Урок 5

Введение в ООП

Введение в объектно-ориентированное программирование, классы, объекты, конструкторы, инкапсуляция, модификаторы доступа

Что такое класс

Класс определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Наиболее важная особенность класса состоит в том, что он определяет новый тип данных, которым можно воспользоваться для создания объектов этого типа, т.е. класс — это шаблон (чертеж), по которому создаются объекты (экземпляры класса). Для определения формы и сущности класса указываются данные, которые он должен содержать, а также код, воздействующий на эти данные.

Если мы хотим работать в нашем приложении с документами, то необходимо для начала объяснить что такое документ, описать его в виде класса (чертежа) Document. Рассказать какие у него должны быть свойства: название, содержание, количество страниц, информация о том, кем он подписан и т.д. В этом же классе мы описываем что можно делать с документами: печатать в консоль, подписывать, изменять содержание, название и т.д. Результатом такого описания и будет наш класс Document. Однако это по-прежнему всего лишь чертеж. Если нам нужны конкретные документы, то необходимо создавать объекты: документ №1, документ №2, документ №3. Все эти документы будут иметь одну и ту же структуру (название, содержание, …), с ними можно выполнять одни и те же действия, НО наполнение будет разным (в первом документе содержится приказ о назначении работника на должность, во втором, о выдаче премии отделу разработки и т.д.).

Заметка. Приведенный пример является абстрактным, и нет необходимости говорит о том, что структура таких документов будет совершенно разной.

Посмотрим на упрощенную форму объявления класса:

```
модификатор_доступа class имя_класса {
    модификатор_доступа тип_переменной имя_поля; // первое поле
    модификатор_доступа тип_переменной имя_поля; // второе поле
    // ...
    модификатор_доступа тип_переменной имя_поля; // n-е поле

    модификатор_доступа имя_коструктора(список_аргументов) {
        // ...
    }

    модификатор_доступа тип_метода имя_метода(список_аргументов) {
        // тело метода
    }
    // ...
    модификатор_доступа тип_метода имя_метода(список_аргументов) {
        // тело метода
    }
}
```

Пример класса User (пользователь):

```
public class User {
    private int id;
```

```
    private String name;
    private String position;
    private int age;

    public User(int id, String name, String position, int age) {
        this.id = id;
        this.name = name;
        this.position = position;
        this.age = age;
    }

    public void info() {
        System.out.println("id: " + id + "; Имя пользователя: " + name+ "; 
Должность: " + position + "; Возраст: " + age);
    }

    public void changePosition(String position) {
        this.position = position;
        System.out.println("Пользователь " + name + " получил новую должность: " 
+ position);
    }
}
```

Как правило, переменные, объявленные в классе, описывают свойства будущих объектов, а методы - их поведение. Например, в классе User (пользователь) можно объявить переменные: int id, String name, String position, int age; которые говорят о том, что у пользователя есть идентификационный номер (id), имя (name), должность (position) и возраст (age). Методы info() и changePosition(), объявленные в классе User, означают что мы можем выводить информацию о нем в консоль (info) и изменять его должность (changePosition).

Переменные, объявленные в классе, называются полями экземпляра, каждый объект класса содержит собственные копии этих переменных, и изменение значения поля у одного объекта никак не повлияет на это же поле другого объекта.

Важно! Код должен содержаться либо в теле методов, либо в блоках инициализации и не может “висеть в воздухе”, как показано в следующем примере.

```
public class User {
    // ...

    public void info() {
        System.out.println("id: " + id + "; Имя пользователя: " + name + "; 
Должность: " + position + "; Возраст: " + age);
    }

    age++; // Ошибка
    System.out.println(age); // Ошибка

    public void changePosition(String position) {
        this.position = position;
        System.out.println("Пользователь " + name + " получил новую должность: " +
```

```
position);
    }
}
```

Поля экземпляра и методы, определённые в классе, называются членами класса. В большинстве классов действия над полями осуществляются через его методы.

