# Различия между абстрактными классами и интерфейсами в Java

[Введение](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#0)  
[1. Фундаментальное отличие](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#1)  
[2. Технические отличия](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#2)  
[2.1. Синтаксис создания](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#3)  
[2.2 Синтаксис использования](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#4)  
[2.3 Наличие конструктора](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#5)  
[2.4 Типы переменных](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#6)  
[2.5 Модификаторы доступа методов](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#7)  
[2.6 Методы с реализацией](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#8)  
[2.7 Наследование](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#9)  
[3. Рекомендации к применению абстрактных классов и интерфейсов](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#10)  
[4. Примеры реализации абстрактных классов и интерфейсов в JDK](https://topjava.ru/blog/razlichiya-mezhdu-abstraktnymi-klassami-i-interfeysami-v-java#11)

**Введение**

В процессе профессионального развития каждого начинающего программиста всегда наступает тот момент, когда требуется понять и осознать разницу между [интерфейсом](https://topjava.ru/blog/istoriya-evolutsii-interfeysov-v-java) и абстрактным классом. Ясно представлять себе границы их использования, а также уметь выбирать, какой из этих способов лучше применить в том или ином месте кода.

Кроме того, обсуждаемый в данной статье вопрос о различиях, является одним из самых распространенных на собеседованиях, на который зачастую отвечают либо неполно, либо неверно.

Почему же на этот вопрос отвечают неправильно? Дело в том, что Java развивается, и границы отличий между абстрактным классом и интерфейсом постепенно стираются. А новички, читая устаревшие книги по Java (или не читая их вовсе) или даже современные статьи-перепечатки, где рассказывается об интерфейсах до Java 8, не имеют представления о том, что в своем развитии они ушли далеко вперед.

Например, до появления Java 8, при возникновении вопроса на собеседовании типа: «Назовите разницу между абстрактным классом и интерфейсом», вы смело могли сказать, что интерфейс представляет собой контракт, содержащий методы без реализации, которые должны быть реализованы в классах, имплементирующих этот интерфейс. А абстрактный класс — это класс, который содержит (или не содержит) абстрактные методы и общий для его потомков код.

Также, раньше вы могли сказать, что интерфейс не может содержать статические, приватные или методы по умолчанию, и что в нем вообще не может быть реализовано никакой логики. В техническом плане все ограничения остались в прошлом — граница между этими понятиями сейчас уже почти стерта. Но с фундаментальной точки зрения ничего не изменилось!

В данной статье будут рассмотрены как фундаментальные, так и технические различия между интерфейсом и абстрактным классом.

**1. Фундаментальное отличие**

Фундаментальная разница между интерфейсом и абстрактным классом заключается в том, что интерфейс определяет только поведение. Он не сообщает ничего про объект, который будет его реализовывать.

Например, такое поведение как «движение» может быть применимо к разным типам объектов: машина, кот, котировки курса и т. д. Эти объекты не имеют ничего общего, кроме того, что они могут двигаться.

Перефразируем немного иначе — если существует «движущийся» объект, то глядя на реализованный им интерфейс, мы не сможем понять, какой именно тип объекта имеется ввиду. Это может быть машина, кот, котировка курса валют и много других вариантов.

Если перенести это поведение в код, написанный на языке Java, то получится следующее:

**interface** **Movable** {

**void** **move**();

}

Абстрактный же класс описывает некий абстрактный объект (автомобиль, человека, кота и т. д.), а не только поведение.

Если мы рассуждаем про абстрактный «автомобиль», то в нашей голове сразу формируется картинка с объектом. Мы сразу понимаем, что автомобиль содержит двигатель, колеса и может двигаться, поворачивать, ускоряться или тормозить. Он также будет иметь поля для хранения внутренних деталей, таких как мотор, колеса и тормоза. Вы получите все это просто сказав слово «автомобиль».

При этом кот и котировка курса валют не могут быть автомобилем, даже если они также могут двигаться.

На основе вышесказанного создадим класс Automobile:

**abstract** **class** **Automobile** {

Engine engine;

Wheels[] wheels;

Gears[] gears;

**abstract** **void** **move**();

**abstract** **void** **turn**();

**abstract** **void** **accelerate**();

**abstract** **void** **brake**();

}

Интерфейс и абстрактный класс — понятия не взаимозаменяемые. Даже если абстрактный класс с абстрактными методами выглядит подобно интерфейсу, а интерфейс, с его методами по умолчанию, подобен абстрактному классу с методами, имеющими реализации, то это все равно будут два фундаментально разных понятия. Если нам нужно поведение — необходимо использовать интерфейс. Если речь про концептуальный объект — мы должны использовать абстрактный класс.

А теперь поговорим про различия в технической реализации.

