

## Инкапсуляция

00:00–01:42

Привет!

В этом видеоматериале мы познакомимся с одним из трёх принципов объектно-ориентированного программирования — инкапсуляцией.

**Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с этими данными, в одном классе и скрыть детали их реализации от пользователя.

То есть если, например, вы создаёте класс для корзины интернет-магазина, то все методы по работе с данными этой корзины вы должны реализовать в этом же классе в соответствии с принципом инкапсуляции.

Кроме того, инкапсуляция даёт возможность использовать методы, не влезая в их внутреннюю реализацию и фактически не думая о ней, и даёт уверенность, что переменные, которые мы защитили, используя принцип инкапсуляции, не будут изменяться неконтролируемо. Об этих двух аспектах мы и поговорим в этом видео.

Методы, как вы помните, — это фрагменты кода, которые названы какими-то именами, по которым их можно вызывать, и в которые можно передавать какие-то параметры. Также методы могут возвращать или не возвращать значение, а могут быть и без параметров.

Когда вы создаёте классы и методы, вы должны их создавать таким образом, чтобы они отражали предметную область, то есть соответствовали реальным объектам, их свойствам и поведению.

Опять же, если вы программируете функционал корзины для интернет-магазина, целесообразно создать соответствующий ей класс `Basket` и реализовать в нём методы управления этой корзиной — как минимум методы для добавления и удаления товаров.

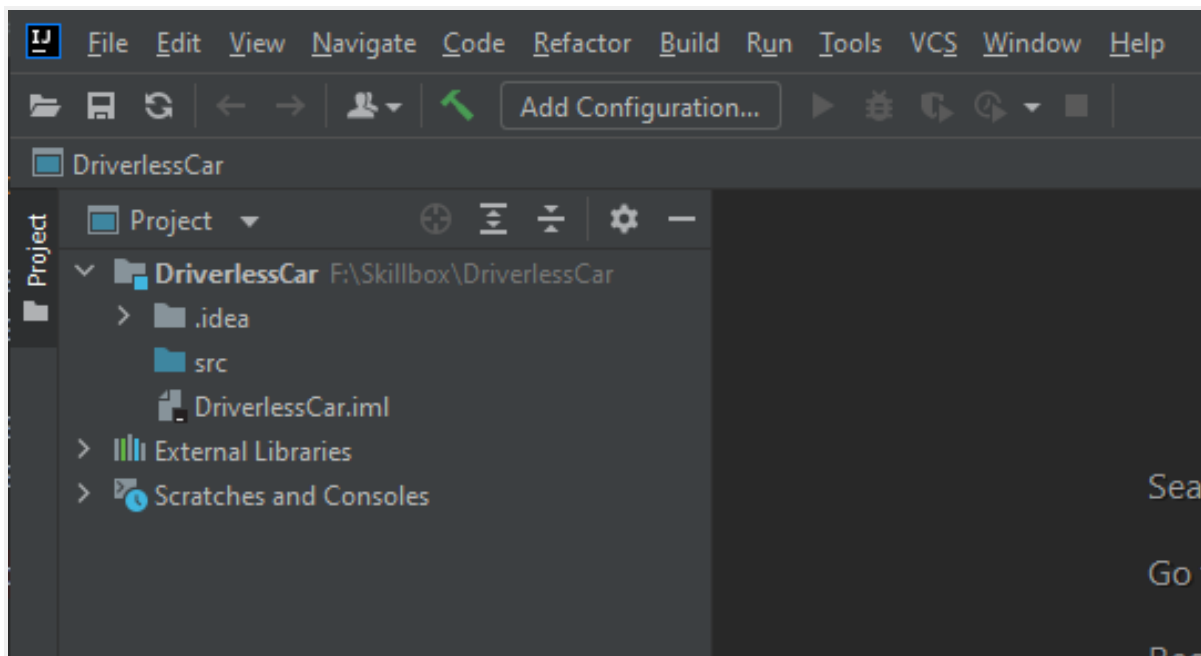
А ещё перед созданием методов нужно подумать над тем, чтобы они были понятны и могли в коде использоваться повторно, то есть переиспользоваться.

