Основы программирования на языке Java. Уровень 1. Занятие 7

Степулёнок Денис Олегович — denis.stepulenok@oracle.com
Солодкая Анастасия Сергеевна — a.s.solodkaya@gmail.com

28 октября 2016 года

Абстрактные классы и методы. Интерфейсы. Анонимные классы

- Распределение обязанностей между классами
- Интерфейсы, как альтернатива множественному наследованию
- Маркер-интерфейсы, функциональные интерфейсы
- Интерфейс Comparable и правильная сортировка объектов

Контрольные вопросы для повторения I

Что такое ООП?

ООП — методология программирования, в которой программа = совокупность объектов, каждый из которых является экземпляром конкретного класса. ООП использует в качестве базовых элементов взаимодействие объектов.

Что такое объект?

Объект — именнованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение, обладающий именем набор данных (полей и свойств объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним. Объект — конкретный экземпляр класса.

Основые принципы ООП

4 кита «ООП»:

- Абстракция
- Инкапсуляция
- Наследование

Контрольные вопросы для повторения II

- Полиморфизм

Наследование — процесс благодаря которому один объект может приобрести свойства другого объекта (наследование всех свойств одного объекта другим) и добавлять черты характерны только для него самого!
class Dog extends Animal ...
Суперкласс -> Подкласс Родительский -> Дочерний
Абстракция — выделение общих характеристик объекта, исключая набор незначительных (зависит от конкретной задачи).
С помощью принципа **абстракции** данных, данные преобразуются в объекты. Данные обрабатываются в виде цепочки сообщений между отдельными объектами. Все объекты проявляют свои уникальные признаки поведения. Огромный плюс абстракции в том, что она отделяет реализацию объектов от их деталей, что в свою очередь позволяет управлять функциями высокого уровня через функции низкого уровня.
Инкапсуляция (encapsulation) — сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса), это свойство

Контрольные вопросы для повторения III

которое позволяет закрыть доступ к полям и методам класса другим классам, а предоставлять им доступ только через интерфейс (метод). При использовании ОбОр подхода не принято применять прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято задействовать специальные методы этого класса для получения и изменения его свойств (геттеры и сеттеры).
Полиморфизм (polymorphism) (от греческого polymorphos) — свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для иерархии классов действий. Выполнение каждого конкретного действия будет определяться конкретным классом. В более общем смысле, концепцией полиморфизма является идея «один интерфейс, множество методов». Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Данные обрабатываются в виде цепочки сообщений между отдельными объектами. Все объекты проявляют свои уникальные признаки поведения.

Контрольные вопросы для повторения IV

Огромный плюс абстракции в том, что она отделяет реализацию объектов от их деталей, что в свою очередь позволяет управлять функциями высокого уровня через функции низкого уровня.

Контрольные вопросы для повторения I

В чём преимущества ОО языков программирования?

Они представляют реальные объекты в жизни, например, Машина, Джип, Счет в банке и тд... Инкапсуляция, наследование и полиморфизм делает его еще мощнее.

Основные преимущества:

- повторное использование кода(наследование)
- реальное отображение предметной области. Объекты соответствуют реальному миру

Гибкость против хрупкости I

Жизнь не стоит не месте.
Не могут стоять на месте и программы, которые мы пишем. Чтобы не отставать от сегодняшнего, близкого к кошмару, темпа изменений, нам необходимо приложить все усилия для написания программ слабосвязанных и гибких, насколько это возможно. В противном случае мы придем к тому, что наша программа быстро устареет или станет слишком хрупкой, что не позволит устранять ошибки, и может в конечном итоге оказаться в хвосте сумасшедшей гонки в будущее.
Шпионы, диссиденты, революционеры и им подобные часто организованы в небольшие группы, называемые ячейками. Хотя отдельные личности в каждой ячейки могут знать друг о друге, они не знают ничего об участниках других ячеек. Если одна ячейка раскрыта, то никакое количество "сыворотки правды"неспособно выбить из ее участников информацию об их сподвижниках вне пределов ячейки. Устранение взаимодействий между ячейками убережет всех.

Гибкость против хрупкости II

Этот принцип хорошо надо применить и к написанию программ. Разбейте вашу программу на ячейки (модули) и ограничьте взаимодействие между ними. Если один модуль находится под угрозой и должен быть заменен, то другие модули должны быть способны продолжить работу.

