Java для молодых специалистов Банка ВТБ

Урок 1

Введение в ООП

Принципы ООП, классы, объекты.

Дополнительные материалы

Используемая литература

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

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**, — они 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 у другого.

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, определенные в классе, называются членами класса. В большинстве классов действия над полями осуществляются через его методы.

Первый класс

Представим, что нам необходимо работать в приложении с информацией о котах. Java ничего не знает о том, что это такое. Поэтому создадим новый класс (тип данных) и объясним. Для этого начнем прописывать класс **Cat**. Обозначим у котов три свойства: **name** (кличку), **color** (цвет) и **age** (возраст). 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: класс **Cat** должен находиться в файле **Cat.java**.

Рассказали Java, что такое коты. Теперь, если хотим создать в приложении объект кота, следует воспользоваться оператором:

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

Позже подробно разберем, что происходит в этой строке. Пока достаточно знать, что мы создали объект типа **Cat** (экземпляр класса **Cat**), и, 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.

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 имена объекта и 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
```

Сначала мы создали два объекта типа **Cat**: **cat1** и **cat2**, — у них одинаковый набор полей (**name**, **color**, **age**), но каждому в эти поля записали разные значения. Как видно по результату печати в консоли, изменение значений полей одного объекта не влияет на значения полей другого. Данные объектов **cat1** и **cat2** изолированы друг от друга.

Создание объектов

Мы рассмотрели, как создавать новые типы данных (классы) и их объекты. Разберем подробнее, 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 объект, создать и положить его в выделенную часть памяти, сохранить ссылку в переменную.

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

Непосредственно для создания объекта применяется оператор **new**, который резервирует память под объект и возвращает ссылку на него. Она 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 еще 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()**) — вместо двух строк из листинга выше.

Подробно об операторе new

Оператор **new** динамически выделяет память для нового объекта, общая форма его применения имеет вид:

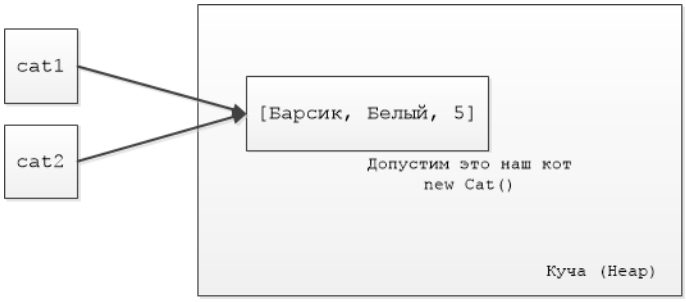
```
Имя_класса имя_переменной = new Имя_класса();
```

Имя_класса() в правой части выполняет вызов конструктора данного класса, который позволяет подготовить 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**, то есть переменные **cat1** и **cat2** будут ссылаться на разные объекты в памяти. Но это не так: **cat1** и **cat2** будут ссылаться на один объект. Присваивание переменной **cat1** значения переменной **cat2** не привело к выделению области памяти или копированию объекта, а только к тому, что переменная **cat2** ссылается на тот же объект, что и переменная **cat1**.



Изменения, внесенные в объекте по ссылке **cat2**, окажут влияние на объект, на который ссылается переменная **cat1**, — поскольку это один объект в памяти.

Конструкторы

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, тратим одну строку кода (**Cat cat1 = new Cat()**). Поля этого объекта заполнятся автоматически значениями по умолчанию (целочисленные — 0, логические — **false**, ссылочные — **null** и так далее). Чтобы дать коту имя, указать его возраст и цвет, прописываем еще три строки кода. В таком подходе есть несколько недостатков:

1. Мы напрямую обращаемся к полям объекта, чего не стоит делать в соответствии с принципами инкапсуляции.

2. Если полей у класса будет намного больше, то для создания одного объекта потребуется 10–20+ строк кода, что очень неудобно. Было бы неплохо иметь возможность сразу при создании объекта указывать значения его полей.

Для инициализации объектов при их создании в Java предназначены конструкторы. Имя конструктора обязательно должно совпадать с именем класса, а синтаксис — быть аналогичным синтаксису метода. Если создать 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** у всех будут одинаковые имена, цвет и возраст (белый двухлетний Барсик).

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

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

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

Параметризированные конструкторы

Все созданные с помощью конструктора из предыдущего примера объекты-коты будут одинаковыми, пока мы вручную не поменяем значения их полей. Чтобы можно было указывать начальные значения полей объектов, необходимо создать параметризированный конструктор.

Важная заметка! В приведенном ниже примере в аргументах конструктора используется нижнее подчеркивание _. Так проще понимать логику заполнения полей объекта. Потом заменим 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 будем создавать объект кота, надо будет обязательно указать его имя, цвет и возраст. Набор полей, которые будут заполнены через конструктор, вы определяете сами, и 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 значений (Барсик, Коричневый, 4) и (Мурзик, Белый, 5) будут переданы в качестве аргументов конструктора (**_name**, **_color**, **_age**), а конструктор запишет полученные значения в поля объекта (**name**, **color**, **age**). То есть начальные значения полей каждого из объектов будут определяться тем, что мы передадим ему в конструкторе. Теперь не надо обращаться напрямую к полям объектов — в одну строку можем проинициализировать новый объект.

Ключевое слово this

Чтобы метод ссылался на вызвавший его объект, в Java определено ключевое слово **this**. Им можно пользоваться в теле любого метода для ссылки на текущий объект — то есть тот, 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 так же, как предыдущая. Ключевое слово **this** применяется в данном случае для того, чтобы отличить аргумент конструктора от поля объекта.

Перегрузка конструкторов

Перегрузка возможна и для обычных методов, и для конструкторов. Мы можем не объявлять ни одного конструктора или объявить несколько. Как и при перегрузке методов, имеет значение набор аргументов: не может быть нескольких конструкторов с одинаковым набором.