**2. Технические отличия**

В этом разделе мы обсудим технические (синтаксические) различия в реализации этих концепций в Java. Именно так обычно отвечают на вопрос о разнице между интерфейсами и абстрактными классами (но мы то уже знаем, что это только вторая часть ответа, и при этом не главная).

**2.1. Синтаксис создания**

Итак, при создании абстрактного класса указывается ключевое слово abstract, а при определении интерфейса — interface.

Пример абстрактного класса:

**public** **abstract** **class** **MyAbstractClass** {

*// поля и конструкторы*

*// абстрактные методы*

*// методы с реализацией*

}

Пример интерфейса:

**public** **interface** **MyInterface** {

*// объявление констант*

*// методы без реализации*

*// статические методы*

*// методы по умолчанию (default)*

*// приватные методы*

}

**2.2 Синтаксис использования**

При наследовании от абстрактного класса используется ключевое слово extends (с англ. «расширяет»), а при реализации интерфейса — implements (с англ. «реализует»).

В приведенном ниже коде класс MyClass расширяет MyAbstractClass:

**public** **class** **MyClass** **extends** **MyAbstractClass** {

*// реализация абстрактных методов*

*// иной код*

}

А тут класс MyClass реализует интерфейс MyInterface:

**public** **class** **MyClass** **implements** **MyInterface** {

*// реализация методов интерфейса*

*// иной код*

}

Класс может одновременно и наследоваться от абстрактного класса (только одного) и реализовать один или множество интерфейсов.

Например, класс MyClass реализует интерфейс MyInterface, MyInterface2 и MyInterface3:

**class** **MyClass** **extends** **MyAbstractClass** **implements** **MyInterface**,

**MyInterface\_2**, **MyInterface\_3** {

*// реализация абстрактных методов абстрактного класса*

*// реализация методов из интерфейсов*

*// иной код*

}

**2.3 Наличие конструктора**

Как вы знаете, невозможно создать экземпляр абстрактного класса. Но, объявить и определить в нем конструктор мы можем. В противном случае за нас это сделает компилятор, создав конструктор по умолчанию. Без него код просто не скомпилируется, поскольку при создании конкретного класса первым оператором будет неявный вызов super() [конструктора](https://topjava.ru/blog/rukovodstvo-po-konstruktoram-v-java) суперкласса, в данном случае абстрактного.

Добавим в абстрактный класс MyAbstractClass конструктор:

**public** **abstract** **class** **MyAbstractClass** {

**public** **MyAbstractClass**() {

System.out.println("Конструктор из MyAbstractClass");

}

}

Также, добавим конструктор в MyClass:

**public** **class** **MyClass** **extends** **MyAbstractClass** {

**public** **MyClass**() {

System.out.println("Конструктор из MyClass");

}

}

В классе Main создадим объект и запустим программу:

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

MyAbstractClass myObject = **new** MyClass();

}

}

Результат выполнения программы:

Для интерфейсов понятия «конструктор» не существует.

**2.4 Типы переменных**

Все переменные в интерфейсах неявно являются public static final (т.е. константами). «final» подразумевает, что переменной обязательно должно быть присвоено значение во время инициализации.

Рассмотрим следующий код:

**public** **interface** **MyInterface** {

*// строка не компилируется*

**int** value\_1;

**int** value\_2 = 1;

**public** **final** **int** value\_3 = 1;

**static** **int** value\_4 = 1;

**public** **final** **static** **int** value\_5 = 1;

**static** **final** **int** value\_6 = 1;

}

Поскольку value\_1 не присвоено конкретное значение, а она является неявно final, код с такой строкой не скомпилируется. Остальные строки не вызовут ошибок, т.к. public static final можно не указывать (если их указать, то IDE выделит их серым цветом, подчеркивая «избыточность»).

В абстрактных классах переменные могут быть любыми — абстрактность класса не накладывает ограничений.

Например, следующий код является корректным:

**public** **abstract** **class** **MyAbstractClass** {

**int** value\_1;

**int** value\_2 = 1;

**private** **static** **int** value\_3;

**final** **int** value\_4 = 1;

**protected** **static** **final** **int** value\_5 = 1;

}

**2.5 Модификаторы доступа методов**

Модификаторы доступа для абстрактных классов могут быть любыми. При этом, все методы кроме абстрактных должны иметь реализацию.

А вот в случае с интерфейсами модификаторы доступа могут быть только двух типов, public и private (последний, начиная с Java 9).

Причем модификатор доступа private может использоваться только с методами, имеющими реализацию. Такие методы могут использоваться только методами по умолчанию, находящимися в интерфейсе. Класс, реализующий интерфейс, не будет иметь к ним доступ. Методы интерфейса без реализации являются неявно public, поэтому этот модификатор доступа можно не писать.

