

## Статья

НАЧАТЬ

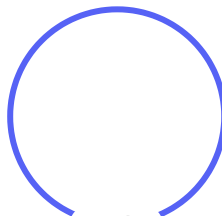
## Профессия Java-разработчик за 12 месяцев

Записывайся на новый курс от JavaRush!

Обучение  
в группеВидеоуроки с опытными  
преподавателямиПрактические  
домашние заданияДополнительные  
задачи и проекты

Поспеши: в группе осталось несколько мест.

ПОДРОБНЕЕ

[Статьи](#) [Авторы](#) [Все группы](#) [Все статьи](#)[JavaRush](#) / [Java блог](#) / [Java Developer](#) / Интерфейсы в Java

АВТОР

**Aditi Nawghare**Инженер-программист в  
**Siemens**

28 сентября 2018



355024



217

## Интерфейсы в Java

Статья из группы [Java Developer](#)

Присоединиться

Привет! Сегодня поговорим о важном понятии в Java — интерфейсы.

Слово тебе наверняка знакомо. Например, интерфейсы есть у большинства компьютерных программ и игр. В широком смысле интерфейс — некий «пульт», который связывает две взаимодействующие друг с другом стороны.

Простой пример интерфейса из повседневной жизни — пульт от телевизора.

Он связывает два объекта, человека и телевизор, и выполняет разные задачи: прибавить или убавить звук, переключить каналы, включить или выключить телевизор.

Одной стороне (человеку) нужно обратиться к интерфейсу (нажать на кнопку пульта), чтобы вторая сторона выполнила действие. Например, чтобы телевизор переключил канал на следующий. При этом пользователю не обязательно знать устройство телевизора и то, как внутри него реализован процесс смены канала.



Все, к чему пользователь имеет доступ — это *интерфейс*. Главная задача — получить нужный результат.

Какое это имеет отношение к программированию и Java? Прямое :)

Создание интерфейса очень похоже на создание обычного класса, только вместо слова `class` мы указываем слово `interface`.

Давай посмотрим на простейший Java-интерфейс, и разберемся, как он работает и для чего нужен:

```
1 public interface Swimmable {  
2  
3     public void swim();  
4 }
```

Мы создали интерфейс `Swimmable` — «*умеющий плавать*». Это что-то вроде нашего пульта, у которого есть одна «кнопка»: метод `swim()` — «плыть».

Как же нам этот «пульт» использовать?

Для этого метод, т.е. кнопку нашего пульта, нужно имплементировать. Чтобы использовать интерфейс, его методы должны реализовать какие-то классы нашей программы.

Давай придумаем класс, объекты которого подойдут под описание «умеющий плавать». Например, подойдет класс утки — `Duck`:

```
1 public class Duck implements Swimmable {
2
3     public void swim() {
4         System.out.println("Уточка, плыви!");
5     }
6
7     public static void main(String[] args) {
8
9         Duck duck = new Duck();
10        duck.swim();
11    }
12 }
```

Что же мы здесь видим?

Класс `Duck` «связывается» с интерфейсом `Swimmable` при помощи ключевого слова `implements`. Если помнишь, мы использовали похожий механизм для связи двух классов в наследовании, только там было слово «*extends*».

«`public class Duck implements Swimmable`» можно для понятности перевести дословно: «публичный класс `Duck` реализует интерфейс `Swimmable`».

Это значит, что класс, связанный с каким-то интерфейсом, должен реализовать все его методы. *Обрати внимание:* в нашем классе `Duck` прямо как в интерфейсе `Swimmable` есть метод `swim()`, и внутри него содержится какая-то логика.

Это обязательное требование. Если бы мы просто написали «`public class Duck implements Swimmable`» и не создали бы метод `swim()` в классе `Duck`, компилятор выдал бы нам ошибку:

*Duck is not abstract and does not override abstract method swim() in Swimmable*

Почему так происходит?

Если объяснять ошибку на примере с телевизором, получится, что мы даем человеку в руки пульт с кнопкой «переключить канал» от телевизора, который не умеет переключать каналы.

Тут уж нажимай на кнопку сколько влезет, ничего не заработает. Пульт сам по себе не переключает каналы: он только дает сигнал телевизору, внутри которого реализован сложный процесс смены канала.


Так и с нашей уткой: она должна уметь плавать, чтобы к ней можно было обратиться с помощью интерфейса `Swimmable`.