```
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 добавить конструктор по умолчанию вручную.

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

В этом случае допустимы 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, которые указываются при выполнении оператора **new**.

Важно! У классов **всегда** есть конструктор, даже если вы не пропишете свою реализацию для него.

Инкапсуляция

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

То есть класс должен представлять собой «черный ящик»: им 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 с наследованием и пакетами — рассмотрим их позднее. В Java определяются следующие модификаторы доступа: **public**, **private** и **protected**, а также уровень доступа, предоставляемый по умолчанию. Если ни один из трех модификаторов явно не указан в том месте, где он ожидается, это означает, что применяется модификатор 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) {
    // ...
}
```

Инкапсуляция говорит о том, что доступ к данным объекта должен осуществляться только через методы. Если поле экземпляра открыто для изменения напрямую присваиванием через точку, то на такое нарушение инкапсуляции должны быть веские основания. Для доступа к данным обычно используются методы, определенные в классе этого объекта: геттеры и сеттеры. Они позволяют полностью контролировать процесс установки и получения значений.

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.

```java
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 приватного поля создан только геттер, то вне класса это поле будет доступно только для чтения. Если не создан ни геттер, ни сеттер, то работа с полем извне класса может осуществляться только косвенно, через другие методы этого класса.
Пусть в классе **Cat** есть private-поле «вес», для которого нет геттеров и сеттеров. Тогда мы не можем через сеттер изменить вес кота напрямую. Но если мы его покормим, 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 величину, записанную в поле **weight**.

Особенности всех уровней доступа в языке Java

	private	**Модификатор отсутствует**	**protected**	**public**
Один и тот же класс	+	+	+	+
Подкласс, производный от класса из того же пакета	-	+	+	+
Класс из того же 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 — очищать память, иначе она может закончиться. В Java освобождение оперативной памяти осуществляется автоматически и называется сборкой мусора. Если на объект нет ссылок, считается, что он больше не нужен, и занимаемую им память можно освободить. Во время выполнения программы сборка мусора выполняется с периодически, а не как только один или несколько объектов перестают использоваться.

