**Мобильная разработка.**

**Занятие №8. Основы объектно-ориентированного программирования**

**Класс и объект — в чем разница?**

Мир, в котором мы живем, состоит из объектов. Если мы посмотрим вокруг, то увидим, что нас окружают дома, деревья, автомобили, мебель, посуда, компьютеры. Все эти предметы являются объектами, и каждый из них обладает набором определенных характеристик, поведением и назначением. Мы привыкли к объектам, и мы их используем всегда для вполне конкретных целей. Например, если нам необходимо доехать до работы, мы пользуемся автомобилем, если захотим поесть – посудой, а если отдохнуть – нам понадобится удобный диван. Человек привык мыслить объектно для решения задач в повседневной жизни. Это послужило одной из причин использования объектов в программировании, а такой подход к созданию программ назвали объектно-ориентированным.

Представьте, что вы разработали новую модель телефона и хотите наладить её серийное производство. Как разработчик телефона, вы знаете для чего он нужен, как он будет функционировать, и из каких деталей он будет состоять (корпус, микрофон, динамик, провода, кнопки и т.д.). При этом только вы знаете, как соединить эти детали. Однако вы не планируете выпускать телефоны лично, для этого у вас есть целый штат работников. Чтобы вам не пришлось каждый раз объяснять, как соединить детали телефона, и чтобы все телефоны при производстве получались одинаковыми, прежде чем начать их выпуск, вам понадобиться сделать чертеж в виде описания устройства телефона. В ООП такое описание, чертеж, схема или шаблон называется классом, из которого при выполнении программы создается объект.

Класс — это описание еще не созданного объекта, как бы общий шаблон, состоящий из полей, методов и конструктора, а объект – экземпляр класса, созданный на основе этого описания.

С точки зрения ООП каждый класс представляет собой совокупность:

- полей класса — переменных для хранения данных, описывающих класс. Это те свойства, параметры, характеристики, которые описывают состояние сущности;

- методов класса — функций для работы с полями класса. Это те действия, которые можно производить с этой сущностью.

Работая ранее, мы пользовались Scanner для считывания символов, которые вводятся пользователем:

*Scanner scan = new Scanner(System.in);*

Здесь Scanner - это класс стандартной библиотеки, а scan – это объект. То есть класс – это есть реализация в коде программы. А объект – это конкретный экземпляр написанного класса.

Класс реализован единожды в коде программы, а объектов мы можем создавать сколько хотим.

При этом нам не важно, какая работа проходит внутри - главное мы используем методы этого класс. Все эти методы описаны разработчиками этих классов в мануалах к ним.

*Scanner sc = new Scanner(System.in);*

*Scanner scan = new Scanner(System.in);*

Это разные объекты одного класса.

**Основы ООП**

В основе ООП лежат четыре принципа:

- абстракция - выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.

- инкапсуляция — объединение данных и процедур для работы с этими данными в единое целое;

- наследование — средство получения новых классов из существующих;

- полиморфизм — создание общего интерфейса для группы близких по смыслу действий.

В идеологии ООП разработчик начинает написание программы с проектирования классов. В процессе создания программы — это важнейшая и действительно сложная задача, для решения которой предлагается множество подходов и методик, а также разработаны специальные инструменты для графической иллюстрации структуры классов и их связей между собой (например, диаграммы классов языка графического описания UML).

Попробуем на примере игры разобраться что же такое объекты, классы и принципы ООП.

**Абстракция**

Допустим в игре у нас есть два типа персонажей – войны и маги. Вне зависимости от того, кем является персонаж, войном или магом, у него есть свойства – имя, здоровье и метод ходить. Поэтому мы можем реализовать класс Юнита, у которого будут эти свойства и методы.

### Только что мы воспользовались принципом ООП – Абстракция.

Абстракция означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.

Научимся создавать собственные классы в языке Java. Прежде всего, нужно написать ключевое слово class. После этого нужно написать имя нового класса. Это имя должно быть создано по общим правилам имен идентификаторов (в том числе переменных) в Java.

Описание класса содержит:

- описание данных, которые хранятся в классе (поля/переменные);

- описание методов, которые обрабатывают эти данные.

