Расширение интерфейсов

Ключевое слово `extends` позволяет одному интерфейсу наследовать другой. Синтаксис определения такого наследования аналогичен синтаксису наследования классов. Когда класс реализует интерфейс, наследующий другой интерфейс, он должен предоставлять реализации всех методов, определенных по цепочке наследования интерфейсов. Ниже приведен характерный пример расширения интерфейсов.

```
// Один интерфейс может расширять другой
interface A {
    void meth1();
    void meth2();
}

// Теперь интерфейс B включает в себя методы meth1() и meth2()
// и добавляет метод meth3()
interface B extends A {
    void meth3();
}

// Этот класс должен реализовать все методы из интерфейсов A и B
class MyClass implements B {
    public void meth1() {
        System.out.println("Реализация метода meth1().");
    }

    public void meth2() {
        System.out.println("Реализация метода meth2().");
    }

    public void meth3() {
        System.out.println("Реализация метода meth3().");
    }
}

class IFExtend {
    public static void main(String arg[]) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

В порядке эксперимента можете попытаться удалить реализацию метода `meth1()` из класса `MyClass`. Это приведет к ошибке во время компиляции. Как отмечалось ранее, любой класс, реализующий интерфейс, должен реализовать и все определенные в этом интерфейсе методы, в том числе и любые методы, унаследованные от других интерфейсов.

Методы по умолчанию

Как пояснялось ранее, до версии JDK 8 в интерфейсе нельзя было вообще реализовывать методы. Это означало, что во всех предыдущих версиях Java ме-

тоды, указанные в интерфейсе, были абстрактными и не имели своего тела. Это традиционная форма интерфейса, обсуждавшаяся в предыдущих разделах главы. Но с выпуском версии JDK 8 положение изменилось, поскольку появилась новая возможность вводить в интерфейс так называемый *метод по умолчанию*. Иными словами, метод по умолчанию позволяет теперь объявлять в интерфейсе метод не абстрактным, а с конкретным телом. На стадии разработки версии JDK 8 метод по умолчанию назывался *методом расширения*, поэтому в литературе можно встретить оба обозначения этого нового языкового средства Java.

Главной побудительной причиной для внедрения методов по умолчанию было стремление предоставить средства, позволявшиеся расширять интерфейсы, не нарушая уже существующий код. Напомним, что если ввести новый метод в широко применяемый интерфейс, такое дополнение нарушит уже существующий код из-за того, что не удастся обнаружить реализацию нового метода. Данное затруднение позволяет устранить метод по умолчанию, поскольку он предоставляет реализацию, которая будет использоваться в том случае, если не будет явно предоставлена другая реализация. Таким образом, ввод метода по умолчанию в интерфейс не нарушит уже существующий код.

Еще одной побудительной причиной для внедрения метода по умолчанию было стремление указывать в интерфейсе, по существу, необязательные методы в зависимости от того, каким образом используется интерфейс. Например, в интерфейсе можно определить группу методов, воздействующих на последовательность элементов. Один из этих методов можно назвать `remove()` и предназначить его для удаления элемента из последовательности. Но если интерфейс предназначен для поддержки как изменяемых, так и неизменяемых последовательностей, то метод `remove()` оказывается, по существу, необязательным, поскольку его нельзя применять к неизменяемым последовательностям. В прошлом в классе, предназначавшемся для обработки неизменяемых последовательностей, приходилось реализовывать фактически пустой метод `remove()`, даже если он и не был нужен. А теперь реализацию по умолчанию метода `remove()`, не выполняющего никаких действий или генерирующего исключение, можно указать в интерфейсе. Благодаря этому исключается потребность реализовывать свой замещающий вариант этого метода в классе, предназначаемом для обработки неизменяемых последовательностей. Следовательно, если предоставить в интерфейсе метод `remove()` по умолчанию, его реализация в классе окажется необязательной.

Важно отметить, что внедрение методов по умолчанию не изменяет главную особенность интерфейсов: неспособность сохранять данные состояния. В частности, в интерфейсе по-прежнему недопустимы переменные экземпляра. Следовательно, интерфейс отличается от класса тем, что он не допускает сохранение состояния. Более того, создавать экземпляр самого интерфейса нельзя. Поэтому интерфейс должен быть по-прежнему реализован в классе, если требуется получить его экземпляр, несмотря на возможность определять в интерфейсе методы по умолчанию, начиная с версии JDK 8.

И последнее замечание: как правило, методы по умолчанию служат специальным целям. А создаваемые интерфейсы по-прежнему определяют, главным образом, *что* именно следует сделать, но не *как* это сделать. Тем не менее внедрение методов по умолчанию доставляет дополнительные удобства для программирования на Java.

Основы применения методов по умолчанию

Метод по умолчанию определяется в интерфейсе таким же образом, как и метод в классе. Главное отличие состоит в том, что в начале объявления метода по умолчанию указывается ключевое слово `default`. Рассмотрим в качестве примера следующий простой интерфейс:

