Интерфейсы 9

Интерфейсы и абстрактные классы улучшают структуру кода и способствуют отделению интерфейса от реализации.

В традиционных языках программирования такие механизмы не получили особого распространения. Например, в C++ существует лишь косвенная поддержка этих концепций. Сам факт их существования в Java показывает, что эти концепции были сочтены достаточно важными для прямой поддержки в языке.

Мы начнем с понятия *абстрактного класса*, который представляет собой своего рода промежуточную ступень между обычным классом и интерфейсом. Абстрактные классы — важный и необходимый инструмент для создания классов, содержащих нереализованные методы. Применение «чистых» интерфейсов возможно не всегда.

Абстрактные классы и методы

В примере с классами музыкальных инструментов из предыдущей главы методы базового класса Instrument всегда оставались «фиктивными». Попытка вызова такого метода означала, что в программе произошла какая-то ошибка. Это объяснялось тем, что класс Instrument создавался для определения *общего интерфейса* всех классов, производных от него.

В этих примерах общий интерфейс создавался для единственной цели— его разной реализации в каждом производном типе. Интерфейс определяет базовую форму, общность всех производных классов. Такие классы, как Instrument, также называют *абстрактными базовыми классами* или просто *абстрактными классами*.

Если в программе определяется абстрактный класс вроде Instrument, создание объектов такого класса практически всегда бессмысленно. Абстрактный класс создается для работы с набором классов через общий интерфейс. А если Instrument только выражает интерфейс, а создание объектов того класса не имеет смысла, вероятно, пользователю лучше запретить создавать такие объекты. Конечно, можно заставить все методы Instrument выдавать ошибки, но в этом

случае получение информации откладывается до стадии выполнения. Ошибки такого рода лучше обнаруживать во время компиляции.

В языке Java для решения подобных задач применяются *абстрактные методы*[1]. Абстрактный метод незавершен; он состоит только из объявления и не имеет тела. Синтаксис объявления абстрактных методов выглядит так:

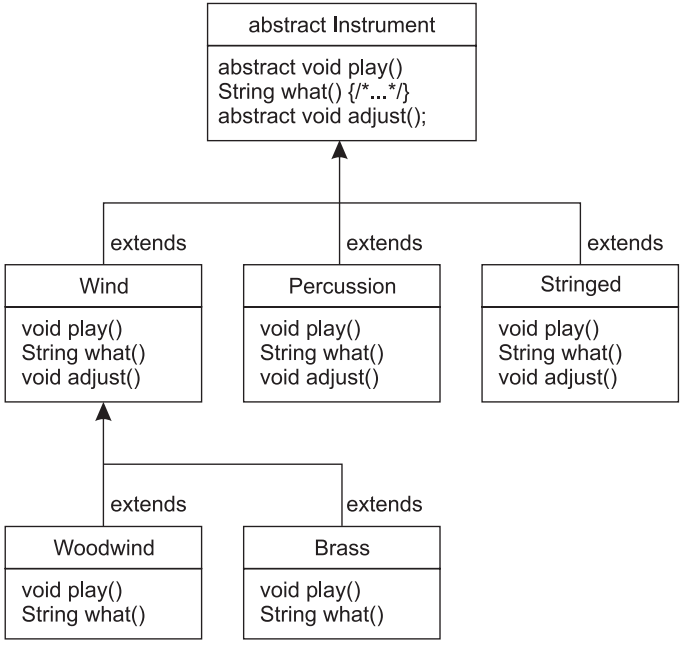
```
abstract void f();
```

Класс, содержащий абстрактные методы, называется *абстрактным классом*. Такие классы тоже должны помечаться ключевым словом abstract (в противном случае компилятор выдает сообщение об ошибке).

Если вы объявляете класс, производный от абстрактного класса, но хотите иметь возможность создания объектов нового типа, вам придется предоставить определения для всех абстрактных методов базового класса. Если этого не сделать, производный класс тоже останется абстрактным, и компилятор заставит пометить *новый* класс ключевым словом abstract.

Можно создавать класс с ключевым словом abstract даже тогда, когда в нем не имеется ни одного абстрактного метода. Это бывает полезно в ситуациях, где в классе абстрактные методы просто не нужны, но необходимо запретить создание экземпляров этого класса.

Класс Instrument очень легко можно сделать абстрактным. Только некоторые из его методов станут абстрактными, поскольку объявление класса как abstract не подразумевает, что все его методы должны быть абстрактными. Вот что получится:



[1] Аналог *чисто виртуальных методов* языка C++.