Первый класс

Представим, что нам необходимо работать в нашем приложении с котами. Java ничего не знает о том, что такое коты, поэтому нам необходимо создать новый класс (тип данных), и объяснить что же такое кот. Для этого начнем потихоньку прописывать класс Cat. Пусть у котов есть три свойства: name (кличка), color (цвет) и age (возраст); и они пока ничего не умеют делать.

```
public class Cat {
    String name;
    String color;
    int age;
}
```

Важно! Имя класса должно совпадать с именем файла, в котором он объявлен, т.е. класс Cat должен находиться в файле Cat.java

Итак, мы рассказали Java что такое коты, теперь если мы хотим создать в нашем приложении кота, следует воспользоваться следующим оператором.

```
Cat cat1 = new Cat();
```

Подробный разбор того, что происходит в этой строке, будет проведен в следующем пункте. Пока же нам достаточно знать, что мы создали объект типа Cat (экземпляр класса Cat), и для того чтобы с ним работать, положили его в переменную, которой присвоили имя cat1. На самом деле, в переменной не лежит весь объект, а только ссылка где его искать в памяти, но об этом позже.

Объект cat1 создан по “чертежу” Cat, и значит у него есть поля name, color, age, с которыми можно работать (получать или изменять их значения). Для доступа к полям объекта служит операция-точка, которая связывает имя объекта с именем поля. Например, чтобы присвоить полю color объекта cat1 значение “White”, нужно выполнить следующий оператор:

```
cat1.color = "Белый";
```

Операция-точка служит для доступа к полям и методам объекта по его имени. Рассмотрим пример консольного приложения, работающего с объектами класса Cat.

```
public class CatDemoApp {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
        Cat cat2 = new Cat();
        cat1.name = "Барсик";
        cat1.color = "Белый";
        cat1.age = 4;
        cat2.name = "Мурзик";
```

```
        cat2.color = "Черный";
        cat2.age = 6;
        System.out.println("Кот1 имя: " + cat1.name + " цвет: " + cat1.color + "
возраст: " + cat1.age);
        System.out.println("Кот2 имя: " + cat2.name + " цвет: " + cat2.color + "
возраст: " + cat2.age);
    }
}
```

Результат работы программы:

```
Кот1 имя: Барсик цвет: Белый возраст: 4
Кот2 имя: Мурзик цвет: Черный возраст: 6
```

Вначале мы создали два объекта типа Cat: cat1 и cat2, соответственно они имеют одинаковый набор полей (name, color, age), однако каждому из них мы в эти поля записали разные значения. Как видно из результатом печати в консоле, изменение значения полей одного объекта, никак не влияет на значения полей другого объекта. Данные объектов cat1 и cat2 изолированы друг от друга.

Создание объектов

Как создавать новые типы данных (классы) мы разобрались, мельком посмотрели и как создаются объекты наших классов. Давайте теперь поподробнее разберем как создавать объекты, и что при этом происходит.

Создание объекта проходит в два этапа. Сначала создается переменная, имеющая интересующий нас тип (в данном случае Cat), в нее мы сможем записать ссылку на будущий объект (поэтому при работе с классами и объектами мы говорим о ссылочных типах данных). Затем необходимо выделить память под наш объект, создать и положить объект в выделенную часть памяти, и сохранить ссылку на этот объект в памяти в нашу переменную.

Заметка. Область памяти, в которой создаются и хранятся объекты, называется кучей (heap).

Для непосредственного создания объекта применяется оператор new, который динамически резервирует память под объект и возвращает ссылку на него, в общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором new.

```
public static void main(String[] args) {
    Cat cat1;
    cat1 = new Cat();
}
```

В первой строке кода переменная cat1 объявляется как ссылка на объект типа Cat и пока ещё не ссылается на конкретный объект (первоначально значение переменной cat1 равно null). В следующей строке выделяется память для объекта типа Cat, и в переменную cat1 сохраняется ссылка на него. После выполнения второй строки кода переменную cat1 можно использовать так, как если бы она была объектом типа Cat. Обычно новый объект создается в одну строку (*Cat cat1 = new Cat()*).