Ключевое слово static. Иногда возникает необходимость создать поле класса, общее для всех его объектов, или метод, который можно использовать, не создавая объекты 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 статический, он доступен до создания любых объектов его класса и без ссылки на конкретный объект. Наиболее распространенным примером статического члена служит метод **main()**. При создании объектов класса копии статических полей не создаются и все объекты этого класса используют одно 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 не могут использовать ссылки типа **this** или **super**.

Это следствие того, что static-метод не связан ни с одним из объектов.

Основы наследования и полиморфизм

Одним из основополагающих принципов ООП является наследование, которое позволяет создать класс (суперкласс), определяющий общие черты набора классов, а затем этот общий класс могут наследовать более специализированные классы (подклассы), дополняя особыми характеристиками. Подкласс наследует члены, определенные в суперклассе, присоединяя к ним собственные.

Важно! Подкласс будет наследовать члены, определенные в суперклассе, в соответствии с их модификаторами доступа. Если у суперкласса будет private-поле, то подкласс не унаследует это поле.

Для реализации наследования используется ключевое слово **extends** в следующей форме:

```
class имя_подкласса extends имя_суперкласса
```

Представим, что в приложении нам придется работать с разными домашними животными. У всех должна быть кличка (**name**), и все должны уметь прыгать. Если нужен десяток разных классов животных, то поле **name** и метод **jump()** надо будет прописывать в каждом, дублируя код. Вместо этого можем создать суперкласс **Animal**, в котором описать общие для всех подклассов черты. После

этого создавать подклассы и наследоваться от класса **Animal**. В приведенном ниже примере представлена только структура Animal- и Cat-классов, и можно представить, что помимо **Cat** есть **Dog**, **Hamster** и другие.

```
public class Animal {
    String name;

    public Animal() {
    }

    public Animal(String name) {
        this.name = name;
    }

    public void animalInfo() {
        System.out.println("Животное: " + name);
    }

    public void jump() {
        System.out.println("Животное подпрыгнуло");
    }
}

public class Cat extends Animal {
    String color;

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

    public void catInfo() {
        System.out.println("Кот имя: " + name + " цвет: " + color);
    }
}

public class AnimalsApp {
    public static void main(String[] args) {
        Animal animal = new Animal("Дружок");
        Cat cat = new Cat("Барсик", "Белый");
        animal.animalInfo();
        cat.animalInfo();
        cat.catInfo();
    }
}

// Результат:
// Животное: Дружок
// Животное: Барсик
// Кот имя: Барсик цвет: Белый
```

Заметка. В приведенном примере специально оставлен конструктор по умолчанию для класса **Animal** и методы **animalInfo()** и **catInfo()**. Эти части кода в дальнейшем оптимизируются, как только мы разберемся в соответствующих темах.

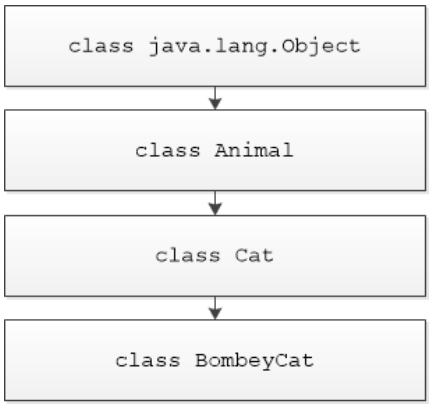
Подкласс **Cat** включает в себя все члены своего суперкласса **Animal**. Именно поэтому объект **cat** имеет доступ к методу **animalInfo()**, а в методе **catInfo()** возможна непосредственная ссылка на переменную **color**, как если бы она была частью класса **Cat**.

У объекта-кота появилось свойство цвета (**color**) и метод **catInfo()**, которых нет в суперклассе **Animal**.