Если она этого не умеет, интерфейс `Swimmable` не свяжет две стороны — человека и программу. Человек не сможет использовать метод `swim()`, чтобы заставить объект `Duck` внутри программы плавать.


Теперь ты увидел более наглядно, для чего нужны интерфейсы.


Интерфейс описывает поведение, которым должны обладать классы, реализующие этот интерфейс. «Поведение» — это совокупность методов.


Если мы хотим создать несколько мессенджеров, проще всего сделать это, создав интерфейс `Messenger`. Что должен уметь любой мессенджер? В упрощенном виде, принимать и отправлять сообщения.


 **Получите профессию  
Java-разработчика**


на онлайн-курсе с ментором 

и сертификацией 


 Онлайн-лекции с опытными менторами

 Групповое обучение и поддержка в закрытом чате

 10 проектов в вашем портфолио и сотни часов кодинга

 Помощь в поиске первой работы

Скоро стартуют занятия в новой группе – поспешите!

Подробнее 

```
1 public interface Messenger{
2
3     public void sendMessage();
4
5     public void getMessage();
6 }
```

И теперь мы можем просто создавать наши классы-мессенджеры, имплементируя этот интерфейс. Компилятор сам «заставит» нас реализовать их внутри классов.

Telegram:

```
1 public class Telegram implements Messenger {
2
3     public void sendMessage() {
4
5         System.out.println("Отправляем сообщение в Telegram!");
6     }
7
8     public void getMessage() {
9         System.out.println("Читаем сообщение в Telegram!");
10    }
11 }
```

WhatsApp:

```
1 public class WhatsApp implements Messenger {
2
3     public void sendMessage() {
4
5         System.out.println("Отправляем сообщение в WhatsApp!");
6     }
7
8     public void getMessage() {
9         System.out.println("Читаем сообщение в WhatsApp!");
10    }
11 }
```

Viber:

```
1 public class Viber implements Messenger {
2
3     public void sendMessage() {
4
5         System.out.println("Отправляем сообщение в Viber!");
6     }
7
8     public void getMessage() {
9         System.out.println("Читаем сообщение в Viber!");
10    }
11 }
```

Какие преимущества это дает? Самое главное из них — слабая связанность.

Представь, что мы проектируем программу, в которой у нас будут собраны данные клиентов. В классе `Client` обязательно нужно поле, указывающее, каким именно мессенджером клиент пользуется.

Без интерфейсов это выглядело бы странно:

```
1 public class Client {
2
3     private WhatsApp whatsapp;
4     private Telegram telegram;
5     private Viber viber;
6 }
```

Мы создали три поля, но у клиента запросто может быть всего один мессенджер. Просто мы не знаем какой. И чтобы не остаться без связи с клиентом, приходится «заталкивать» в класс все возможные варианты. Получается, один или два из них всегда будут `null`, и они вообще не нужны для работы программы.

Вместо этого лучше использовать наш интерфейс:

```
1 public class Client {
2
3     private Messenger messenger;
4 }
```

Это и есть пример «слабой связанности»! Вместо того, чтобы указывать конкретный класс мессенджера в классе `Client`, мы просто упоминаем, что у клиента есть мессенджер. Какой именно — определится в ходе работы программы.

Но зачем нам для этого именно интерфейсы? Зачем их вообще добавили в язык?

Вопрос хороший и правильный! Того же результата можно добиться с помощью обычного наследования, так ведь?

Класс `Messenger` — родительский, а `Viber`, `Telegram` и `WhatsApp` — наследники. Действительно, можно и так.

Но есть одна загвоздка. Как ты уже знаешь, множественного наследования в Java нет. А вот множественная реализация интерфейсов — есть. Класс может реализовывать сколько угодно интерфейсов.

Представь, что у нас есть класс `Smartphone`, у которого есть поле `Application` — установленное на смартфоне приложение.

```
1 public class Smartphone {  
2  
3     private Application application;  
4 }
```

Приложение и мессенджер, конечно, похожи, но все-таки это разные вещи. Мессенджер может быть и мобильным, и десктопным, в то время как `Application` — это именно мобильное приложение.

Так вот, если бы мы использовали наследование, не смогли бы добавить объект `Telegram` в класс `Smartphone`. Ведь класс `Telegram` не может наследоваться одновременно от `Application` и от `Messenger`! А мы уже успели унаследовать его от `Messenger`, и в таком виде добавить в класс `Client`.