Подробное рассмотрение оператора new

Оператор new динамически выделяет память для нового объекта, общая форма применения этого оператора имеет следующий вид:

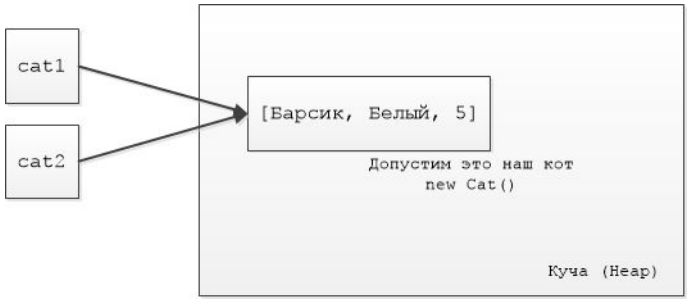
```
Имя_класса имя_переменной = new Имя_класса();
```

Имя_класса() в правой части выполняет вызов конструктора данного класса, который позволяет подготовить наш объект к работе.

Рассмотрим еще один пример, в котором создаются новые объекты.

```
public static void main(String[] args) {
    Cat cat1 = new Cat();
    Cat cat2 = cat1;
}
```

На первый взгляд может показаться, что переменной cat2 присваивается ссылка на копию объекта cat1, т.е. переменные cat1 и cat2 будут ссылаться на разные объекты в памяти. **Но это не так**. На самом деле cat1 и cat2 будут ссылаться на один и тот же объект. Присваивание переменной cat1 значения переменной cat2 не привело к выделению области памяти или копированию объекта, лишь к тому, что переменная cat2 ссылается на тот же объект, что и переменная cat1.



Таким образом, любые изменения, внесённые в объекте по ссылке cat2, окажут влияние на объект, на который ссылается переменная cat1, поскольку это один и тот же объект в памяти.

Конструкторы

Давайте еще раз взглянем на один из предыдущих примеров.

```
public class CatDemoApp {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
        cat1.name = "Барсик";
        cat1.color = "Белый";
        cat1.age = 4;
        System.out.println("Кот1 имя: " + cat1.name + " цвет: " + cat1.color + "
возраст: " + cat1.age);
    }
}
```

Чтобы создать объект мы тратим одну строку кода (*Cat cat1 = new Cat()*). Поля этого объекта заполнятся автоматически значениями по-умолчанию (целочисленные - 0, логические - false, ссылочные - null и т.д.). Нам бы хотелось дать коту какое-то имя, указать его возраст и цвет, поэтому мы прописываем еще три строки кода. В таком подходе есть несколько недостатков: во-первых, мы напрямую обращаемся к полям объекта (чего не стоит делать, в соответствии с принципами инкапсуляции, о которых речь пойдет чуть позже), а во-вторых, если полей у класса будет намного больше, то для создания всего лишь одного объекта будет уходить 10-20+ строк кода, что очень

неудобно. Было бы неплохо иметь возможность сразу при создании объекта указывать значения его полей.

Для инициализации объектов при создании в Java предназначены **конструкторы**. Имя конструктора обязательно должно совпадать с именем класса, а синтаксис аналогичен синтаксису метода. Если создаться конструктор класса Cat как показано ниже, он автоматически будет вызываться при создании объекта.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat() {
        System.out.println("Это конструктор класса Cat") ;
        name = "Барсик";
        color = "Белый";
        age = 2;
    }
}

public class MainClass {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
    }
}
```

Теперь, при создании объектов класса Cat, все коты будут иметь одинаковые имена, цвет и возраст (а именно, белый двухлетний Барсик).

Заметка. Еще раз обращаем внимание, что в строке `Cat cat1 = new Cat();` подчеркнутая часть кода и есть вызов конструктора класса Cat.

Важно! У классов **всегда** есть конструктор. Даже если вы не пропишите свою реализацию конструктора, то Java автоматически создаст пустой конструктор по-умолчанию. Для класса Cat, он будет выглядеть так:

```
public Cat() {
}
```

Параметризированные конструкторы

При использовании конструктора из предыдущего примера, все созданные коты будут одинаковыми, пока мы вручную не поменяем значения их полей. Чтобы можно было указывать начальные значения полей наших объектов необходимо создать параметризированный конструктор.

Важно! В приведенном ниже примере, в аргументах конструктора используется нижнее подчеркивание _, это сделано для упрощения понимания логики заполнения полей объекта. И в будущем будет заменено на более корректное использование ключевого слова this.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String _name, String _color, int _age) {
        name = _name;
        color = _color;
        age = _age;
    }
}
```