А вот как выглядит реализация примера оркестра с использованием абстрактных классов и методов:

```
//: interfaces/music4/Music4.java
// Абстрактные классы и методы.
package interfaces.music4;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

abstract class Instrument {
     private int i; // Память выделяется для каждого объекта
     public abstract void play(Note n);
     public String what() { return "Instrument"; }
     public abstract void adjust();
}

class Wind extends Instrument {
     public void play(Note n) {
              print("Wind.play() " + n);
     }
     public String what() { return "Wind"; }
     public void adjust() {}
}

class Percussion extends Instrument {
     public void play(Note n) {
              print("Percussion.play() " + n);
     }
     public String what() { return "Percussion"; }
     public void adjust() {}
}

class Stringed extends Instrument {
     public void play(Note n) {
              print("Stringed.play() " + n);
     }
     public String what() { return "Stringed"; }
     public void adjust() {}
}

class Brass extends Wind {
     public void play(Note n) {
              print("Brass.play() " + n);
     }
     public void adjust() { print("Brass.adjust()"); }
}

class Woodwind extends Wind {
     public void play(Note n) {
              print("Woodwind.play() " + n);
     }
     public String what() { return "Woodwind"; }
}

public class Music4 {
     // Работа метода не зависит от фактического типа объекта,
     // поэтому типы, добавленные в систему, будут работать правильно:
```

продолжение ⇩

```
        static void tune(Instrument i) {
                // ...
                i.play(Note.MIDDLE_C);
        }
        static void tuneAll(Instrument[] e) {
                for(Instrument i : e)
                        tune(i);
        }
        public static void main(String[] args) {
                // Восходящее преобразование при добавлении в массив:
                Instrument[] orchestra = {
                        new Wind(),
                        new Percussion(),
                        new Stringed(),
                        new Brass(),
                        new Woodwind()
                };
                tuneAll(orchestra);
        }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~
```

Как видите, объем изменений минимален.

Создавать абстрактные классы и методы полезно, так как они подчеркивают абстрактность класса, а также сообщают и пользователю класса, и компилятору, как следует с ним обходиться. Кроме того, абстрактные классы играют полезную роль при переработке программ, потому что они позволяют легко перемещать общие методы вверх по иерархии наследования.

Интерфейсы

Ключевое слово interface становится следующим шагом на пути к абстракции. Оно используется для создания полностью абстрактных классов, вообще не имеющих реализации. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов.

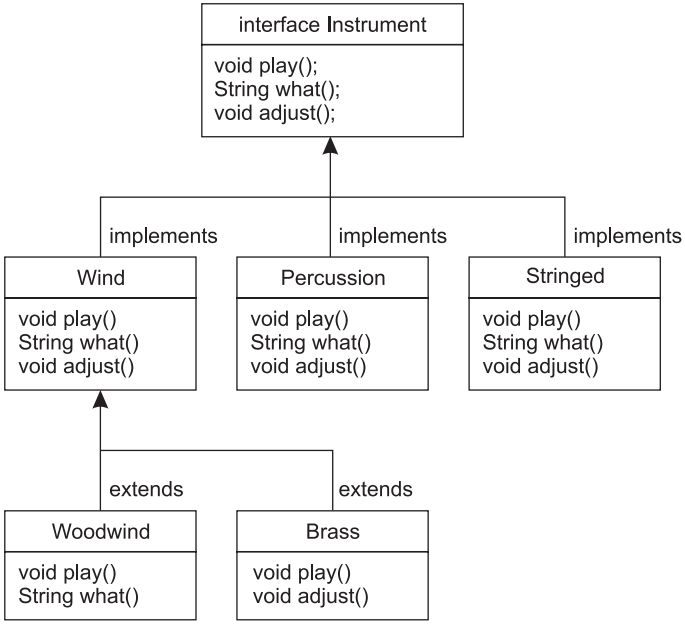
Ключевое слово interface фактически означает: «Именно так должны выглядеть все классы, которые *реализуют* данный интерфейс». Таким образом, любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того. Интерфейс определяет своего рода «протокол взаимодействия» между классами.

Однако интерфейс представляет собой нечто большее, чем абстрактный класс в своем крайнем проявлении, потому что он позволяет реализовать подобие «множественного наследования» C++: иначе говоря, создаваемый класс может быть преобразован к нескольким базовым типам.

Чтобы создать интерфейс, используйте ключевое слово interface вместо class. Как и в случае с классами, вы можете добавить перед словом interface специфи-

катор доступа public (но только если интерфейс определен в файле, имеющем то же имя) или оставить для него дружественный доступ, если он будет использоваться только в пределах своего пакета. Интерфейс также может содержать поля, но они автоматически являются статическими (static) и неизменными (final).