## Создание класса

**01:42–08:18**

Отдельный метод должен отражать какое-то поведение объекта. Давайте рассмотрим этот момент на практике. Представьте, что вам необходимо разработать систему управления беспилотным автомобилем — то есть автомобилем, который будет ездить без водителя.

Создаём проект `DriverlessCar`.



И прямо сейчас перед вами стоит задача спроектировать понятную и удобную систему управления коробкой передач.

Давайте создадим класс «Коробка передач» — по-английски GearBox.

```
public class GearBox {  
}
```

Что должен уметь этот класс? Во-первых, он должен уметь хранить своё состояние, то есть текущую передачу. Давайте будем хранить передачу в виде цифры, создадим переменную `int gear`.

```
public class GearBox {  
    public int gear;  
}
```

Давайте пропишем, какие значения может принимать эта переменная, укажем это в комментариях. Давайте условно обозначим нейтральную передачу числом 0, заднюю — числом -1 (задняя), и у нас ещё будет шесть основных передач — от 1 до 6. По умолчанию присвоим ей значение 0, то есть у нас будет нейтральная передача при создании этого объекта.

```
public class GearBox {  
    /**  
     * 0 - neutral  
     * -1 - rear  
     * 1-6  
     */  
    public int gear = 0;  
}
```

Теперь давайте подумаем, что должна уметь коробка передач. Во-первых, она должна уметь повышать передачу. Создадим соответствующий метод — `shiftUp` и в его коде пропишем повышение передачи.

```
public void shiftUp() {  
    gear = gear + 1;  
}
```

Но мы с вами знаем, и даже прописали это в комментариях, что есть пределы переключения скоростей — передача номер 6 является максимальной.

Для начала обозначим этот предел переменной, назовём её maxGear.

```
public int maxGear = 6;
```

Теперь в методе shiftUp будем проверять, что передача не увеличивается выше этого значения.

```
public void shiftUp() {  
    gear = gear < maxGear ? gear + 1 : gear;  
}
```

Прописав такой код, мы защищаем переменную gear от изменения выше заданного значения. Теперь значение этой переменной никогда не станет выше 6.

Теперь давайте реализуем метод, который будет понижать скорость. Назовём его shiftDown, он будет понижать передачу на одну ступень.

```
public void shiftDown() {  
    gear = gear - 1;  
}
```

По аналогии с предыдущим методом мы с вами понимаем, что здесь тоже нужно защитить переменную, но уже от чрезмерного понижения значения. В нашем случае скорость не может быть меньше первой.

Для этого создадим ещё одну переменную со значением минимальной передачи minGear...

```
public int minGear = 1;
```

...и проверим, чему равна текущая передача. Если она больше minGear, то мы можем её уменьшить, если нет, то мы уже не можем её уменьшить.

```
public void shiftDown() {  
    gear = gear > minGear ? gear - 1 : gear;  
}
```

То есть если вы будете двигать рычаг вниз и понижать скорость в реальной коробке передач, то меньше первой скорости не станет и на нейтральную передачу не переключится. И если вы не в курсе, то скорости в коробке передач переключаются по одной — нельзя одним движением переключить с 1-й на 5-ю или с 4-й на 1-ю.

Теперь давайте реализуем переключение на заднюю скорость. В реальной коробке передач переключить на заднюю можно только из нейтрального положения или с первой скорости. Для этого реализуем соответствующий метод.

```
public void switchRear() {  
    gear = gear > 1 ? gear : -1;  
}
```

Осталось реализовать переключение на нейтральную передачу. Её можно включить при любой передаче.

```
public void switchNeutral() {  
    gear = 0;  
}
```

Что нам ещё здесь может понадобиться? Например, получение значения текущей передачи. Создадим метод getCurrentGear.

```
public int getCurrentGear() {  
    return gear;  
}
```

Последнее, что следует сделать, чтобы защитить переменные `gear`, `minGear`, `maxGear` от изменения, — это сделать их **private**.

```
private int gear = 0;
private int minGear = 1;
private int maxGear = 6;
```

Таким образом, мы с вами реализовали методы управления переменной `gear`. В данном случае мы можем сказать, что класс `GearBox` инкапсулирует эту переменную. То есть фактически переменная `gear` защищена от возможности прямого изменения снаружи этого класса. Теперь нельзя взять и поменять её значение на произвольное.