Рассмотрим следующий код:

**public** **interface** **MyInterface2** {

**void** **publicAbstractMethod\_1**();

**public** **void** **publicAbstractMethod\_2**();

**void** **publicAbstractMethod\_3**();

**private** **void** **privateMethod**() {

*// Реализация метода*

}

}

Методы publicAbstractMethod\_1 и publicAbstractMethod\_3 являются неявно public.

**2.6 Методы с реализацией**

Для абстрактного класса все методы кроме абстрактных должны иметь реализацию.

Для интерфейсов, начиная с Java 8, вводится понятие метода по умолчанию. Такие методы, во-первых, должны иметь реализацию в интерфейсе, а во-вторых, быть помечены ключевым словом default. Они также являются неявно public. При этом они не должны в обязательном порядке иметь реализацию в реализующем интерфейс классе, но могут быть в нем переопределены.

Также, начиная с Java 8 в интерфейсах допустимы статические методы, которые неявно являются public, но могут быть явно private. Т. е. если указать модификатор доступа private, метод будет приватным, а если ничего не указывать, то публичным.

Например, код ниже скомпилируется:

**public** **interface** **MyInterface3** {

**default** **void** **defaultMethod**() {

*// Реализация метода*

}

**public** **default** **void** **defaultMethod\_2**() {

*// Реализация метода*

}

**private** **static** **void** **privateStaticMethod**() {

*// Реализация метода*

}

**public** **static** **void** **publicStaticMethod\_1**() {

*// Реализация метода*

}

**static** **void** **publicStaticMethod\_2**() {

*// Реализация метода*

}

}

**2.7 Наследование**

Интерфейс не может реализовывать интерфейс, не может наследовать абстрактный класс, но может наследовать (используя ключевое слово extends) множество других интерфейсов.

Абстрактный класс может наследовать как обычный класс, так и абстрактный. В обоих случаях это будет только один класс (в Java нет множественного наследования классов).

В то же время абстрактный класс также может реализовать до 65 535 интерфейсов (это связано с ограничением константы interfaces\_count в структуре ClassFile).

В этом месте давайте немного отойдем от темы сравнения и затронем множественное наследование. В некоторых источниках однозначно указывается, что множественного наследования в Java нет. В других источниках говорится, что множественного наследования как бы нет, но оно реализуется через интерфейсы.

Обратимся за разъяснением к официальной [документации](https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html). В ней говорится о трех типах множественного наследования:

* наследование состояния (Inheritance of State)
* наследование реализации (Inheritance of Implementation)
* наследование типа (Inheritance of Type)

Java не поддерживает множественное наследование состояния, но поддерживает множественное наследование реализации (как вариант, на основе методов по умолчанию от интерфейсов). Она также поддерживает множественное наследование типов (поскольку может реализовать более одного интерфейса).

Именно таким должен быть ответ на вопрос о множественном наследовании в Java.

**3. Рекомендации к применению абстрактных классов и интерфейсов**

В соответствии с [рекомендациями](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) Oracle абстрактный класс нужно использовать в следующих случаях:

* Необходимо выделить общий код между несколькими тесно связанными классами  
  Пояснение: это типовой рефакторинг, целью которого является устранение дублирования кода.
* Мы ожидаем, что классы, расширяющие абстрактный класс, имеют много общих методов, полей или требуют модификаторов доступа, отличных от public (protected и private)  
  Пояснение: ранее мы писали, что в интерфейсах методы, имеющие реализацию (помеченные ключевым словом default), являются неявно public. Если же метод, имеющий реализацию, помечен явно как private, то он не сможет быть использован в классах, реализующих этот интерфейс, а только в других методах интерфейса. Поэтому, если нам нужны методы с модификаторами доступа не public, мы должны использовать абстрактный класс.
* Мы хотим объявить не static или не final поля для изменения состояния объекта.  
  Пояснение: ранее мы писали, что все переменные в интерфейсах неявно являются public static final и из-за этого не могут быть изменены.

Для использования интерфейсов существуют следующие причины:

* Планируется, что несвязанные между собой классы будут реализовывать интерфейс. Например, интерфейсы Comparable и Cloneable реализуются многими несвязанными между собой классами.  
  Пояснение: как мы выяснили ранее, концептуальное назначение интерфейса — описание "поведения", а не "состояния" (в отличие от абстрактного класса). Соответственно, реализовать это поведение могут любые классы, несвязанные между собой, которые просто должны что-то "уметь" делать или "как-то" себя вести.
* Требуется детализировать или определить поведение определенного типа данных, но при этом мы не хотим беспокоиться о том, кто реализует его поведение.  
  Пояснение: пояснение к прошлому пункту также актуально и для этого — нам все равно, какие классы будут реализовывать наш интерфейс и в каких пакетах находятся. Мы просто хотим, чтобы они вели себя в соответствии с "нашим контрактом", прописанным в интерфейсе в виде методов без реализации.
* Мы хотим воспользоваться преимуществами множественного наследования типов

Добавим наглядный пример выбора между абстрактным классом и интерфейсом (подробности читайте в статье, посвященной [паттерну Шаблонный метод](https://topjava.ru/blog/pattern-shablonnyy-metod-v-java)).