При такой форме конструктора, когда мы будем кота, необходимо будет обязательно указать его имя, цвет и возраст. Набор полей, которые будут заполнены через конструктор, вы определяете сами, то есть вы не обязаны все поля, которые есть в классе записывать в аргументы конструктора.

```
public static void main(String[] args) {
    Cat cat1 = new Cat("Барсик", "Коричневый", 4);
    Cat cat2 = new Cat("Мурзик", "Белый", 5);
}
```

Наборы значение (Барсик, Коричневый, 4) и (Мурзик, Белый, 5) будут переданы в качестве аргументов конструктора (_name, _color, _age), а конструктор уже перезапишет полученные значения в поля объект (name, color, age). То есть начальные значения полей каждого из объектов будет определяться тем, что мы передадим ему в конструкторе. Как видите, теперь нам нет необходимости обращаться напрямую к полям объектов, и мы в одну строку можем проинициализировать наш новый объект.

Ключевое слово this

Иногда требуется, чтобы метод ссылался на вызвавший его объект. Для этой цели в Java определено ключевое слово this. Им можно пользоваться в теле любого метода для ссылки на текущий объект, т.е. объект у которого был вызван этот метод.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }
}
```

Эта версия конструктора действует точно так же, как и предыдущая. Ключевое слово this применяется в данном случае для того, чтобы отличить аргумент конструктора от поля объекта.

Перегрузка конструкторов

Наряду с перегрузкой обычных методов возможна перегрузка и конструкторов. Мы можем как не объявлять ни одного конструктора, так и объявить их несколько. Также как и при перегрузке методов,

имеет значение набор аргументов, не может быть нескольких конструкторов с одним и тем же набором аргументов.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    public Cat(String name) {
        this.name = name;
        this.color = "Неизвестно";
        this.age = 1;
    }
}
```

Важно! Как только вы создали в классе свою реализацию конструктора, конструктор по-умолчанию автоматически создаваться не будет. И если вам понадобится такая форма конструктора (в которой нет аргументов, и которая ничего не делает), необходимо будет конструктор по-умолчанию вручную.

```
  public Cat() {
  }
```

В этом случае допустимы будут следующие варианты создания объектов:

```
public static void main(String[] args) {
    // Cat cat1 = new Cat(); этот конструктор больше не работает
    Cat cat2 = new Cat("Барсик");
    Cat cat3 = new Cat("Мурзик", "Белый", 5);
}
```

Соответствующий перегружаемый конструктор вызывается в зависимости от аргументов, указываемых при выполнении оператора new.

Важно! Не лишним будет напомнить что у классов **всегда** есть конструктор, даже если вы не пропишите свою реализацию конструктора.

Инкапсуляция

Инкапсуляция связывает данные с манипулирующим ими кодом и позволяет управлять доступом к членам класса из отдельных частей программы, предоставляя доступ только с помощью определенного ряда методов, что позволяет предотвратить злоупотребление этими данными.

То есть класс должен представлять собой «черный ящик», которым можно пользоваться, но его внутренний механизм защищен от повреждений.

Способ доступа к члену класса определяется модификатором доступа, присутствующим в его объявлении. Некоторые аспекты управления доступом связаны, главным образом, с наследованием и пакетами, и будут рассмотрены позднее. В Java определяются следующие модификаторы доступа: public, private и protected, а также уровень доступа, предоставляемый по умолчанию.

Любой **public** член класса доступен из любой части программы. Компонент, объявленный как **private**, доступен только внутри класса, в котором объявлен. Если в объявлении члена класса отсутствует явно указанный модификатор доступа (**default**), то он доступен для подклассов и других классов из данного пакета. Если же требуется, чтобы элемент был доступен за пределами его текущего пакета, но только классам, непосредственно производным от данного класса, то такой элемент должен быть объявлен как **protected**.

Модификатор доступа предшествует остальной спецификации типа члена.

```
public int num;
protected char symb;
boolean active;

private void calculate(float x1, float x2) {
    // ...
}
```

Как правило, доступ к данным объекта должен осуществляться только через методы, определённые в классе этого объекта. Поле экземпляра вполне может быть открытым, но на то должны иметься веские основания. Для доступа к данным обычно используются методы: геттеры и сеттеры.

Геттер позволяет узнать содержимое поля, его тип совпадает с типом поля для которого он создан, а имя, как правило, начинается со слова get, к которому добавляется имя поля.