Общая форма оформления класса:

//ПСЕВДОКОД class ИмяКласса {

типПоля1 имяПоля1;

типПоля2 имяПоля2;

типРезультата1 имяМетода1 (параметрыМетода1) { //телоМетода1 }

типРезультата2 имяМетода2 (параметрыМетода2) { //телоМетода2 }

}

Для соблюдения конвенции по оформлению кода принято использовать стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово пишется с заглавной буквы (CamelCase). В Java используют UpperCamelCase для названий классов и lowerCamelCase — для названий полей и методов.

Заметим, что создание нового класса означает создание нового типа данных.

Теперь можно создавать переменные (объекты) этого типа:

//ПСЕВДОКОД

ИмяКласса имяОбъекта;

Заметим, что в этой строчке кода мы только объявили переменную, то есть сообщили компилятору, что в переменной myUnit1 мы собираемся хранить ссылку на объект — экземпляр класса Unit. Самого объекта в памяти еще нет.

Мы даже можем явно указать, что объектная переменная пока ни на что не ссылается, используя значение null:

Чтобы создать сам объект, необходимо использовать операцию new (мы уже встречались с этой операцией, когда создавали массивы):

Unit myUnit1 = new Unit();

При выполнении оператора new создается новый объект указанного класса, и полученная в результате ссылка на область памяти, в которой был размещен созданный объект, сохраняется в переменную myUnit1. При этом автоматически вызывается специальный метод класса, называемый конструктор, который мы рассмотрим подробно в следующей теме модуля.

Для обращения к полям (и методам) созданного объекта нужно использовать операцию «точка».

**Ключевое слово this**

Иногда будет требоваться, чтобы метод ссылался на вызвавший его объект. Чтобы это было возможно, в Java определено ключевое слово this. Оно может использоваться внутри любого метода для ссылки на текущий объект. То есть this всегда служит ссылкой на объект, для которого был вызван метод.

**Наследование**

Теперь мы хотим все-таки реализовать классы Война и Мага. У Война мы бы добавили свойство Оружие и метод Драться, а у Мага свойство Мана и метод Колдовать.

Но и у Война и у Мага также остаются свойства имя, здоровье и метод ходить и чтобы их не описывать два раза, мы воспользуемся следующим принципом ООП- наследованием. В данном случае классы Воин и Маг будут являться классами-наследниками от класса-родителя Юнит. И нам не нужно в каждом классе описывать методы класса Юнит, классы Войн и Маг получат их через наследование.

Наследование — механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии классов.

**Ключевое слово super**

С помощью ключевого слова super можно обратиться к члену суперкласса (класса-родителя) из класса наследника. То есть, если через ключевое слово this мы получаем доступ к методу текщего класса, то через ключевое слово мы можем получить доступ к методу суперкласса (класса-родителя).

**Инкапсуляция**

Следующий принцип ООП — Инкапсуляция. Инкапсуляция позволит скрыть детали реализации, и открыть только то что необходимо в последующем использовании. Другими словами инкапсуляция – это механизм контроля доступа.

Зачем же это нужно?

Думаю, вам бы не хотелось, чтобы кто-то, что-то изменял в написанной вами библиотеки.

И если это опытный программист, то это простить еще можно, но все равно не приятно, а вот если это начинающий или не осторожный который с легкой руки задумает изменить код, да ещё не в ту степь, нам ведь такого не хочется! Чтобы обезопасить себя от таких поступков, существует инкапсуляция.

Цель инкапсуляции – уйти от зависимости внешнего интерфейса класса (то, что могут использовать другие классы) от реализации. Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса. Давайте рассмотрим, как ею пользоваться.

Существует 4 вида модификаторов доступа: public, protected, private и default.

Public – уровень предполагает  доступ к компоненту с этим модификатором из экземпляра любого класса и любого пакета.

Protected – уровень  предполагает  доступ к компоненту с этим модификатором из экземпляров родного класса и классов-потомков, независимо от того, в каком пакете они находятся.

Default – уровень предполагает  доступ к компоненту с этим модификатором из экземпляров любых классов, находящихся в одном пакете с этим классом.

Private – уровень предполагает  доступ к компоненту с этим модификатором только из этого класса.

То есть, если мы захотим закрыть поле name главного класса Unit от всех, то мы сделаем методы setName и getName() private. Тогда даже классы-наследники не смогут воспользоваться этими методами.