Существует паттерн «Шаблонный метод», определение которого звучит так: «шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые его шаги, не изменяя структуру в целом».

Суть паттерна заключается в размещении в абстрактном классе метода с реализацией, в которой вызываются абстрактные методы. Далее, подклассы абстрактного класса реализуют их каждый по своему.

Так вот вопрос, а что нам мешает разместить абстрактные и шаблонные методы в интерфейсе? В качестве шаблонного метода мы могли бы использовать метод по умолчанию. Вроде, звучит логично, согласитесь?

Но, обратите еще раз внимание на определение шаблонного метода. Там есть такие слова: «… не изменяя структуру в целом». Подклассы абстрактного класса не должны иметь возможность переопределить шаблонный метод с целью изменения логики (алгоритма) его работы. По этой причине шаблонные методы объявляются как final и не могут использовать интерфейсы с их методами по умолчанию (могут быть переопределены) вместо абстрактных классов. Вся идея паттерна в этом случае будет нарушена.

Для того чтобы рассмотреть разницу между интерфейсом и абстрактным классом немного глубже, рекомендуется обратиться к книге «Java. Эффективное программирование», Блох Д., 3-е издание.

В разделе 4.6 этой книги говорится о следующих причинах предпочтения интерфейсов абстрактным классам:

1. Существующие классы можно легко приспособить для реализации но­вого интерфейса.  
Пояснение: для этого достаточно написать implements и реализовать необходимые методы. Но уже имеющиеся классы в общем случае не могут быть переделаны для расширения нового абстрактного класса. Если вы хотите, чтобы два класса расширяли один и тот же абстрактный класс, вам придется поднять этот класс в иерархии настолько высоко, чтобы он стал предком обоих этих классов. Это может привести к нарушению логики иерархии типов, заставляя всех потомков нового абстрактного класса расширять его независимо от того, насколько это целесообразно.  
  
2. Интерфейсы идеально подходят для создания миксинов.  
Пояснение: Миксин (mixin) — это тип, который класс может реализовать в дополнение к своему "первичному типу", объявляя о том, что этот класс предоставляет некоторое необязательное поведение. Например, Comparable является таким интерфейсом-миксином, т.к. добавляет (примешивает) к первоначальным возможностям типа дополнительную функциональность.  
Использовать абстракт­ные классы для создания миксинов нельзя по той же причине, по которой их невозможно приспособить к уже имеющимся классам: класс не может иметь больше одного родителя, и в иерархии классов нет подходящего места, куда можно поместить миксин.  
  
3. Интерфейсы позволяют создавать неиерархические каркасы типов.  
Пояснение: иерархии типов прекрасно подходят для организации некоторых сущностей, но зато сущности других типов невозможно аккуратно уложить в строгую ие­рархию. Альтернативой им является раздутая иерархия классов.  
  
4. Интерфейсы обеспечивают безопасное и мощное развитие функци­ональности с использованием шаблона Декоратор. Паттерн Декоратор позволяет динамически (в ходе выполнения программы) добавлять объекту новые возможности (состояние и/или поведение) на основе композиции.

**4. Примеры реализации абстрактных классов и интерфейсов в JDK**

Примером абстрактного класса в JDK является AbstractMap. Его подклассы (HashMap, TreeMap и ConcurrentHashMap) имеют общие методы (get, put, isEmpty, containsKey и containsValue), определенных в нем.

Примером класса в JDK, реализующего несколько интерфейсов, является HashMap, который реализует интерфейсы Serializable, Cloneable и Map<K, V>. Читая этот список интерфейсов, вы можете сделать вывод, что экземпляр HashMap (независимо от разработчика или компании, реализующей класс) может быть клонирован, сериализуем и имеет функциональные возможности структуры данных Map. Кроме того, интерфейс Map<K, V> был улучшен методами по умолчанию, такими как merge и forEach, которые не нужно определять классам, реализующими этот интерфейс.

Обратите внимание, что многие программные библиотеки используют как абстрактные классы, так и интерфейсы. Например, класс HashMap реализует несколько интерфейсов, а также расширяет абстрактный класс AbstractMap.