Класс **Animal** является суперклассом для **Cat**, но он остается полностью независимым и самостоятельным классом. Если один класс является суперклассом для другого, это не исключает возможности его самостоятельного использования.

Важно! Для каждого создаваемого класса можно указать только один суперкласс — в Java не поддерживается множественное наследование. Если суперкласс не указан явно, то класс наследуется от класса **java.lang.Object** (в приведенном выше примере класс **Animal** является подклассом суперкласса **Object**).

Один подкласс может быть суперклассом другого подкласса.



Важно! Абсолютно все классы в Java являются прямыми или косвенными наследниками класса **Object** (из пакета **java.lang**). **Cat** является подклассом **Animal**, а **Animal** — подкласс **Object**, следовательно и **Cat** — тоже подкласс **Object**.

Ключевое слово super

Ключевое слово **super** означает обращение к суперклассу. У **super** две общие формы: первая служит для вызова конструктора суперкласса, вторая — для обращения к члену суперкласса, скрываемому членом подкласса.

Из подкласса можно вызывать конструктор, определенный в его суперклассе, используя следующую форму ключевого слова **super**:

```
super(список_аргументов)
```

Здесь **список_аргументов** определяет аргументы, требующиеся конструктору суперкласса.

Важно! Если необходимо вызвать конструктор суперкласса через **super()**, то этот вызов должен быть первым оператором, выполняемым в конструкторе подкласса. Стоит отметить, что если мы этого не сделаем, то Java сама первой строкой в конструкторе подкласса будет осуществлять вызов конструктора по умолчанию из суперкласса.

Такая конструкция позволяет заполнять даже поля суперкласса с модификатором доступа **private**. Например:

```
public class Animal {
    private int a;
    protected int z;

    public Animal(int a) {
        this.a = a;
    }
}

public class Cat extends Animal {
    private int b;
    protected int z;

    public Cat(int a, int b) {
        super(a);         // Вызываем конструктор Animal
        this.b = b;
    }

    public void test() {
        z = 10;           // Обращение к полю z класса Cat
        super.z = 20;     // Обращение к полю z класса Animal
    }
}

public class BombeyCat extends Cat {
    private int c;

    public BombeyCat(int a, int b, int c) {
        super(a, b);     // Вызываем конструктор Cat
        this.c = c;
    }
}
```

Вторая форма применения ключевого слова **super** действует подобно ключевому слову **this**, но ссылка указывает на суперкласс. Вторая форма пригодна, когда имена членов подкласса скрывают члены суперкласса с такими же именами: в примере выше поле **z** класса **Cat** скрывает поле **z**

суперкласса, поэтому для доступа к полю суперкласса используется запись **super.z**. Это справедливо и для методов.

Важно! Если вы только начинаете программировать, то КРАЙНЕ НЕ РЕКОМЕНДУЕТСЯ объявлять поля с одинаковыми именами в суперклассе и его подклассах — легко запутаться, с каким из полей работаете. Такое объявление переменных имеет смысл, только если без этого никак не обойтись и вы четко понимаете, что делаете.

Порядок вызова конструкторов

При вызове конструктора **BombeyCat** будут по цепочке вызваны конструкторы родительских классов, начиная с первого.

```
BombeyCat bombeyCat = new BombeyCat ();
// Animal() => Cat() => BombeyCat()
```

Конструкторы вызываются в порядке наследования, поскольку суперклассу ничего не известно о его подклассах, поэтому любая инициализация должна быть выполнена в нем независимо от инициализации, выполняемой подклассом. Следовательно, она должна выполняться в первую очередь.

Переопределение методов

Если у супер- и подкласса совпадают сигнатуры методов, то говорят, что метод из подкласса переопределяет метод из суперкласса. Когда переопределенный метод вызывается из своего подкласса, он всегда ссылается на свой вариант, определенный в подклассе. А вариант метода, определенного в суперклассе, будет скрыт.