Если методы являются доступными для какого-либо класса (protected для наследников и public для других классов и пакетов), то мы можем воспользоваться предоставленной разработчиком возможностью и изменить код открытого метода. То есть переопределить, например, в классе-наследнике открытый метод и написать в теле метода свой код. Это скажется на функционировании класса-наследника, но не класса-родителя.

Остальные уровни доступа мы подробно рассмотрим уже на практике и увидим как в зависимости от уровня доступа меняется механизм использования свойств и методов.

**Полиморфизм**

Полиморфизм — это реализация одинакового по смыслу действия различным способом в зависимости от типа объекта.

Давайте вернем доступ до свойства name на public и рассмотрим следующий пример. Например, мы хотим для Война и Мага добавлять к имени приставку «Warrior» и «Wizard» соответственно.

Тогда мы переопределим метод setName в классах Воин и Маг. И теперь попытаемся вызвать один и тот же метод setName для класса Воин и для класса Маг. При вызове метода setName родительского класса для объектов разного типа компилятор сам понимает, какой метод нужно использовать.

**Итог**

**Объе́ктно-ориенти́рованное программи́рование (ООП)** — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Еще раз уточним различие между классом и объектом:

- класс — это описание типа данных, некая абстрактная логическая конструкция;

- объект — это нечто реально существующее. Он занимает место в памяти.

### **Обзор классов-оболочек примитивных типов**

Мы уже хорошо знаем примитивные (базовые) типы данных языка Java, не являющиеся классами. Эти типы данных не вписываются в общую «объектно-ориентированную картину мира» Java.

Для того чтобы с примитивными типами данных можно было работать так же, как с остальными объектными, для них существуют классы-оболочки (wrapper classes). Они инкапсулируют в себе эти примитивные типы и предоставляют широкий набор методов для работы с ними. Такой прием достаточно широко применяется.

Далее рассмотрим классы-оболочки, соответствующие шести примитивным числовым типам данных.

| Примитивный тип | Класс-оболочка |
| --- | --- |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

Создавать объект любого из классов-оболочек просто:

*Integer i = Integer.valueOf(50);*

Автоупаковка (autoboxing) — автоматическая инкапсуляции данных примитивного типа в его класс-оболочку. При этом не нужно создавать объекты с помощью операции new. Автораспаковка (auto-unboxing) — обратный процесс преобразования объектов в соответствующие им примитивные типы.

Подобные средства появились начиная с версии JDK 5. Добавление подобных возможностей значительно упрощает написание кода.

Ниже приведен пример современного способа создания объекта типа Integer:

*Integer iObj = 5; //автоупаковка*

*int i = iObj; //автораспаковка*

# Класс String

Строка представляет собой последовательность символов. Для работы со строками в Java определен класс String, который предоставляет ряд методов для манипуляции строками.

Для создания новой строки мы можем использовать один из конструкторов класса String, либо напрямую присвоить строку в двойных кавычках:

*String str1 = "Java";*

*String str2 = new String(); // пустая строка*

*String str3 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});*

При работе со строками важно понимать, что объект String является неизменяемым (immutable). То есть при любых операциях над строкой, которые изменяют эту строку, фактически будет создаваться новая строка.

Поскольку строка рассматривается как набор символов, то мы можем применить метод length() для нахождения длины строки или длины набора символов.

System.out.println("Test String".length());

Физически объект String представляет собой ссылку на область в памяти, в которой размещены символы.

То есть:

*String s1 = “”Это строка”;*

*String s2 = “”Это строка”;*

*if(s1 == s2) // false.*

Для сравнения именно текста строк можно воспользоваться методом equals:

*String x = "Test String";*

*System.out.println("Test String".equals(x)); // true*

Для перевода экземпляра любого Java-класса или любого примитивного типа данных к строковому представлению, можно использовать метод valueOf:

*String c = String.valueOf(123.4F);*

Переменная String может не указывать на какой-либо объект и иметь значение null, но значение null не эквивалентно пустой строке. Например, в следующем случае мы столкнемся с ошибкой выполнения:

*String s = null;   // строка не указывает на объект*

*if(s.length()==0) System.out.println("String is empty"); //Ошибка*

Так как переменная не указывает ни на какой объект String, то соответственно мы не можем обращаться к методам объекта String. Чтобы избежать подобных ошибок, можно предварительно проверять строку на null:

*String s = null;   // строка не указывает на объект*

*if(s!=null && s.length()==0) System.out.println("String is empty");*