Наследование классов Java

В этом материале мы разберём один из ключевых принципов ООП — наследование классов.

Зачем нужно наследование классов
Представьте, что вы используете класс, в котором много методов и переменных. Вам нужно добавить в этот класс новую функциональность, например ещё один метод. Однако в сам класс добавить его вы не можете: либо метод выбивается из общей логики класса, либо класс разработан не вами и менять его нельзя.

Конечно, вы можете создать новый класс и скопировать в него переменные и методы из того класса, который хотите расширить, но в этом случае у вас произойдёт дублирование кода. А дублирование кода — это плохая практика, поскольку, если в приложении какой-то код будет повторяться, поддерживать его будет сложнее.

Избегание дублирования кода — один из важных принципов программирования. Чтобы решить эту задачу, не изменяя исходный класс и не прибегая к дублированию кода, можно использовать наследование — создать наследник от класса, который вы хотите расширить.

Таким образом, первое, для чего нужно наследование, — это повторное использование кода уже существующих классов. Второе — примерно то же, для чего нужно вообще всё объектно ориентированное программирование.

А для чего оно, собственно, нужно? Одно из основных назначений ООП — отражение предметной области, для которой разрабатывается программа, с целью упрощения её проектирования и дальнейшего развития. Наследование помогает лучше моделировать предметную область.

Звучит довольно абстрактно, поэтому самое время перейти к практическому примеру.

Пример
У нас есть проект, в котором пока только один класс — Bus (в пер. с англ. — автобус):

```java
public class Bus {
    private double tankFullnessRate; // насколько заполнен топливный бак (от нуля до единицы)
    private final double consumptionRate; // расход топлива на 1 км
    /*
    условно, если бак заполнен на 100% (единица), а расход — 0,01, бака хватит на 100 км
     */

    private static int count;

    public Bus(double consumptionRate) {
        this.consumptionRate = consumptionRate;
        count++;
    }

    public boolean run(int distance) {    // проезд автобуса на определённое расстояние в километрах
        if (powerReserve() < distance) { // проверяется, хватит ли топлива на проезд этого расстояния
            return false;
        }
        tankFullnessRate -= distance * consumptionRate;
        return true; // из переменной с объёмом топливного бака вычитается путь в километрах, умноженный на
        // расход топлива на 1 км
    }

    public final void refuel(double tankRate) { // заправка автобуса
        double total = tankFullnessRate + tankRate; // насколько заполнен топливный бак плюс количество доливаемого топлива
        tankFullnessRate = total > 1 ? 1 : total; // если попытаться заправить бак больше чем на 100%, он заполнится только до 100%
    }

    public int powerReserve() {
       return (int) (tankFullnessRate / consumptionRate); // на сколько километров хватит оставшегося запаса топлива
    }

    public double getConsumptionRate() { // уровень потребления топлива
        return consumptionRate;
    }

    public double getTankFullnessRate() { // степень наполненности бака
```

```
        return tankFullnessRate;
    }

    // метод, который возвращает количество созданных
автобусов

    public static int getCount() {
        return count;
    }

    public static void setCount(int count) {
        Bus.count = count;
    }

    @Override
    public String toString() {
        return "Bus{" +
                "tankFullnessRate=" + tankFullnessRate +
                ", consumptionRate=" + consumptionRate +
                '}';
    }
}
```

Посмотрим, что есть в классе. В нём есть две переменные:

```
private double tankFullnessRate; // насколько заполнен
топливный бак (от нуля до единицы)
private final double consumptionRate; // расход топлива на 1
км
```

Как следует из комментария к первой переменной, в ней хранится число, которое показывает, насколько заполнен топливный бак. Если это число равно 0, значит, бак пустой. Если 1 — бак заполнен на 100%. А если, например, 0,5, то он заполнен наполовину.

Во второй переменной хранится число, показывающее расход топлива на 1 км. Расход тоже выражен в относительных единицах измерения степени заполненности бака (от нуля до единицы). То есть условно, если бак заполнен на 100% и указано, что расход — 0,01 (1%), бака хватит на 100 км.

Далее в классе написан конструктор с единственным параметром `consumptionRate`:

```
public Bus(double consumptionRate) {
```

```
    this.consumptionRate = consumptionRate;
    count++;
}
```

Поскольку эта переменная обозначена как final:

```
private final double consumptionRate;
```

она устанавливается при создании объекта этого класса и больше никогда не меняется.

Это значит, что для каждого отдельного автобуса расход топлива будет зафиксирован в памяти компьютера на протяжении всего срока его эксплуатации в качестве значения переменной этого объекта.