Допустим, любое животное в нашем приложении должно уметь подавать голос. По умолчанию при вызове метода **voice()** будем видеть стандартное сообщение, что животное издало свой звук. Для тех классов, где подразумеваются конкретные звуки, можем переопределить метод **voice()**, и конкретизировать, например, что **Cat** мяукает.

```
public class Animal {
    void voice() {
        System.out.println("Животное издало звук");
    }
}
public class Dog extends Animal {
}

public class Cat extends Animal {
    @Override
    void voice() {
        System.out.println("Кот мяукнул");
    }
}

public class AnimalsApp {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Cat cat = new Cat();
        Dog dog = new Dog();
        animal.voice();
        cat.voice();
        dog.voice();
    }
}

// Результат:
// Животное издало звук
// Кот мяукнул
// Животное издало звук
```

Когда метод **voice()** вызывает объект типа **Cat**, выбирается вариант этого метода, определенный в классе **Cat**. У объекта класса **Dog** своей реализации метода **voice()** не было, поэтому **dog** выполнил вариант метода **voice()**, который описан в суперклассе **Animal**.

Важно! Над методами подклассов, переопределяющими методы суперклассов, можно ставить аннотацию *@Override*, но она не является обязательной. Она всего лишь проверит, действительно ли в родительском классе есть метод, который вы собрались переопределять.

Если при переопределении метода необходима функциональность из этого метода суперкласса, можно использовать конструкцию **super.method()**. Пример и результат использования:

```
public class Cat extends Animal {
    @Override
    public void voice() {
        super.voice(); // вызываем метод voice() суперкласса
        System.out.println("Кот мяукнул");
    }
}

public class CatsApp {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.voice();
    }
}

// Результат:
// Животное издало звук
// Кот мяукнул
```

Важно! Переопределение методов выполняется только в том случае, если имя, список аргументов и возвращаемый тип обоих методов совпадают. В противном случае методы считаются перегруженными.
Переопределенные методы позволяют поддерживать в Java полиморфизм во время выполнения. Благодаря ему можно определить в общем классе методы, которые станут общими для всех производных от него классов, а в подклассах — конкретные реализации некоторых или всех этих методов.

При работе с суперклассами и подклассами можно создать ссылку на суперкласс и записать в нее объект подкласса.

```
public class DemoApp {
    public static void main(String[] args) {
        Animal animal = new Cat();
        animal.voice();
        if (animal instanceof Cat) {
            ((Cat)animal).methodFromCatClass();
            System.out.println("В animal действительно лежит кот");
        }
    }
}
```

Несмотря на то, что объект типа **Cat** лежит в переменной типа **Animal**, реализацию метода **voice()** он будет брать именно ту, которая ближе к нему, то есть описанную в классе **Cat**. При обращении к объекту типа **Cat** через ссылку на **Animal** будем видеть только те методы и поля, которые предоставляет класс **Animal**.

Если в классе **Cat** есть метод **methodFromCatClass()**, который мы захотим выполнить через переменную **animal**, необходимо явно указать класс, с которым мы работаем: **((Cat)animal)**. Это называется casting — «закастить». После этого сможем пользоваться методами и полями из класса **Cat**.

Если в **animal** будет лежать ссылка не на объект типа **Cat** и вы используете запись вида **((Cat)animal)**, эта операция приведет к ошибке в работе программы — исключению **ClassCastException**. Чтобы избежать этого, можно воспользоваться оператором **instanceof**, который проверяет принадлежность объекта к классу.

Класс Object

Абсолютно все классы в Java наследуются от класса **java.lang.Object**. Особое внимание необходимо уделить нескольким методам этого класса: equals(), hashCode() и toString().