Сеттер используется для изменения значения поля, имеет тип void и именуется по аналогии с геттером, только get заменяется на set. Сеттер позволяет добавлять ограничения на изменение полей — в примере ниже с помощью сеттера не получится указать коту отрицательный или нулевой возраст.

```
public class Cat {
    private String name;
    private int age;

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        } else {
            System.out.println("Введен некорректный возраст");
        }
    }
```

```
    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Заметка. Если для поля создан только геттер, то вне класса это поле будет доступно только для чтения. Если ни создан ни геттер, ни сеттер, то работа с полем снаружи класса может осуществляться только косвенно, через другие методы этого класса. (Пусть в классе Cat есть поле вес, оно private и для него нет геттеров и сеттеров. Тогда мы не можем через сеттер изменить вес кота напрямую, но если мы его покормим, то кот может сам набрать вес. Мы не можем запросить вес через геттер и получить конкретное значение, но у кота может быть метод info(), который выведет в консоль нам величину, записанную в поле weight.

Особенности всех уровней доступа в языке Java сведены в таблицу:

	private	Модификатор отсутствует	protected	public
Один и тот же класс	+	+	+	+
Подкласс, производный от класса из того же самого пакета	-	+	+	+
Класс из того же самого пакета, не являющийся подклассом	-	+	+	+
Подкласс, производный от класса другого пакета	-	-	+	+
Класс из другого пакета, не являющийся подклассом, производный от класса данного пакета	-	-	-	+

Дополнительные вопросы

Сборка «мусора». Поскольку выделение памяти под объекты осуществляется динамически с помощью оператора new, в процессе выполнения программы необходимо периодически удалять объекты из памяти для ее очистки, иначе она может и закончиться. В Java освобождение оперативной памяти осуществляется автоматически и называется сборкой «мусора». При отсутствии любых ссылок на объект считается, что он больше не нужен, и занимаемую им память можно освободить. Во время выполнения программы сборка «мусора» выполняется изредка и не будет выполняться немедленно, как только один или несколько объектов больше не используются.

Ключевое слово static. Иногда возникает необходимость создать поле класса, общее для всех объектов этого класса, или метод, который можно было бы использовать без необходимости создания объектов класса, в котором он прописан. Обращение к такому полю или методу должно осуществляться через имя класса. Для этого при объявлении поля или метода указывается ключевое слово static. Когда член класса объявлен как static (статический), он доступен до создания любых объектов его класса и без ссылки на конкретный объект. Наиболее распространённым примером

статического члена служит метод main(). При создании объектов класса копии статических полей не создаются и все объекты этого класса используют одно и то же статическое поле.

На методы, объявленные как static, накладываются следующие ограничения:

1. Они могут непосредственно вызывать только другие статические методы.
2. Им непосредственно доступны только статические переменные.
3. Они никоим образом не могут использовать ссылки типа this или super.

Домашнее задание

1. Создать класс "Сотрудник" с полями: ФИО, должность, email, телефон, зарплата, возраст.
2. Конструктор класса должен заполнять эти поля при создании объекта.
3. Внутри класса «Сотрудник» написать метод, который выводит информацию об объекте в консоль.
4. Создать массив из 5 сотрудников.

   Пример:
   Person[] persArray = new Person[5]; // Вначале объявляем массив объектов
   persArray[0] = new Person("Ivanov Ivan", "Engineer", "ivivan@mailbox.com", "892312312", 30000, 30); // потом для каждой ячейки массива задаем объект
   persArray[1] = new Person(...);
   ...
   persArray[4] = new Person(...);

5. С помощью цикла вывести информацию только о сотрудниках старше 40 лет.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. – М.: Вильямс, 2014. – 864 с.
2. Брюс Эккель Философия Java // 4-е изд.: Пер. с англ. – СПб.: Питер, 2016. – 1168 с.
3. Г. Шилдт Java 8. Полное руководство // 9-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 1376 с.
4. Г. Шилдт Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.

Используемая литература

1. Брюс Эккель Философия Java // 4-е изд.: Пер. с англ. – СПб.: Питер, 2016. – 1168 с.
2. Г. Шилдт Java 8. Полное руководство // 9-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 1376 с.
3. Г. Шилдт Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.