```
public interface MyIF {
  // Это объявление обычного метода в интерфейсе.
  // Он НЕ предоставляет реализацию по умолчанию
  int getNumber();

  // А это объявление метода по умолчанию. Обратите
  // внимание на его реализацию по умолчанию
  default String getString() {
    return "Объект типа String по умолчанию";
  }
}
```

В интерфейсе `MyIF` объявляются два метода. Первый из них, `getNumber()`, является стандартно объявляемым в интерфейсе и вообще не предоставляет никакой реализации. А второй метод, `getString()`, включает в себя реализацию по умолчанию. В данном случае он просто возвращает символьную строку "`Объект типа String по умолчанию`". Обратите особое внимание на порядок объявления метода `getString()`, где в самом начале указан модификатор доступа `default`. Этот синтаксис можно обобщить. В частности, для того чтобы определить метод по умолчанию, его объявление достаточно предварить ключевым словом `default`.

В связи с тем что в объявление метода `getString()` включена его реализация по умолчанию, его совсем не обязательно переопределять в классе, реализующем интерфейс `MyIF`. Например, объявление приведенного ниже класса `MyIFImp` вполне допустимо.

```
// Реализовать интерфейс MyIF
class MyIFImp implements MyIF {
  // В этом классе должен быть реализован только метод getNumber(),
  // определенный в интерфейсе MyIF.
  // А вызов метода getString() разрешается по умолчанию
  public int getNumber() {
    return 100;
  }
}
```

В следующем примере кода получается экземпляр класса `MyIFImp`, который используется для вызова обоих методов, `getNumber()` и `getString()`:

```
// Использовать метод по умолчанию
class DefaultMethodDemo {
public static void main(String args[]) {

    MyIFImp obj = new MyIFImp();

    // Метод getNumber() можно вызвать, т.к. он явно реализован
    // в классе MyIFImp:
    System.out.println(obj.getNumber());

    // Метод getString() также можно вызвать, т.к. в интерфейсе
    // имеется его реализация по умолчанию:
```

```
    System.out.println(obj.getString());
  }
}
```

Ниже приведен результат, выводимый при выполнении данного кода.

```
100
Объект типа String по умолчанию
```

Как видите, в данном примере кода автоматически используется реализация метода `getString()` по умолчанию. Определять и тем более реализовывать этот метод в классе `MyIFImp` совсем не обязательно. (Разумеется, реализация метода `getString()` в классе *потребуется* в том случае, если его предполагается использовать в этом классе для каких-нибудь других целей, а не только по умолчанию.)

В классе, реализующем интерфейс, вполне возможно определять собственную реализацию метода по умолчанию. Например, метод `getString()` переопределяется в классе `MyIFImp2`, как показано ниже. Теперь в результате вызова метода `getString()` возвращается другая символьная строка.