Начнем с самого простого - метода **toString()**, который предназначен для преобразования любого объекта в текстовый вид. Допустим мы создали объект класса Cat и передаем его в качестве аргумента методу System.out.println(), что же мы увидим в консоли?

```
public class Cat {
   private String name;

   public Cat(String name) {
       this.name = name;
   }

   public static void main(String[] args) {
       Cat cat = new Cat("Барсик");
       System.out.println(cat);
   }
}

// Результат:
// Cat@1b6d3586
```

Получился какой-то набор символов. Если присмотреться, то мы видим что вместо объекта отпечаталось имя класса, символ @, и потом последовательность `Cat@1b6d3586`. Откуда все это берется? По-умолчанию у объекта cat срабатывает метод toString(), доставшийся ему по наследованию от класса Object, и полученная строка печатается в консоль. Если заглянуть в класс Object, то увидим следующее:

```
public class Object {
   // ...
   public String toString() {
      return getClass().getName() + "@" + Integer.toHexString(hashCode());
   }
   // ...
}
```

Из этого куска кода видно, что при распечатке объекта, печатается имя его класса, символ @, и хэш-код в шестнадцатеричном представлении (что такое хэш-код скажем чуть-чуть позднее). Если мы хотим, чтобы печаталось что-то более осмысленное, то можем переопределить этот метод.

```
public class Cat {
    private String name;

    public Cat(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat: " + name;
    }

    public static void main(String[] args) {
        Cat cat = new Cat("Барсик");
        System.out.println(cat);
    }
}

// Результат:
// Cat: Барсик
```

Итак, метод toString() объясняет Java как мы хотим представлять объекты наших классов в текстовом виде.

Следующим важным методом является **hashCode()**. Его практическое применение вы встретите при работе с коллекциями (HashSet, LinkedHashSet, HashMap и др.), пока только посмотрим за что он отвечает, и как его можно переопределять. Метод hashCode() возвращает число типа int, в зависимости от содержимого объекта.

```
public class Object {
    // ...
    public native int hashCode();
    // ...
}
```

Мы не сможем посмотреть стандартную Java реализацию этого метода, так как он написан в нативном виде. Если в процессе работы программы у одного и того же объекта (при условии что он не меняет свое состояние) вызывать hashCode(), этот метод должен возвращать одно и то же значение. При этом между запусками программы, hashCode у одного и того же объекта не обязательно будет постоянным (при использовании стандартной реализации).

Давайте попробуем переопределить метод hashCode() в нашем классе Cat.

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

```
    public Cat(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat: " + name;
    }

    @Override
    public int hashCode() {
        return name.hashCode() + age * 71;
    }

    public static void main(String[] args) {
        Cat cat = new Cat("Барсик");
        System.out.println(cat);
    }
}
```

Теперь хэш-код объектов типа Cat, будет зависеть от имени и возраста кота.

Третий важный метод, о котором необходимо сказать, это конечно же **equals()**. Вы должны помнить, что для сравнения объектов вместо == необходимо использовать метод equals(). Но тут возникает вопрос, если мы написали свой собственный класс (Cat), то откуда Java узнает как сравнивать объекты этого класса, ответ - никак. Метод equals() в классе Object, по-умолчанию сравнивает пару объектов просто через оператор ==. Поэтому необходимо переопределять этот метод в наших классах.

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

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

    @Override
    public String toString() {
        return "Cat: " + name;
    }

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

```
        }
        Cat another = (Cat)o;
        return this.age == another.age && this.name.equals(another.name);
    }

    @Override
    public int hashCode() {
        return name.hashCode() + age * 71;
    }

    public static void main(String[] args) {
        Cat cat1 = new Cat("Барсик", 5);
        Cat cat2 = new Cat("Барсик", 5);
        System.out.println(cat1.equals(cat2));
    }
}
```

Результатом сравнения cat1 и cat2 будет теперь конечно же true. Итак, что же написано в методе equals()? Первым условием мы проверяем и не сравниваем ли мы объект самого с собой? (например, cat1.equals(cat1)). В таком случае, конечно же такие объекты равны. Второе условие проверяет, что объект, переданный в качестве аргумента существует и является объектом типа Cat, в противном случае говорим что сравнивать бесполезно, и возвращаем false. Если же мы дошли до return, то нам осталось только указать по значениям каких полей мы хотим проводить сравнение двух котов. В данном случае это поля name и age. Если их значения равны, значит и оба кота равны между собой. Вот теперь Java знает каким образом ей сравнивать объекты “нашего” класса Cat.

