите добавленный в массив дубликат наиболее простым способом.

Code

code/Lesson_33/src/example2/Audi.java

```
package example2;

/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Audi extends Car {

    public void makeSound() {
        System.out.println("Audi make sound");
    }
}
```

code/Lesson_33/src/example2/Bwm.java

```
package example2;

/**
 * @author Andrej Reutow
```

```
 * created on 20.10.2023
 */
public class Bwm extends Car {

    public void repair() {
        System.out.println("Bwm reapir");
    }
}
```

code/Lesson_33/src/example2/Car.java

```
package example2;

/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Car extends Transport {


    @Override
    public void move() {
        System.out.println("My car is moving");
    }

    public void drive() {
        System.out.println("My car is driving");
        super.move();
        this.move();
        move();
    }
}
```

code/Lesson_33/src/example2/CarAppRunner2.java

```
package example2;

/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class CarAppRunner2 {
    public static void main(String[] args) {
        Car carCar = new Car();
```

```
            carCar.drive();
        }
    }
```

code/Lesson_33/src/example2/Plane.java

```
    package example2;

    /**
     * @author Andrej Reutow
     * created on 20.10.2023
     */
    public class Plane extends Transport {

        @Override
        public void move() {
            System.out.println("Plane is moving");
        }

        public void fly() {
            System.out.println("Plane is flying");
            move();
        }
    }
```

code/Lesson_33/src/example2/Transport.java

```
    package example2;

    /**
     * @author Andrej Reutow
     * created on 20.10.2023
     */
    public class Transport {

        public void move() {
            System.out.println("Transport is moving");
        }
    }
```

code/Lesson_33/src/exsample3/Example3Runner.java

```
    package exsample3;
```

```
import exsample3.entity.Animal;
import exsample3.entity.Dog;

/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Example3Runner {

    public static void main(String[] args) {
        // создание объекта не возможно, т.к. конструктор приватный
        //  SomeUtilityClass utilityClass = new SomeUtilityClass();

        Animal animal = new Dog("Black", "Tom", "Такса");
    }
}
```

code/Lesson_33/src/exsample3/SomeUtilityClass.java

```
package exsample3;

/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class SomeUtilityClass {

    /**
     * Приватный конструктор, так как это класс утилитов.
     */
    private SomeUtilityClass() {
    }

    public static void someMth1() {
        // code ...
    }

    public void someNonStaticMth1() {
        // code ...
    }
}
```

code/Lesson_33/src/exsample4/Example4Runner.java

```java
package exsample4;

/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Example4Runner {

    public static void main(String[] args) {
        User userVasja = new User("Vasja", "Pupkin");
        User userVasja2 = new User("Vasja", "Pupkin-2");
        User unserMarina = new User("Marina", "Pupkin");

        System.out.println(userVasja == userVasja); // true
        System.out.println(userVasja == userVasja2); // fasle

        System.out.println(userVasja.equals(userVasja2)); // false
    }
}
```

code/Lesson_33/src/exsample4/User.java

```java
package exsample4;

/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class User {

    private String name;
    private String lastName;

    public User(String name, String lastName) {
        this.name = name;
        this.lastName = lastName;
    }

//    @Override
//    public boolean equals(Object obj) {
//        if (obj != null && obj instanceof User) {
//            User otherUser = (User) obj;
//            return name.equals(otherUser.name);
```

```
//        }
//
//        return false;
//    }


    @Override
    public boolean equals(Object object) {
        if (this == object) return true;
        if (object == null || getClass() != object.getClass()) return false

        User user = (User) object;

        return this.name.equals(user.name) &&
                this.lastName.equals(user.lastName);
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }
}
```

code/Lesson_33/src/Car.java

```
/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Car extends Transport {

    @Override
    public void move() {
        System.out.println("My car is moving");
    }

    public void drive() {
        System.out.println("My car is driving");
        move();
    }
}
```

code/Lesson_33/src/CarAppRunner.java

```java
/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class CarAppRunner {
    public static void main(String[] args) {
        Car carCar = new Car();
        carCar.drive();

        Transport transportCar = new Car();
        Transport transportPlane = new Plane();
        transportCar.move();
        transportPlane.move();

        Transport[] transports = {transportCar, transportPlane};

        for (int i = 0; i < transports.length; i++) {
            Transport currentIterableTransport = transports[i]; // итерируе

            if (currentIterableTransport instanceof Plane) {
                Plane plane = (Plane) currentIterableTransport;
                plane.fly();
            }

            if (currentIterableTransport instanceof Car) {
                ((Car) currentIterableTransport).drive();
            }
        }

//        Car car = (Car) transportPlane; // Transport  = Plane
//        car.drive();
//        Plane plane = (Plane) carCar; // ошибка на стадии компиляции


    }
}
```

code/Lesson_33/src/CarAppRunner2.java

```java
/**
 * @author Andrej Reutow
```

```
 * created on 20.10.2023
 */
public class CarAppRunner2 {
    public static void main(String[] args) {
        Transport transportTruck = new Truck();
        transportTruck.move(); // Transport is moving

        Transport transportCar = new Car();
        transportCar.move(); // My car is moving


        // Статический полиморфизм (перегрузка методов и операторов)
        go();
        go("my param 1");
        go(1);
        go("my str param", 2);

        // проверка статических переменных
        Transport plane1 = new Plane();
        Transport plane2 = new Plane();
        Transport plane3 = new Plane();

        plane1.move(); // Non static var: 1, Static var: 1
        plane3.move(); // Non static var: 1, Static var: 2
        plane2.move(); // Non static var: 1, Static var: 3

        Plane.idCounter++; // 4 + 1

        plane1.move(); // Non static var: 2, Static var: 5

        Plane.idCounter += 100; // 5 + 100
        plane2.move(); // Non static var: 2, Static var: 106
        plane2.move(); // Non static var: 3, Static var: 107

    }

    public static void go() {
        System.out.println("Go");
    }

    public static void go(String param1) {
        System.out.println("Go, param1 " + param1);
    }
```

```
    public static void go(Integer param1) {
        System.out.println("Go, param1 " + param1);
    }

    public static void go(String param1, Integer param2) {
        System.out.println("Go, param1: " + param1 + " param2: " + param2);
    }
}
```

code/Lesson_33/src/Plane.java

```
/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Plane extends Transport {

    public static int idCounter = 0;
    public int idCounterNonStatic = 0;

    @Override
    public void move() {
        idCounter++;
        idCounterNonStatic++;
        System.out.println("Non static var: " + idCounterNonStatic);
        System.out.println("Static var: " + idCounter);
        System.out.println("Plane is moving");
    }

    public void fly() {
        System.out.println("Plane is flying");
        move();
    }
}
```

code/Lesson_33/src/Transport.java

```
/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Transport {
```

```
    public void move() {
        System.out.println("Transport is moving");
    }
}
```

code/Lesson_33/src/Truck.java

```
/**
 * @author Andrej Reutow
 * created on 20.10.2023
 */
public class Truck extends Transport {

    public void someTruckMth() {
        System.out.println("This method exists just by Truck class");
    }
}
```