# Лекция 8. Интерфейсы в Java. Перегрузка и переопределение методов

# Интерфейсы в Java

Интерфейс - [синтаксическая](https://ru.wikipedia.org/wiki/Синтаксис_(программирование)) конструкция в коде программы, используемая для [декларации](https://ru.wikipedia.org/wiki/Спецификация) услуг, предоставляемых [классом](https://ru.wikipedia.org/wiki/Класс_(программирование)) или [компонентом](https://ru.wikipedia.org/wiki/Компонент_(программирование)).

Интерфейс определяет границу взаимодействия между классами, специфицируя [абстракцию](https://ru.wikipedia.org/wiki/Абстракция_данных), реализация которой осуществляется компонентом, исполняющим контракт интерфейса.

Интерфейс – это инкапсуляция с «действии»: публичный контракт (методы интерфейса) является открытым, тогда как реализация – сокрыта в тех классах, которые реализуют этот интерфейс. Можно считать интерфейсы чисто абстрактными классами, т.е. классы, не содержащие полей и состоящие только из абстрактных методов.

В Java один и тот же интерфейс может быть реализован разными классами, включая абстрактные. Более того, любой класс может реализовывать несоклько интерфейсов. В формуле 1 представлено объявление интерфейса в Java.

|  |
| --- |
| [модификатор\_доступа] interface Имя\_интерфейса  [extends Список\_родительских\_интерфейсов]  {  // декларация методов интерфейса  } |
| Формула 1. Объявление интерфейса в Java |

|  |
| --- |
| **public** **interface** SomeInterface  {  //методы интерфейса  } |
| Пример 1. Декларация интерфейса |
| **public** **interface** SuperSomeInterface  {  // родительский интерфейс к SomeInterface  }  **public** **interface** SomeInterface **extends** SuperSomeInterface  {  // методы интерфейса  } |
| Пример 2. Наследование интерфейсов |

Существует несколько правил при описании методов интерфейсов (см. формулу 2):

* Методы интерфейса являются абстракными, однако, ключевое слово **abstract** указывать не нужно.
* Методы интерфейса являются **public** по умолчанию, поэтому данный модификатор можно так же опустить.

|  |
| --- |
| [модификатор\_доступа] interface Имя\_интерфейса  [extends Список\_родительских\_интерфейсов]  {  public abstract тип\_возвращаемого\_значения имя\_метода ([список\_параметров]);  public тип\_возвращаемого\_значения имя\_метода ([список\_параметров]);  тип\_возвращаемого\_значения имя\_метода ([список\_параметров]);  } |
| Формула 2. Способы декларации мотодов интерфейса в Java |

Наследование интерфейсов приводит к тому, что тот класс, который реализует интерфейс, реализует методы своего интерфейса, и все методы всех суперинтерфейсов данного интерфейса. Реализация интрфейса происходит с помощью ключевого слова «implemets» (см. формулу 3).

|  |
| --- |
| [модификатор\_доступа] Имя\_класса **implemets**  [Список\_интерфейсов]  {} |
| Формула 2. Способы декларации методов интерфейса в Java |
| **public** **interface** SuperSomeInterface  {  **void** methodOfSuperSomeInterface();  }  **public** **interface** SomeInterface  {  **void** methodOfSomeInterface();  }  **public** **class** SomeClass **implements** SomeInterface  {  @Override **public** **void** methodOfSuperSomeInterface()  {  // унаследованный интерфейсом SomeInterface метод родителя SuperSomeInterface  }  @Override **public** **void** methodOfSomeInterface()  {  // метод интерфейса SomeInterface  }  } |
| Пример 3. Реализация интерфейса классом |

# Приведение ссылочных типов в Java

В силу полиморфизма, любой класс, реализующий некий интерфейс фактически приводится к типу этого интерфейса и типам всех его родителей. Данное правило справедливо и к наследованную классов: любой класс приводим ко всем своим суперклассам. Это связано именно с понятием динамического и статического связывания типов.

На этапе статического связывания компилятор проверяет, что тип переменной соответствует типу значения, присвоенного ей при инициализации (см. пример 4). Фактически это проверка ссылок на соответствие типов.

|  |
| --- |
| **public** **class** Executor  {  **public** **static** **void** main(String[] args)  {  SuperSomeInterface ssi=**new** SomeClass(); //строка 1  SomeInterface si = (SomeInterface) ssi; // строка 2  SomeInterface siObj = (SomeInterface) new Object(); // строка 3  }  } |
| Пример 4. Статическое связывание |

В примере 4, в строке 1, перменная ssi типа SuperSomeInterface инициализируется значением типа SomeClass, что справедливо в силу того, что класс SomeClass реализует данный интерфейс, а значит приводим к его типу. Так как тип SuperSomeInterface – наиболее общий, то операция приведения (cast) не нужна. Действует правило расширения типа: наиболее общему типа можно приравнять значение наиболее частного типа.

В строке 2, однако, этого сделать нельзя: тип SomeInterface хоть и является наследником SuperSomeInterface, но SuperSomeInterface ничего не знает о своих потомках, именно поэтому присвоение без явного приведения (cast) невозможно. Более того, на этапе статического связывания, компилятор не проверяет, что в ssi реально лежит ссылка на объект типа SomeClass, который реализует интерфейс SomeInterface, а лишь проверяет, что переменная типа SuperSomeInterface совместима с типом ссылки si. Почему это может быть не так? Потому что тип ссылки ssi шире, чем типа si, а значит, есть вероятность, что реализацией данного типа может быть другой класс или интерфейс (не SomeInterface или SomeClass), что приведет к ошибке типа ClassCastException.