Например, нельзя включить заднюю передачу, если вы сейчас на пятой. И нельзя перескочить через одну передачу — например, со второй на четвёртую. То есть мы реализовали логику работы нашей коробки передач, используя принцип инкапсуляции — сокрытия реализации и переменных от внешнего воздействия.

Кроме того, мы разместили все методы управления текущей передачей в одном единственном классе. С помощью инкапсуляции мы, по сути, регламентируем способы работы с нашей коробкой передач.

Используя инкапсуляцию, мы защищаем состояние объекта от неверных изменений и определяем правила работы с этим состоянием, предоставляя внешнему пользователю интерфейс, то есть строго определённые методы.

```
public class GearBox {

    /**
     * 0 - neutral
     * -1 - rear
     * 1-6
     */
```

```
private int gear = 0;
private int minGear = 1;
private int maxGear = 6;

public void shiftUp() {
    gear = gear < maxGear ? gear + 1 : gear;
}

public void shiftDown() {
    gear = gear > minGear ? gear - 1 : gear;
}

public void switchRear() {
    gear = gear > 1 ? gear : -1;
}

public void switchNeutral() {
    gear = 0;
}

public int getCurrentGear() {
    return gear;
}
}
```

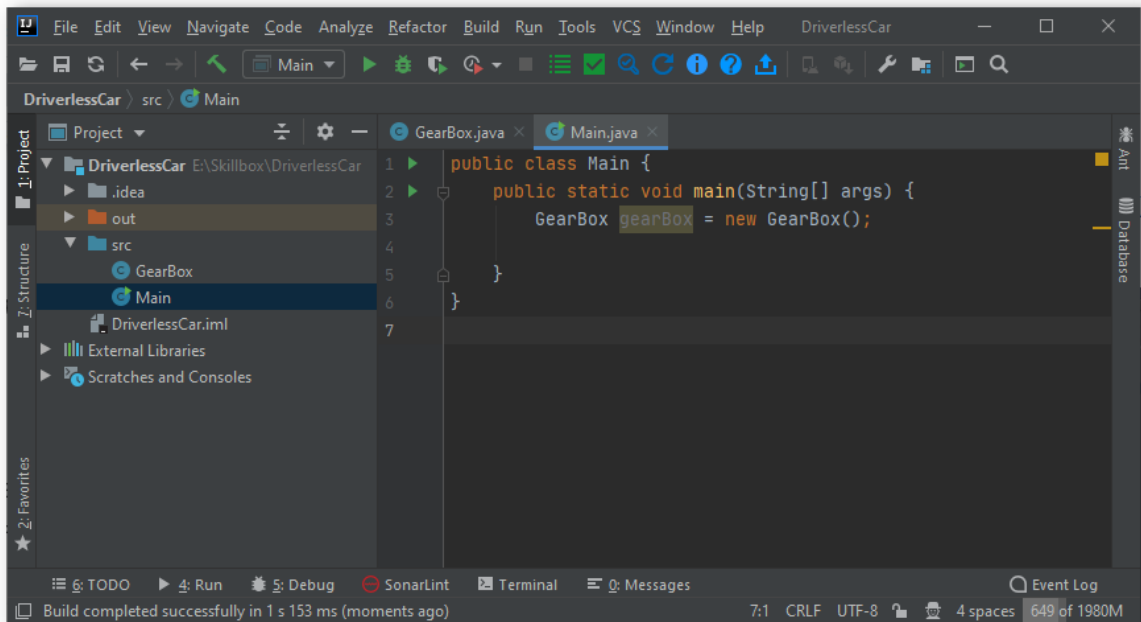
Инкапсуляция позволяет выходить на новый уровень программирования: пользоваться классами и объектами, не вникая в их реализацию. Нам не нужно думать, какая там переменная, какие там механизмы, как их контролировать, как контролировать состояние объекта и его правильное изменение.

