Переопределение методов

Разберём переопределение методов — расширение методов родительского класса в дочернем.

Переопределение методов по-английски называется overriding, override означает «замещать». Не путайте этот термин с перегрузкой методов (перегрузка — overloading). Перегрузка позволяет создавать в одном классе методы с одинаковым именем, но разными наборами параметров.

Рассмотрим переопределение методов на примере. Возьмём класс ElectricBus, который унаследован от класса Bus с добавлением ещё одной переменной — минимального порога заряда аккумулятора:

```
private final double minimalTankFullnessRate;
```

Эта переменная пока нигде не используется, даже среда разработки подсвечивает её серым. Используем её. Чтобы всё работало верно, нужно учитывать эту переменную при расчёте запаса хода, то есть в методе powerReserve(). Переопределим этот метод. Начнём вводить имя метода, автоматически появится подсказка. Нажмём на неё, и этот метод будет создан:

```
@Override
public int powerReserve() {
 return super.powerReserve();
}
```

Пропишем в методе новую логику. Сначала рассчитаем, сколько заряда аккумулятора можно расходовать. Создадим для этого новую переменную. Назовём её remainingRate — «оставшийся уровень». Из полного уровня вычтем минимальный уровень:

```
double remainingRate = getTankFullnessRate() -
minimalTankFullnessRate; // Оставшийся уровень.
```

Если это число будет отрицательным или равным нулю, значит, запаса хода больше нет, мы можем вернуть нуль и завершить на этом выполнение метода:

```
if (remainingRate <= 0) {
```

```
    return 0;
}
```

Рассчитаем и вернём запас хода исходя из этого числа. Сначала разделим оставшийся уровень на уровень потребления и затем приведём к целому числу:

```
return (int) (remainingRate / getConsumptionRate());
```

Перейдём в класс Main. В нём уже написан код, который использует класс ElectricBus. В Main написано следующее: сначала создаём электробус с параметрами «потребление — одна десятая процента, минимальный уровень потребления — десять процентов»:

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

Заполняем бак до ста процентов:

```
bus.refuel(1);
```

Проверяем резерв и расходуем вначале на 50 км, потом на 900 км, потом ещё на 100 км:

```
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());
```

Уровень минимального разряда аккумулятора — 10%. Полного бака хватает на 900 км. Поэтому после расхода на 50 км проехать ещё 900 км не получится — резерва полного бака хватает не на 1000 км, а только на 900. Если после 50 км мы попытаемся проехать ещё 900, вылетим из цикла после проверки, метод run не сработает. Если бы аккумулятор можно было разрядить до нуля, возможно бы было проехать 1000 км.

Запустим код и посмотрим, как работает логика класса ElectricBus:

```
Резерв: 900
Едем 50 км: true
Резерв: 850
```

```
Едем 900 км: false
Резерв: 850
Едем 100 км: true
Резерв: 750
```

Вначале печатается резерв в 900 км, из него уже вычтен минимальный заряд в 10%. Электробус проезжает 50 км, остаётся резерв на 850 км. Проехать ещё 900 км не получится, а 100 — вполне возможно. Всё работает верно, согласно новой логике метода powerReserve().

Вы наверняка заметили символ @ и слово Override прямо перед созданным методом. Эта надпись называется аннотацией. Она показывает, что метод переопределён. Это указание необходимо для удобочитаемости и для документации JavaDoc. Если убрать эту аннотацию, ничего страшного не случится — только разработчик не сразу может понять, что этот метод переопределён. Посмотрим, что будет, если попробовать создать в этом классе новый метод, например геттер для новой переменной, и установить аннотацию @Override:

```
@Override
public double getMinimalTankFullnessRate(){
    return minimalTankFullnessRate;
}
```

Если дописать над методом @Override, то надпись подчеркнётся красным, что указывает на ошибку, которая произойдёт при компиляции кода. При попытке скомпилировать проект вы увидите ошибку:

```
java: method does not override or implement a method from a supertype
```

В сообщении написано, что этот метод не переопределяет метод из класса родителя.

Корректность аннотации может проверяться средой разработки. Аннотация @Override нужна для упрощения написания кода, чтобы видеть, какой метод переопределён, и быть уверенным, что в дочернем классе этот метод точно

соответствует родительскому. Ошибочный метод из этого класса можно удалить.

Наследование и переопределение — удобные инструменты, позволяющие расширять классы и делать программы лучше приспособленными к расширению функционала.

На практике иногда нужно запретить переопределение метода или даже наследование целого класса. Сделать это можно с помощью ключевого слова final, которое добавляется перед именем класса или метода, — такого же ключевого слова final, с помощью которого создаются неизменяемые переменные.