Сведение связанности к минимуму I

Непосредственное пересечение отношений между объектами может быстро привести к комбинаторному взрыву отношений зависимости.
Признаки этого явления:

- Простые изменения в одном модуле, которые распространяются в системе через модули, не имеющие связей
- Разработчики, которые боятся изменить программу, поскольку они не уверены, как и на чем скажется это изменение

Системы, в которых имеется большое число ненужных зависимостей, отличаются большой сложностью (и высокими затратами) при сопровождении и в большинстве случае весьма нестабильны.
Для того чтобы поддерживать число зависимостей на минимальном уровне, надо пользоваться **законом Деметра** при проектировании наших методов и классов.
Закон Деметры (Law of Demeter) — набор правил проектирования при разработке программного обеспечения, в частности объектно-ориентированных программ, накладывающий ограничения на

Сведение связанности к минимуму II

взаимодействия объектов (модулей). Обобщенно, закон Деметры является специальным случаем слабого зацепления (loose coupling). Правила были предложены в конце 1987 в северо-восточном Университете (Бостон, Массачусетс, США).
Каждый класс и модуль:

- должен обладать ограниченным знанием о других классах (модулях): знать о модулях, которые имеют «непосредственное» отношение к этому модулю
- должен взаимодействовать только с известными ему модулями «друзьями», не взаимодействовать с незнакомцами
- обращаться только к непосредственным «друзьям»

Название взято из проекта «Деметра», который использовал идеи аспектно-ориентированного и адаптивного программирования. Проект был назван в честь Деметры, греческой богини земледелия, чтобы подчеркнуть достоинства философии программирования «снизу-вверх».

Закон Деметра для функций I

Минимизируйте связывание между модулями

```
public class Demeter {
    A a;

    int func() {
        return 5;
    }

    // Закон Деметера для функций:
    public void example(B b) { // Любой метод объекта
        // b - аргумент метода
        C c = new C(); // должен обращаться только к методам принадлежащим:
        int f = func(); // Самому объекту
        // Аргументы метода
        System.out.println(b.toString()); // Любым параметрам, переданным в метод
        b.methodB(); // OK
        a = new A(); // Любым создаваемым им объектам
        a.notify(); // любым непосредственно содержащимся объектам компонентов
        // Примеры нарушения:
        a.b.methodB(); // Обращение к методу вложенного объекта
    }

    public static void main(String[] args) {
        Demeter demeter = new Demeter();
```

Распределение обязанностей между классами

Закон Деметра для функций II

```
        demeter.example(new B());
    }
}
```

Интерфейсы и множественное наследование I

Множественное наследование классов не поддерживается в Java.
При этом класс может реализовывать множество интерфейсов.

```
public class Interfaces {
    interface I1 {
        void method1();
    }

    interface I2 {
        void method2();
    }

    class X implements I1, I2 {

        @Override
        public void method1() {
            System.out.println("X.method1");
        }

        @Override
        public void method2() {
            System.out.println("X.method2");
        }
    }
```

Интерфейсы и множественное наследование II

```
}
```

Маркер-интерфейсы, функциональные интерфейсы I

Интерфейс-маркер (marker interface pattern) — шаблон проектирования, применяемый в ЯП с проверкой типов во время выполнения. Шаблон предоставляет возможность связать метаданные (интерфейс) с классом даже при отсутствии в языке явной поддержки для метаданных.
Класс реализует интерфейс (помечается интерфейсом), а взаимодействующие с классом методы проверяют наличие интерфейса.
Пример «Serializable» — экземпляры могут быть записаны в ObjectOutputStream.

Маркер-интерфейсы, функциональные интерфейсы II

```
import java.io.*;

public class SerializableDemo implements Serializable {
    private String name;
    private int priority;

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        SerializableDemo demo = new SerializableDemo();
        demo.name = "Test name";
        demo.priority = 4;
        // Сохраняем в файл
        try (ObjectOutputStream s = new ObjectOutputStream(new FileOutputStream("s.dat"))) {
            s.writeObject(demo);
        }
        // Загружаем из файла
        try (ObjectInputStream s = new ObjectInputStream(new FileInputStream("s.dat"))) {
            SerializableDemo demo2 = (SerializableDemo) s.readObject();
            System.out.println("demo2.name = " + demo2.name);
            System.out.println("demo2.priority = " + demo2.priority);
        }
    }
}
```

Comparable и правильная сортировка объектов I

Когда мы сортируем числа или строки — понятно как их сортировать. Чтобы правильно реализовывать сортировку объектов нужно реализовать интерфейс Comparable:

```
import java.util.Arrays;

public class ComparableDemo {
    public static void main(String[] args) {
        Task[] tasks = {new Task(1, "First"), new Task(3, "Second"), new Task(5, "Third")};
        Arrays.sort(tasks); // Сортируем
        int i = 0;
        for (Task t : tasks) {
            System.out.println(++i + ". " + t);
        }
    }

    // Если не укажем, получим java.lang.ClassCastException: ИмяКласса cannot be cast to
        java.lang.Comparable
    private static class Task implements Comparable<Task> {
        private final int priority;
        private final String name;

        Task(int priority, String name) {
            this.priority = priority;
```

Comparable и правильная сортировка объектов II

```
        this.name = name;
    }

    @Override
    public String toString() {
        return "P" + priority + " " + name;
    }

    @Override
    public int compareTo(Task task) {
        if (priority < task.priority)
            return -1; // Результат < 0: this < task
        if (priority > task.priority)
            return 1; // Результат > 0: this > task
        if (priority != task.priority) // Результат = 0: объекты равны
            throw new AssertionError("Priorities must be equal :)");
        return name.compareTo(task.name); // Если приоритеты равны, сортируем по алфавиту
    }
}
}
```