Мы просто знаем, что можем делать с состоянием объекта по тем методам, которые у него есть, и больше ничего другого мы с ним делать не можем.

## Работа с объектами класса

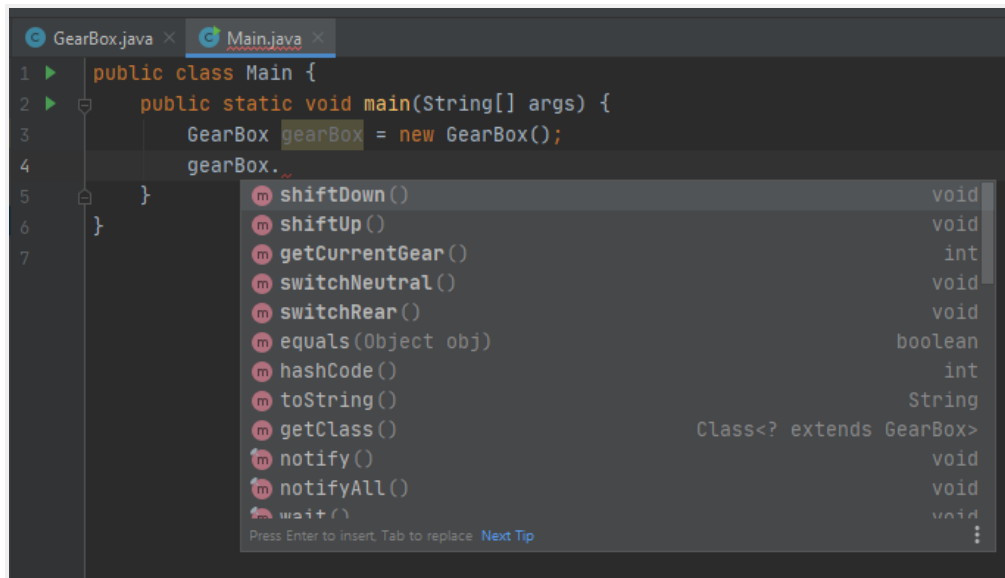
08:18–09:09

Давайте посмотрим на объект класса GearBox и разберёмся, что мы можем с ним сделать. Создадим класс с методом main и создадим в нём объект класса GearBox.



После имени переменной ставим точку и видим методы этого класса (выделены жирным), которые мы можем вызывать. Нам здесь не важна реализация, нам не важно, что внутри этих методов. Мы просто знаем, что мы можем делать с этим объектом.





```
1 public class Main {
2     public static void main(String[] args) {
3         GearBox gearBox = new GearBox();
4         gearBox.
5     }
6 }
7
```

- shiftDown() void
- shiftUp() void
- getCurrentGear() int
- switchNeutral() void
- switchRear() void
- equals(Object obj) boolean
- hashCode() int
- toString() String
- getClass() Class<? extends GearBox>
- notify() void
- notifyAll() void
- wait() void

Press Enter to insert, Tab to replace. [Next Tip](#)

При этом мы уверены, что эти методы контролируют состояние объекта правильно. Важно, чтобы методы назывались так, чтобы по их названию было однозначно понятно, что они делают с состоянием объекта.

## Итоги

**09:09 — до конца**

Итак, мы разобрали, что такое инкапсуляция. Инкапсуляция — это один из ключевых принципов объектно-ориентированного программирования, который заключается в сокрытии переменных (то есть состояния объекта) за методами, через которые этим состоянием можно управлять, а также в реализации этих методов только в этом же классе.

Она позволяет использовать методы классов, не вникая в их реализацию и не беспокоясь о состоянии объектов. Далее мы поговорим с вами о так называемых POJO-классах, а также о геттерах и сеттерах.

## Глоссарий

**Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с этими данными, в одном классе и скрыть детали их реализации от пользователя.