Далее идёт метод run() с параметром `distance`. Этот метод эмулирует проезд автобуса на определённое расстояние в километрах:

```
public boolean run(int distance) {    // проезд автобуса на определённое расстояние в километрах
    if (powerReserve() < distance) { // проверяется, хватит ли топлива на проезд этого расстояния
        return false;
    }
    tankFullnessRate -= distance * consumptionRate;
    return true; // из переменной с объёмом топливного бака вычитается путь в километрах, умноженный на
    // расход топлива на 1 км
}
```

В начале метода проверяется, хватит ли имеющегося топлива на проезд этого расстояния — метод **`powerReserve()`** как раз считает, на какое максимальное расстояние его хватит. И если запаса топлива достаточно, то из переменной с объёмом топливного бака будет вычитаться расход топлива автобуса, умноженный на расстояние в километрах:

```
distance * consumptionRate;
```

Затем метод вернёт значение true. Если запаса топлива недостаточно, то ничего не вычитается и метод просто возвращает false.

Далее следует метод refuel, с помощью которого автобус можно заправить. В этом методе также прописана защита от переполнения бака — если попытаться заправить его более чем на 100%, он заполнится только до 100%:

```
tankFullnessRate = total > 1 ? 1 : total
```

Следующий метод — powerReserve. Он рассчитывает, на сколько километров хватит оставшегося запаса топлива. Делает он это простым делением запаса топлива на расход и приведением результата к целому числу:

```
public int powerReserve() {
  return (int) (tankFullnessRate / consumptionRate); // на
сколько километров хватит оставшегося запаса топлива
}
```

В конце — обычные геттеры, которые возвращают уровень потребления топлива и степень наполненности бака, поскольку эти переменные приватные:

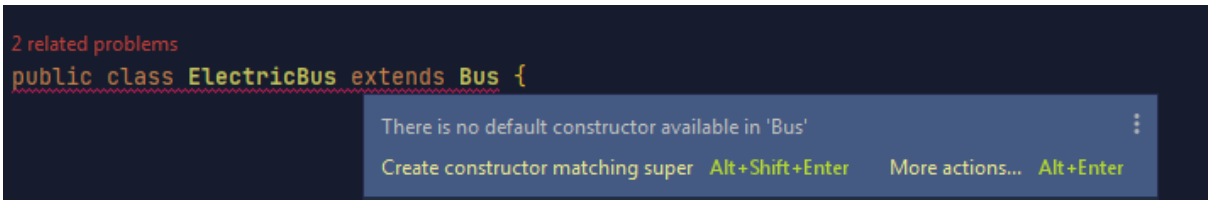
```
public double getConsumptionRate() { // уровень потребления
топлива
   return consumptionRate;
}

public double getTankFullnessRate() { // степень
наполненности бака
   return tankFullnessRate;
}
```

Теперь представьте, что ваша программа стала работать не только с автобусами, но и с электробусами, работающими по такому же принципу. В контексте функциональности, которая уже заложена в классе Bus, есть только одно отличие в способе расчёта резерва: у электробуса нельзя полностью сажать аккумулятор. Когда он близок к полной разрядке, на электробусе ехать нельзя, иначе у аккумулятора резко сократится срок службы. Поэтому у электробусов должен быть установлен минимальный порог заряда аккумулятора.

Используем наследование, чтобы, не дублируя код, создать электробус с учётом этой особенности.

Создадим класс ElectricBus. Поскольку мы хотим, чтобы он наследовался от класса Bus, необходимо написать после его имени ключевое слово extends и указать имя класса Bus. Дословный перевод extends с английского — «расширяет», то есть класс ElectricBus расширяет исходный класс Bus.

Обратите внимание, что среда разработки подчеркнула первую строку красным:



Изображение: Skillbox

```
public class ElectricBus extends Bus {

}
```

Если навести на неё курсор, мы увидим сообщение There is not default constructor available in Bus, которое в переводе означает, что в классе Bus нет доступного конструктора по умолчанию. Конструктор по умолчанию — это конструктор без параметров.

Поскольку в классе Bus есть только один конструктор и он с параметром, то конструктора по умолчанию в этом классе нет, что логично. Но в чём же проблема? Проблема в том, что, если в родительском классе определены конструкторы, при наследовании в дочернем необходимо вызвать один из них. Если нажать на красную лампочку, которая предлагает варианты быстрого исправления ситуации, то мы увидим предложение создать конструктор, соответствующий конструктору в классе-родителе. Класс-родитель в Java называется **суперклассом**.