Но вот реализовать оба интерфейса класс `Telegram` запросто может! Поэтому в классе `Client` мы сможем внедрить объект `Telegram` как `Messenger`, а в класс `Smartphone` — как `Application`. Вот как это делается:

```
1 public class Telegram implements Application, Messenger {
2
3     //...методы
4 }
5
6 public class Client {
7
8     private Messenger messenger;
9
10    public Client() {
11        this.messenger = new Telegram();
12    }
13 }
14
15
16 public class Smartphone {
17
18     private Application application;
19
20    public Smartphone() {
21        this.application = new Telegram();
22    }
23 }
```

Теперь мы используем класс `Telegram` как захотим. Где-то он будет выступать в роли `Application`, где-то — в роли `Messenger`.

Наверняка ты уже обратил внимание, что методы в интерфейсах всегда «пустые», то есть они не имеют реализации.

Причина этого проста: интерфейс описывает поведение, а не реализует его.

«Все объекты классов, имплементирующих интерфейс `Swimmable`, должны уметь плавать»: вот и все, что говорит нам интерфейс. Как там конкретно будет плавать рыба, утка или лошадь — вопрос к классам `Fish`, `Duck` и `Horse`, а не к интерфейсу. Также как переключение канала — задача телевизора. Пульт просто предоставляет тебе кнопку для этого.

Впрочем, в Java8 появилось интересное дополнение — методы по умолчанию (default method).



Например, в твоём интерфейсе есть 10 методов. 9 из них реализованы по-разному в разных классах, но один реализован одинаково у всех. Раньше, до выхода Java8, методы внутри интерфейсов вообще не имели реализации: компилятор сразу выдавал ошибку. Теперь же можно сделать вот так:

```
1 public interface Swimmable {
2
3     public default void swim() {
4         System.out.println("Плыви!");
5     }
6
7     public void eat();
8
9     public void run();
10 }
```

Используя ключевое слово `default`, мы создали в интерфейсе метод с реализацией по умолчанию. Два других метода, `eat()` и `run()`, нам необходимо будет реализовать самим во всех классах, которые будут имплементировать `Swimmable`. С методом `swim()` этого делать не нужно: реализация будет во всех классах одинаковой.

Кстати, ты уже не раз сталкивался с интерфейсами в прошлых задачах, хоть и не замечал этого сам :) Вот очевидный пример:



Для чего в Java нужны интерфейсы - 2

Ты работал с интерфейсами `List` и `Set`! Точнее, с их реализациями — `ArrayList`, `LinkedList`, `HashSet` и прочими.

На этой же схеме видно пример, когда один класс реализует сразу несколько интерфейсов. Например, `LinkedList` реализует интерфейсы `List` и `Deque` (двусторонняя очередь).

Ты знаком и с интерфейсом `Map`, а точнее, с его реализацией — `HashMap`.

Кстати, на этой схеме ты можешь увидеть одну особенность: интерфейсы могут быть

унаследованы друг от друга. Интерфейс `SortedMap` унаследован от `Map`, а `Deque` наследуется от очереди `Queue`. Это нужно, если ты хочешь показать связь интерфейсов между собой, но при этом один интерфейс является расширенной версией другого.

Давай рассмотрим пример с интерфейсом `Queue` — очередь. Мы пока не проходили коллекции `Queue`, но они достаточно простые и устроены как обычная очередь в магазине.

Добавлять элементы можно только в конец очереди, а забирать — только из начала. На определенном этапе разработчикам понадобился расширенный вариант очереди, чтобы добавлять и получать элементы можно было с обеих сторон. Так создали интерфейс `Deque` — двустороннюю очередь. В нем присутствуют все методы обычной очереди, ведь она является «родителем» двусторонней, но при этом добавлены новые методы.

## Aditi Nawghare

Инженер-программист в Siemens

Инженер-программист в Siemens с опытом работы в проектах по анализу данных и разработке программного обеспечения. Она сертифициров ...

[\[Читать полную биографию\]](#)

### Комментарии (217)

ЧТОБЫ ПОСМОТРЕТЬ ВСЕ КОММЕНТАРИИ ИЛИ ОСТАВИТЬ КОММЕНТАРИЙ,