```
class MyIFImp2 implements MyIF {
  // В этом классе предоставляются реализации обоих методов
  // getNumber() и getString()
  public int getNumber() {
    return 100;
  }

  public String getString() {
    return "Это другая символьная строка.";
  }
}
```

Более практический пример

Несмотря на то что в приведенном выше примере демонстрируется механизм применения методов по умолчанию, в нем все же не показана практическая польза от такого нововведения. С этой целью вернемся к интерфейсу `IntStack`, рассмотренному ранее в этой главе. Ради удобства обсуждения допустим, что интерфейс `IntStack` широко применяется во многих программах и что в нем требуется ввести метод, очищающий стек, чтобы подготовить его к повторному использованию. Следовательно, интерфейс `IntStack` требуется дополнить новыми функциональными возможностями, но не нарушая уже существующий код. Прежде это было просто невозможно, но благодаря внедрению методов по умолчанию это совсем не трудно сделать теперь. Например, интерфейс `IntStack` можно усовершенствовать следующим образом:

```
interface IntStack {
  void push(int item); // сохранить элемент в стеке
  int pop(); // извлечь элемент из стека

  // У метода clear() теперь имеется вариант по умолчанию, поэтому
  // его придется реализовать в том существующем классе, где уже
  // применяется интерфейс IntStack
  default void clear() {
    System.out.println("Метод clear() не реализован.");
  }
}
```

В данном примере метод `clear()` по умолчанию выводит сообщение о том, что он не реализован. И это вполне допустимо, поскольку метод `clear()` нельзя вызвать из любого уже существующего класса, реализующего интерфейс `IntStack`, если он не был определен в предыдущей версии интерфейса `IntStack`. Но метод `clear()` может быть реализован в новом классе вместе с интерфейсом `IntStack`. Более того, новую реализацию метода `clear()` потребуется определить лишь в том случае, если он используется. Следовательно, метод по умолчанию предоставляет возможность сделать следующее:

- изящно расширить интерфейс со временем;
- предоставить дополнительные функциональные возможности, исключая замещающую реализацию в классе, если эти функциональные возможности не требуются.

Следует также иметь в виду, что в реальном коде метод `clear()` должен генерировать исключение, а не выводить сообщение. Об исключениях речь пойдет в следующей главе. Проработав материал этой главы, попробуйте видоизменить реализацию метода `clear()` по умолчанию таким образом, чтобы он генерировал исключение типа `UnsupportedOperationException`.

Вопросы множественного наследования

Как пояснялось ранее в этой книге, множественное наследование классов в Java не поддерживается. Но теперь, когда в интерфейсы внедрены методы по умолчанию, может возникнуть вопрос: позволяет ли интерфейс обойти это ограничение? По существу, позволяет. Напомним о следующем главном отличии класса от интерфейса: в классе могут сохраняться данные состояния, особенно с помощью переменных экземпляра, тогда как в интерфейсе этого сделать нельзя.

Несмотря на все сказанное выше, методы по умолчанию предоставляют отчасти возможности, которые обычно связываются с понятием множественного наследования. Например, в одном классе можно реализовать два интерфейса. Если в каждом из этих интерфейсов предоставляются методы по умолчанию, то некоторое поведение наследуется от обоих интерфейсов. Поэтому в какой-то, хотя и ограниченной, степени эти методы все же поддерживают множественное наследование. Нетрудно догадаться, что в подобных случаях может возникнуть конфликт имен.

Допустим, два интерфейса, `Alpha` и `Beta`, реализуются в классе `MyClass`. Что, если в обоих этих интерфейсах предоставляется метод `reset()`, объявляемый с реализацией по умолчанию? Какой из вариантов этого метода будет выбран в классе `MyClass`: из интерфейса `Alpha` или `Beta`? С другой стороны, рассмотрим ситуацию, когда интерфейс `Beta` расширяет интерфейс `Alpha`. Какой вариант метода по умолчанию используется в этом случае? А что, если в классе `MyClass` предоставляется собственная реализация этого метода? Для ответа на эти и другие аналогичные вопросы в Java определен ряд правил разрешения подобных конфликтов.

Во-первых, во всех подобных случаях приоритет отдается реализации метода в классе над его реализацией в интерфейсе. Так, если в классе `MyClass` переопределяется метод по умолчанию `reset()`, то выбирается его вариант, реализуемый

в классе MyClass. Это происходит даже в том случае, если в классе MyClass реализуются оба интерфейса, Alpha и Beta. И это означает, что методы по умолчанию переопределяются их реализацией в классе MyClass.

Во-вторых, если в классе реализуются два интерфейса с одинаковым методом по умолчанию, но этот метод не переопределяется в данном классе, то возникает ошибка. Если же в классе MyClass реализуются оба интерфейса, Alpha и Beta, но метод reset() в нем не переопределяется, то и в этом случае возникает ошибка.

В тех случаях, когда один интерфейс наследует другой и в обоих интерфейсах определяется общий метод по умолчанию, предпочтение отдается варианту метода из наследующего интерфейса. Так, если интерфейс Beta расширяет интерфейс Alpha, то используется вариант метода reset() из интерфейса Beta. Впрочем, используя новую форму ключевого слова super, вполне возможно ссылаться на реализацию по умолчанию в наследуемом интерфейсе. Эта общая форма ключевого слова super выглядит следующим образом:

***имя_интерфейса*.super.*имя_метода*()**

Так, если из интерфейса Beta требуется обратиться по ссылке к методу по умолчанию reset() в интерфейсе Alpha, то для этого достаточно воспользоваться следующим оператором:

