#### **14.5 Методы с дефолтной реализацией в интерфейсах**

### О чём вы узнаете

В этом материале вы познакомитесь с некоторыми возможностями Java, которые появились, начиная с JDK 8. Вы узнаете о дефолтных методах в интерфейсах, а также о ситуациях, в которых они могут быть полезны.

### Для чего нужны дефолтные методы

Посмотрите пример:

public interface TestInterface {  
    *// обычный метод интерфейса*  
    void regularMethod();  
}

Как вам известно, любой класс, реализующий интерфейс, должен реализовать все его методы. Это несложно сделать, если реализация всего одна или две. Однако, что произойдёт, если вы разработали библиотеку, которой пользуются сотни других разработчиков? Вы не сможете приказать им всем добавить новый метод в их реализации.

Именно поэтому в JDK 8 в интерфейсах были добавлены методы с дефолтной реализацией. Они являются эффективным способом решения этой проблемы, позволяя добавлять в интерфейс новые методы, которые автоматически доступны в реализующих этот интерфейс классах. Таким образом, обратная совместимость сохраняется без необходимости изменять код всех классов, то есть в связи с внедрением этого функционала старый код переписывать не придётся. Реализация, описанная в интерфейсе, будет неявно применяться при вызове этого метода у класса. Именно поэтому такие методы называются методами по умолчанию, или дефолтными методами. Чтобы создать  в интерфейсе метод по умолчанию, вам нужно добавить ключевое слово default к объявлению метода.

public interface TestInterface {  
    
    default void defaultMethod() {  
        *// метод интерфейса с дефолтной реализацией*  
    }  
}

### Несколько примеров

Чтобы лучше понять функциональность дефолтных методов, рассмотрите простой пример. Предположим, что у вас есть интерфейс Printer и одна его реализация (реализаций может быть и больше, но ограничимся пока одной).

public interface Printer {  
    void printBlackAndWhite();  
    default void printColor() {  
        System.out.println("Print color");  
    }  
  }

Реализация:

public class SimplePrinter implements Printer{  
  @Override  
  public void printBlackAndWhite() {  
    System.out.println("Print black and white");  
  }  
}

Как видно из примера, нужно реализовать только один из двух методов. Остальные методы реализовывать не требуется. Теперь запустим наш код:

public static void main(String[] args) {  
    Printer printer = new SimplePrinter();  
    printer.printBlackAndWhite();  
    printer.printColor();  
          }

Вывод:

Print black and white  
Print color

Обратите внимание, что дефолтный метод printColor() из нашего интерфейса Printer автоматически доступен в классе SimplePrinter. Кроме того, если в какой-то момент мы решим добавить дополнительные методы по умолчанию в интерфейс Printer, приложение продолжит работу, и нам не придётся реализовывать новые методы в классах. Наиболее распространённое использование методов интерфейса по умолчанию — постепенное предоставление дополнительных функций для данного типа без вреда для реализующих классов.

Как и любые другие методы, дефолтные методы можно переопределять:

public class SimplePrinter implements Printer{  
  @Override  
  public void printBlackAndWhite() {  
    System.out.println("Print black and white");  
  }  
  
  @Override  
  public void printColor() {  
    System.out.println("Print more color");  
  }  
}

И класс будет использовать именно переопределённый метод.

В реализации самой JDK начиная с JDK 8 также широко используются дефолтные методы. Вот несколько примеров:

1. java.util.Collection - этот интерфейс содержит множество методов, например, add, remove, contains, isEmpty и т.д. Начиная с JDK 8, этот интерфейс также содержит несколько default методов, например, forEach, removeIf и spliterator.
2. java.util.List - этот интерфейс расширяет интерфейс Collection и добавляет методы, специфичные для списков, например, get, set, indexOf, subList и т.д. Начиная с JDK 8, этот интерфейс также содержит несколько default методов, например, replaceAll, sort и spliterator.
3. java.util.Map - этот интерфейс представляет собой коллекцию ключ-значение, где каждый ключ может быть связан только с одним значением. Начиная с JDK 8, этот интерфейс также содержит несколько default методов, например, forEach, getOrDefault, putIfAbsent, remove и т.д.
4. java.util.Comparator - этот интерфейс определяет порядок сравнения для объектов. Начиная с JDK 8, этот интерфейс также содержит несколько default методов, например, reversed, thenComparing и nullsFirst.
5. java.util.Function - этот пакет содержит множество функциональных интерфейсов, каждый из которых определяет одну функцию. Начиная с JDK 8, многие из этих интерфейсов содержат default методы, например, andThen, compose, negate и другие

Эти методы существенно расширяют функционал JDK и позволяют писать более чистый и красивый код.

### Дефолтные методы и множественное наследование

Дефолтные методы — очень полезная функция, однако стоит упомянуть о некоторых особенностях. Поскольку Java позволяет классам реализовывать несколько интерфейсов, важно знать, что происходит, когда класс реализует несколько интерфейсов, определяющих одни и те же дефолтные методы. Чтобы лучше это понять, давайте определим новый интерфейс LaserPrinter и немного перепишем класс SimplePrinter:

public interface LaserPrinter {  
  default void printColor() {  
    System.out.println("LaserPrintColor");  
  }  
}

В этом интерфейсе определяется собственный дефолтный метод с таким же именем. Класс SimplePrinter будет реализовывать как Printer, так и LaserPrinter:

public class SimplePrinter implements Printer, LaserPrinter{  
  *//здесь наш код*  
}

В этом случае код просто не скомпилируется, так как возникнет конфликт, вызванный множественным наследованием интерфейсов (так называемая Diamond Problem). Класс SimplePrinter наследует оба метода по умолчанию и компилятор просто не знает, какой из них вызвать. Чтобы решить эту проблему, мы должны явно реализовать такой метод:

public class SimplePrinter implements Printer{  
  @Override  
  public void printBlackAndWhite() {  
    System.out.println("Print black and white");  
  }  
  
  @Override  
  public void printColor() {  
    System.out.println("Print color");  
  }  
}

Мы также можем сделать так, чтобы наш класс использовал дефолтные методы только одного из интерфейсов. Рассмотрите пример, в котором используется метод по умолчанию из интерфейса Printer:

@Override  
  public void printColor() {  
    Printer.super.printColor();  
}

### Заключение

В этом материале вы подробно рассмотрели дефолтные методы интерфейсов в Java. На первый взгляд кажется, что они ничем не отличаются от методов из абстрактных классов. Однако, если нужно обеспечить обратную совместимость с существующим кодом, дефолтные методы являются хорошим средством.