Прислушаемся к рекомендации среды разработки и выберем этот вариант:

```
public ElectricBus(double consumptionRate) {
    super(consumptionRate);
}
```

Обратите внимание: в конструкторе вызван метод с именем super, в который передан параметр, появившийся в этом конструкторе. На самом деле это

не метод, а конструктор класса-родителя. Ключевым словом super в Java обозначается как раз класс-родитель.

Напомним, что мы хотели расширить исходный класс и добавить в него ещё один параметр — минимальный уровень заряда аккумулятора, ниже которого электробусом пользоваться нельзя.

Создадим такую переменную. Пусть это будет скрытая от посторонних глаз, приватная переменная, которую нельзя изменять, типа double. Назовём её minimalTankFullnessRate:

```
private final double minimalTankFullnessRate;
```

Добавим эту переменную в конструктор:

```
public ElectricBus(double consumptionRate, double
minimalTankFullnessRate) {
    super(consumptionRate);
    this.minimalTankFullnessRate = minimalTankFullnessRate;
}
```

Теперь новый класс идентичен родительскому классу Bus, за исключением конструктора и ещё одной переменной. Давайте в этом убедимся — для этого посмотрим, как работает класс Bus.

Перейдём в класс Main. Сначала создадим объект класса Bus. В скобках нужно указать параметры, в нашем случае — параметр потребления топлива. Создадим здесь число 0,001, то есть с расходом топлива на 0,1%, чтобы топлива хватило на 1000 километров, потом заправим автобус до полного бака.

Убедимся в том, что резерв топлива рассчитывается правильно.
Израсходуем топливо на 50 км. Снова проверим резерв.
Израсходуем топливо на 900 км. И снова проверим резерв.
Попробуем израсходовать топливо ещё на 100 км.
Запустим код и посмотрим, как он работает:

```
Bus bus = new Bus(0.001);
bus.refuel(1);
System.out.println("Резерв: " + bus.powerReserve());
System.out.println("Едем 50 км: " + bus.run(50));
```

```
System.out.println("Резерв: " + bus.powerReserve());
System.out.println("Едем 900 км: " + bus.run(900));
System.out.println("Резерв: " + bus.powerReserve());
System.out.println("Едем 100 км: " + bus.run(100));
System.out.println("Резерв: " + bus.powerReserve());
```

Видим, что всё работает:

```
Резерв: 1000
Едем 50 км: true
Резерв: 949
Едем 900 км: true
Резерв: 49
Едем 100 км: false
Резерв: 49
```

Изначально резерв рассчитан на 1000 км. Мы едем 50 км, и такая поездка нам удаётся. Резерв остаётся на 50 км меньше (здесь на 51 — это за счёт погрешностей вычислений с числом double). Дальше едем 900 км, резерв остаётся 49 км. Когда пытаемся проехать ещё 100 км (а резерв — 50), нам это не удаётся — метод run возвращает false, и резерв остаётся таким же.

Теперь проверим, что класс ElectricBus, который мы только что создали, действительно расширяет класс Bus. Для этого просто заменим здесь слово Bus на ElectricBus.

Всё подчеркнулось красным, поскольку конструктор в классе ElectricBus должен содержать два параметра. Укажем и второй параметр. Пусть он будет равен 10%:

```
Bus bus = new ElectricBus(0.001, 0.1);
```

То есть аккумулятор нельзя разряжать более чем на 10%.

Больше ничего менять не будем. Поскольку логику работы методов мы никак не меняли, поведение объекта этого класса, то есть класса ElectricBus, должно быть идентично поведению класса Bus. Давайте это проверим.

Запускаем код и видим тот же результат. Сначала — резерв 1000 км, едем 50 км — всё успешно, остаётся 949; дальше едем 900 км, остаётся 49; пытаемся

проехать ещё 100. Метод run, как и в классе Bus, возвращает false, и резерв не меняется. Всё работает точно так же.

Таким образом, наследуя один класс от другого, мы получаем новый класс, который по умолчанию перенимает все переменные и методы из родительского класса. Обратите внимание, что при создании объекта дочернего класса, в нашем случае ElectricBus, перед именем переменной можно использовать имя как дочернего, так и родительского класса (то есть можно было написать ElectricBus, а можно, как мы и сделали, — просто Bus).

Выводы

На этом занятии вы научились создавать классы-наследники. Делается это с помощью ключевого слова extends — в результате вы получаете дочерний класс, идентичный родительскому.

Вы разобрались, что если в родительском классе нет конструктора по умолчанию, а есть другие конструкторы, то их необходимо вызывать в конструкторах дочернего класса. Для этого следует использовать ключевое слово super, которое в Java применяется для обозначения родительского класса внутри дочернего.

Также вы узнали, что можно вносить изменения в дочерний класс, например добавлять переменные, конструкторы или по-другому реализовывать методы.