Важно! При переопределении методов hashCode() и equals() необходимо обязательно придерживаться следующего:

- Если объекты равны через метод equals(), то их hashCode() **обязательно** должны быть равны;
- Если объекты не равны по equals(), то **желательно** чтобы их hashCode() отличались, но этого не всегда удается достичь (так как hashCode() возвращает не уникальное число)

Абстрактные классы и методы

Иногда суперкласс требуется определить таким образом, чтобы объявить в нем структуру заданной абстракции, не предоставляя полную реализацию каждого метода. Например, определение метода **voice()** в классе **Animal** служит лишь в качестве шаблона, поскольку все животные издают разные звуки, а значит нет возможности прописать хоть какую-то реализацию этого метода в классе **Animal**. Для этого служит абстрактный метод — с модификатором **abstract**. Иногда их называют методами под ответственностью подкласса, поскольку в суперклассе для них реализации не предусмотрено и они обязательно должны быть переопределены в подклассе.

```
abstract void voice();
```

При указании ключевого слова **abstract** в объявлении метода его тело будет отсутствовать. Класс, содержащий хоть один абстрактный метод, должен быть объявлен как абстрактный — в объявлении класса так же добавляется ключевое слово **abstract**.

Нельзя создавать объекты абстрактного класса, поскольку он определен не полностью, и объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс, производный от абстрактного класса, обязан реализовать все абстрактные методы из своего суперкласса — при условии, что подкласс сам не является абстрактным. При этом абстрактный класс вполне может содержать конкретные реализации методов. Пример:

```
public abstract class Animal {
    public abstract void voice();

    public void jump() {
        System.out.println("Животное подпрыгнуло");
    }
}

public class Cat extends Animal {
    @Override
    public void voice() {
        System.out.println("Кот мяукнул");
    }
}
```

Важно! Что нужно помнить об абстрактных классах:

- нельзя создать объект абстрактного класса;
- в абстрактном классе могут быть конкретные реализации методов;
- если в классе объявлен хоть один абстрактный метод, сам класс должен быть объявлен абстрактным.

Несмотря на то, что абстрактные классы не позволяют получать экземпляры объектов, их все же можно применять для создания ссылок на объекты подклассов.

```
Animal a = new Cat();
```

Ключевое слово final в сочетании с наследованием

Существует несколько способов использования ключевого слова **final**:

1. Создание именованной константы.

```
final int MONTHS_COUNT = 12; // final в объявлении поля или переменной
```

2. Предотвращение переопределения методов: подклассы не могут переопределять final-метод.

```
public final void run() {  // final в объявлении метода
}
```

3. Запрет наследования от текущего класса.

```
public final class A {          // final в объявлении класса
}

// public class B extends A {  // Ошибка: класс A не может иметь подклассы
// }
```

Практическое задание

1. Создать классы **Собака**, **Домашний Кот**, **Тигр**, **Животное** (можете добавить два-три своих класса).
2. Животные могут бежать и плыть. В качестве аргумента каждому методу передается длина препятствия.
3. У каждого животного есть ограничения на действия (бег: кот — 200 м, собака — 500 м; плавание: кот — не умеет плавать, собака — 10 м). Результатом выполнения действия будет печать в консоль. Например: **dogBobik.run(150); -> 'Бобик пробежал 150 м'**.
4. Создать один массив с животными и заставляете их по очереди пробежать дистанцию и проплыть.
5. * Добавить подсчет созданных **Домашних Котов**, **Тигров**, **Собак**, **Животных**.

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. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.