Запрет на переопределение методов или наследование классов нужен, чтобы гарантировать, что никто не сможет изменить принципы работы класса или метода помеченным ключевым словом final. Рассмотрим класс String (основной, самый часто используемый класс в Java) в качестве примера. Класс String выглядит следующим образом:

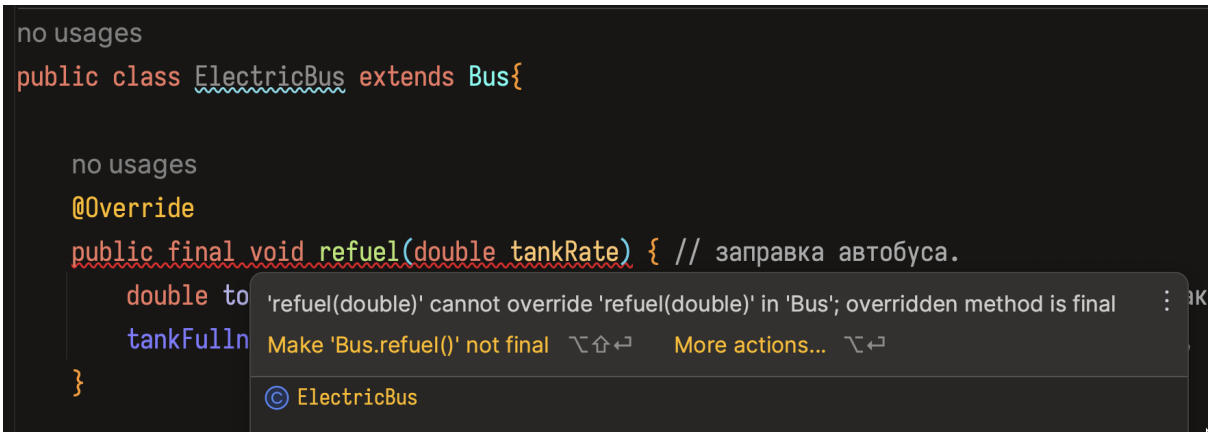
```
public final class String {
    // Код класса String
}
```

Ключевое слово final перед ключевым словом class указывает, что класс String не может быть наследован другими классами. Таким образом, нельзя создать подкласс от класса String и переопределить его методы. Это сделано, чтобы гарантировать, что принципы работы строк будут соблюдаться и класс String будет работать строго определённым образом, как ожидается.

Вернёмся к примеру с автобусами. В классе Bus есть метод refuel(). Допустим, вы не хотите, чтобы этот метод изменялся в классах-наследниках, поскольку это может нарушить всю логику работы этого класса. Запретим переопределение этого метода. Для этого добавим перед типом возвращаемого значения ключевое слово final:

```
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 больше, чем на 100%, бак заполнится
только до 100%.
}
```

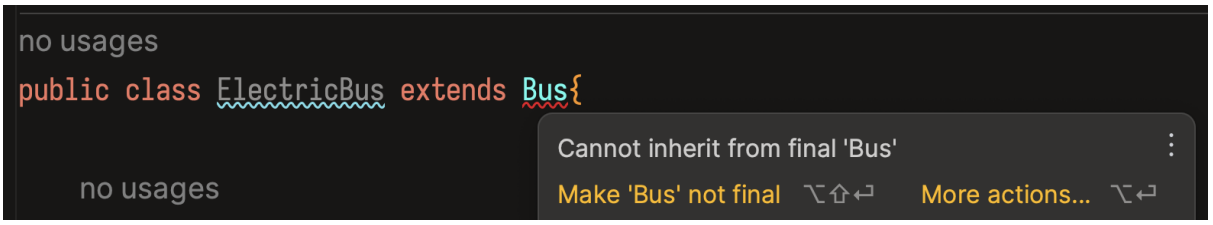
Попробуем переопределить метод в классе ElectricBus: скопируем метод refuel из класса Bus, вставим метод в класс ElectricBus. Метод в классе ElectricBus будет подчёркнут красным цветом, и появится сообщение, что метод не может быть переопределён, поскольку является финальным (помечен как final).



Ключевым словом final можно также запретить наследование класса целиком. Попробуем это сделать, написав перед именем класса слово final:

```
public final class Bus {
```

В классе ElectricBus теперь возникает ошибка: здесь написано, что нельзя наследовать от класса Bus, поскольку он помечен как final:



Вывод

Мы разобрали переопределение методов в классах-наследниках. Теперь вы знаете, что при наследовании классов методы родительского класса можно изменять в дочерних классах путём переопределения.

Также вы узнали, что с помощью ключевого слова final, применяемого для запрета изменения переменных, можно запрещать переопределение методов и наследование целых классов.