Таким образом, при статическом связывании проверяется только совместимость типов.

А при динамическом связывании проверяются реальные типы. Именно на этапе динамического связывания, в runtime, Java-машина будет различать, на объект какого реального типа ссылается ссылка его супертипа, и совместимы ли эти типы.

Так, в строке 3, на этапе компиляции ошибки не возникнет: явным приведением, ссылка на объект типа Object была приведена к типу SomeInterface, а значит присвоение справделиво, типы совместимы. Однако, в runtime, возникает ошибка (см. рис.1). Связано это с тем, что типы Object и SomeInterface не являются родственными.

|  |
| --- |
|  |
| Рис.1. Ошибка приведения типов (ClassCastException) при динамическом связывании |

# Перегрузка (overload) методов класса

Возникают ситуации, когда методы фактически делают одно и тоже (и по замыслу, и по бизнес-логике). Соотвественно, возникает ситуация, при которой необходимо описать несколько методов с одним и тем же именем. Однако, если декларировать несколько методов с одним и тем же именем, возникнет ситуация, когда компилятор сообщит об ошибке компиляции, так как в классе не может быть двух методов с одинаковой декларацией.

Даже, если, чисто теоритически, представить, что такой код будет скомпилирован, то Java-машина, в момент исполнения такого кода, не сможет принять решение о том, код какого метода должен быть вополнен. Чтобы принять это решение, Java-машина анализирует ***сигнатуру*** метода.

***Сигнатура метода*** – это совокупность имени метода и списка типов его аргументов. Тип возвращаемого значения и модификатор доступа в сигнатуру метода не входят.

Следовательно, исходя из типов аргементов, передаваемых методу, Java-машина примет решение о том, какой метод должет быть выполен.

|  |
| --- |
| **public** **class** SomeClass  {  **public** **void** someMethod() //1  {  // метод someMethod без параметров  }  **private** **void** someMethod() //2: ошибка компиляции  {  // метод someMethod с параметром типа int  }  **public** **void** someMethod(**int** i) //3  {  // метод someMethod с параметром типа int  }    **public** **int** someMethod(**int** i) //4: ошибка компиляции  {  // метод someMethod с параметром типа int  **return** **0**;  }  } |
| Пример 5. Перегрузка методов |

В примере 5 приведена перегрузка метода someMethod в классе SomeClass. Он перегружен 4 раза, и только 2 раза – правильно. Пара методов 1 и 2 ошибочна: с точки зрения Java – это декларация одного и того же мотода, несмотря на то, что один метод является public, а второй – private. Оба метода имеют одинаковую сигнатуру (имя и отсутствуюший список аргументов), а значит – это нарушение правил перегрузки методов. Вторая пара (3 и 4) так же имеет одинаковую сигнатура (имя и один параметр типа int), несмотря на то, что метод 4 имеет возвращаемое значение.

Соотвественно, методы 1 и 3 являются правильно перегруженными, так как имеют разную сигнатуру (разный список входных параметров), хоть и имеют одинаковый тип возвращаемого значения (void) и модификатор доступа (public).

# Переопределение (override) мотодов класса

Если в классе-родителе и классе-потомке реализованы методы с одинаковой сигнатурой, то в таком случае говорят, что метод подкласса переопределяет метод суперкласса. Когда переопределенный метод вызывается из подкласса, он всегда будет ссылаться на версию метода, определенную подклассом. Версия метода, определенная суперклассом, будет сокрыта.

|  |
| --- |
| **public** **class** SuperSomeClass  {  **public** **void** someMethod()  {  System.***out***.println("SuperSomeClass.someMethod");  }  }  **public** **class** SomeClass **extends** SuperSomeClass  {  @Override **public** **void** someMethod()  {  System.***out***.println("SomeClass.someMethod");  }  } |
| Пример 6. Переопределение метода |

Результат выполнения кода из примера 6 приведен на рисунке 2.

|  |
| --- |
|  |
| Рис. 2. Результат выполнения кода из примера 6 |

Таким образом, при переопределении методов, их сигнатура **полностью** совпадает (в противном случае, это уже перегрузка методов), и происходит перекрытие поведения метода суперкласса поведением метода класса-потомка. Однако, иногда возникают ситуации, когда поведение метода класса-родителя не нуждается в перекрытии, а должно быть дополнено. В этом случае с помощью ключевого слова «super» через точку происходит вызов метода суперкласса (см. пример 7).

|  |
| --- |
| **public** **class** SuperSomeClass  {  **public** **void** someMethod()  {  System.***out***.println("SuperSomeClass.someMethod");  }  }  **public** **class** SomeClass **extends** SuperSomeClass  {  @Override **public** **void** someMethod()  {  System.***out***.println("+SomeClass.someMethod");  **super**.someMethod(); //вызов метода суперкласса  System.***out***.println("-SomeClass.someMethod");  }  } |
| Пример 7. Вызов метода суперкласса из переопределенного метода класса-потомка |

|  |
| --- |
|  |
| Рис. 3. Результат выполения кода из примера 7 |