Урок 6

Продвинутое ООП

Углубленное изучение вопросов объектно-ориентированного программирования: наследование, полиморфизм

Основы наследования и полиморфизм

Одним из основополагающих принципов ООП является наследование, который позволяет создать класс (суперкласс), определяющий какие-то общие черты набора классов, а затем этот общий класс может наследоваться другими, более специализированными классами (подклассами), каждый из которых будет добавлять свои особые характеристики. Подкласс наследует члены, определённые в суперклассе, добавляя к ним собственные.

Важно! Подкласс будет наследовать члены, определённые в суперклассе, в соответствии с модификаторами доступа этих членов. 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 нашем приложении нам придется работать с различными животными (считаем что они домашние), у всех этих животных должна быть кличка (name), и все они должны уметь прыгать. Если представить, что нам нужно штук 10 разных классов животных, то поле name и метод jump() надо будет в каждом из них прописывать, тем самым дублируя один и тот же код (а если одинаковых полей и методов больше?). Вместо дублирования кода, мы можем создать суперкласс Animal, в котором и описать эти общие для всех подклассов черты. После чего создавать подклассы и наследоваться от класса Animal. В приведенном ниже примере представлена только структура Animal и Cat классов, можно мысленно представить что помимо Cat у нас еще есть Dog, Hamster, и т.д.

```
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 оставлены конструктор по-умолчанию для класса Animal и методы animalInfo() и catInfo(). Эти части кода в дальнейшем немного оптимизируются, как только мы разберемся в соответствующих темах.

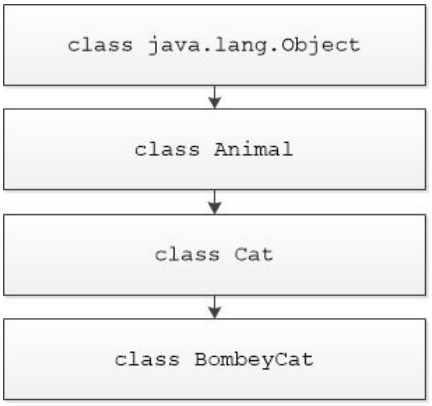
Подкласс Cat включает в себя все члены своего суперкласса Animal. Именно поэтому объект cat имеет доступ к методу animalInfo(), и в методе catInfo() возможна непосредственная ссылка на переменную color, как если бы она была частью класса Cat.

В свою очередь у кота появилось такое свойство как цвет (color) и метод catInfo(), которых нет в суперклассе Animal.

Несмотря на то что класс Animal является суперклассом для класса Cat, он в то же время остаётся полностью независимым и самостоятельным классом. То, что один класс является суперклассом для другого, совсем не исключает возможность 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)

Более того, 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 Cat тоже подкласс Object.

Ключевое слово super

У ключевого слова super имеются 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! Вызов метода super() **всегда должен быть** первым оператором, выполняемым в конструкторе подкласса. Если в конструкторе подкласса явно не использовать super(), то автоматически первой строкой будет вызываться конструктор 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);        // первым делом 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);    // первым делом вызываем конструктор Cat
        this.c = c;
    }
}
```

Вторая форма применения ключевого слова super действует подобно ключевому слову this, за исключением того, что ссылка всегда делается на суперкласс. Вторая форма наиболее пригодна в тех случаях, 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. То же справедливо и для методов.

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

Порядок вызова конструкторов

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

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

Конструкторы вызываются в порядке наследования, поскольку суперклассу ничего неизвестно о своих подклассах, и поэтому любая инициализация должна быть выполнена в нём совершенно независимо от любой 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 переопределённый метод вызывается из своего подкласса, он всегда ссылается на свой вариант, определённый в подклассе. А вариант метода, определённого в суперклассе, будет скрыт.

Пусть любое животное в нашем приложении должно уметь подавать голос, по-умолчанию при вызове метода voice() мы будем видеть стандартное сообщение, что животное издало какой-то звук. Для тех же классов, которые издают вполне конкретные звуки, мы можем переопределить метод voice(), и например, конкретизировать что Cat именно мяукает, а не что-то еще делает.

```
public class Animal {
    void voice() {
        System.out.println("Животное издало какой-то 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, определённый в классе Cat. У объекта же 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 такой метод, который вы собрались переопределять.

Если при переопределении метода необходим функционал из этого метода суперкласса, то 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 одинаковы. В противном случае оба метода считаются перегруженными.
Переопределенные методы позволяют поддерживать в Java полиморфизм во время выполнения, он позволяет 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 существует возможность 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 переменной типа Animal, реализацию метода voice() он будет брать именно ту, которая ближе к нему, то есть описанная в классе Cat. При обращении к объекту типа Cat через ссылку на Animal мы будем видеть только те методы и поля, которые предоставляет нам класс Animal.

Если же у нас в классе Cat есть некий метод methodFromCatClass() и мы захотим его все же выполнить через переменную animal, необходимо явно указать класс с которым мы работаем: `((Cat)animal)`. После чего мы сможем пользоваться методами и полями из класса Cat.

Если в animal будет лежать ссылка не на объект типа Cat и вы используется запись вида `((Cat)animal)`, это операция приведет к ошибке в работе программы (исключению ClassCastException). Чтобы избежать такой ошибки можно воспользоваться оператором instanceof, который проверяет принадлежность объекта к какому-либо классу.

Абстрактные классы и методы

Иногда суперкласс требуется определить таким образом, чтобы объявить в нём 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 этой цели служит абстрактный метод (с модификатором **abstract**). Иногда они называются

методами под ответственностью подкласса, поскольку в суперклассе для них никакой реализации не предусмотрено, и они обязательно должны быть переопределены в подклассе.

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

При указании ключевого слова abstract в объявлении метода, тело метода будет отсутствовать. Класс, содержащий хоть один абстрактный метод, должен быть объявлен как абстрактный (в объявлении класса также добавляется ключевое слово abstract).

Нельзя создавать объекты абстрактного класса, поскольку он определён не полностью. Кроме того, нельзя 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 есть хоть один абстрактный метод, он должен быть объявлен абстрактным;

Несмотря на то, что абстрактные классы не позволяют получать экземпляры объектов, их всё 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:

Первый способ: создание именованной константы.

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

Второй способ: предотвращение переопределения методов.

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

```
}
```

Третий способ: 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
// }
```

Домашнее задание

1. Создать классы Собака и Кот с наследованием от класса Животное.
2. Все животные могут бежать и плыть. В качестве параметра каждому методу передается длина препятствия. Результатом выполнения действия будет печать в консоль. (Например, dogBobik.run(150); -> 'Бобик пробежал 150 м.');
3. У каждого животного есть ограничения на действия (бег: кот 200 м., собака 500 м.; плавание: кот не умеет плавать, собака 10 м.).
4. * Добавить подсчет созданных котов, собак и 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. – 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.