```
Alpha.super.reset();
```

Применение статических методов в интерфейсе

Начиная с версии JDK 8, у интерфейсов появилась еще одна возможность: определять в нем один или несколько статических методов. Аналогично статическим методам в классе, метод, объявляемый как static в интерфейсе, можно вызывать независимо от любого объекта. И для этого не требуется ни реализация такого метода в интерфейсе, ни экземпляр самого интерфейса. Напротив, для вызова статического метода достаточно указать имя интерфейса и через точку имя самого метода. Ниже приведена общая форма вызова статического метода из интерфейса. Обратите внимание на ее сходство с формой вызова статического метода из класса.

имя_интерфейса.имя__статического_метода

В приведенном ниже примере кода демонстрируется ввод статического метода getDefaultNumber() в упоминавшийся ранее интерфейс MyIF. Этот метод возвращает нулевое значение.

```
public interface MyIF {
  // Это объявление обычного метода в интерфейсе.
  // Он НЕ предоставляет реализацию по умолчанию
  int getNumber();

  // А это объявление метода по умолчанию. Обратите
  // внимание на его реализацию по умолчанию
  default String getString() {
    return "Объект типа String по умолчанию";
  }
```

```
  // Это объявление статического метода в интерфейсе
  static int getDefaultNumber() {
    return 0;
  }
}
```

Метод `getDefaultNumber()` может быть вызван следующим образом:

```
int defNum = MyIF.getDefaultNumber();
```

Как упоминалось выше, для вызова метода `getDefaultNumber()` реализация или экземпляр интерфейса `MyIF` не требуется, поскольку это статический метод. И последнее замечание: статические методы из интерфейсов не наследуются ни реализующими их классами, ни подчиненными интерфейсами.

Заключительные соображения по поводу пакетов и интерфейсов

В примерах, представленных в этой книге, пакеты и или интерфейсы используются нечасто. Тем не менее оба эти языковые средства являются очень важной составляющей среды программирования на Java. Буквально все реальные программы, которые придется писать на Java, будут входить в состав пакетов и скорее всего будут реализовывать интерфейсы. Поэтому очень важно освоить пакет и интерфейсы настолько, чтобы свободно пользоваться этими языковыми средствами, программируя на Java.