Для создания класса, реализующего определенный интерфейс (или группу интерфейсов), используется ключевое слово implements. Фактически оно означает: «Интерфейс лишь определяет форму, а сейчас будет показано, как это *работает*». В остальном происходящее выглядит как обычное наследование. Рассмотрим реализацию на примере иерархии классов Instrument:



Классы Woodwind и Brass свидетельствуют, что реализация интерфейса представляет собой обычный класс, от которого можно создавать производные классы.

При описании методов в интерфейсе вы можете явно объявить их открытыми (public), хотя они являются таковыми даже без спецификатора. Однако при реализации интерфейса его методы *должны* быть объявлены как public. В противном случае будет использоваться доступ в пределах пакета, а это приведет к уменьшению уровня доступа во время наследования, что запрещается компилятором Java.

Все сказанное можно увидеть в следующем примере с объектами Instrument. Заметьте, что каждый метод интерфейса ограничивается простым объявлением; ничего большего компилятор не разрешит. Вдобавок ни один из методов интерфейса Instrument не объявлен со спецификатором public, но все методы автоматически являются открытыми:

```
//: interfaces/music5/Music5.java
// Интерфейсы.
package interfaces.music5;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

interface Instrument {
      // Константа времени компиляции:
      int VALUE = 5; // является и static, и final
      // Определения методов недопустимы:
      void play(Note n); // Автоматически объявлен как public
      void adjust();
}

class Wind implements Instrument {
      public void play(Note n) {
               print(this + ".play() " + n);
      }
      public String toString() { return "Wind"; }
      public void adjust() { print(this + ".adjust()"); }
}

class Percussion implements Instrument {
      public void play(Note n) {
               print(this + ".play() " + n);
      }
      public String toString() { return "Percussion"; }
      public void adjust() { print(this + ".adjust()"); }
}

class Stringed implements Instrument {
      public void play(Note n) {
               print(this + ".play() " + n);
      }
      public String toString() { return "Stringed"; }
      public void adjust() { print(this + ".adjust()"); }
}

class Brass extends Wind {
      public String toString() { return "Brass"; }
}

class Woodwind extends Wind {
      public String toString() { return "Woodwind"; }
}

public class Music5 {
      // Работа метода не зависит от фактического типа объекта,
      // поэтому типы, добавленные в систему, будут работать правильно:
      static void tune(Instrument i) {
               // ...
               i.play(Note.MIDDLE_C);
      }
      static void tuneAll(Instrument[] e) {
               for(Instrument i : e)
                        tune(i);
      }
  public static void main(String[] args) {
```

```
        // Восходящее преобразование при добавлении в массив:
                Instrument[] orchestra = {
                        new Wind(),
                        new Percussion(),
                        new Stringed(),
                        new Brass(),
                        new Woodwind()
                };
                tuneAll(orchestra);
        }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~
```

В этой версии присутствует еще одно изменение: метод what() был заменен на toString(). Так как метод toString() входит в корневой класс Object, его присутствие в интерфейсе не обязательно.

Остальной код работает так же, как прежде. Неважно, проводите ли вы преобразование к «обычному» классу с именем Instrument, к абстрактному классу с именем Instrument или к интерфейсу с именем Instrument — действие будет одинаковым. В методе tune() ничто не указывает на то, является класс Instrument «обычным» или абстрактным, или это вообще не класс, а интерфейс.

Отделение интерфейса от реализации

В любой ситуации, когда метод работает с классом вместо интерфейса, вы ограничены использованием этого класса или его субклассов. Если метод должен быть применен к классу, не входящему в эту иерархию, — значит, вам не повезло. Интерфейсы в значительной мере ослабляют это ограничение. В результате код становится более универсальным, пригодным для повторного использования.

Представьте, что у нас имеется класс Processor с методами name() и process(). Последний получает входные данные, изменяет их и выдает результат. Базовый класс расширяется для создания разных специализированных типов Processor. В следующем примере производные типы изменяют объекты String (обратите внимание: ковариантными могут быть возвращаемые значения, но не типы аргументов):

```
//: interfaces/classprocessor/Apply.java
package interfaces.classprocessor;
import java.util.*;
import static net.mindview.util.Print.*;

class Processor {
      public String name() {
              return getClass().getSimpleName();
      }
      Object process(Object input) { return input; }
```

продолжение ⇩

```
}

class Upcase extends Processor {
     String process(Object input) { // Ковариантный возвращаемый тип
               return ((String)input).toUpperCase();
     }
}

class Downcase extends Processor {
     String process(Object input) {
               return ((String)input).toLowerCase();
     }
}

class Splitter extends Processor {
     String process(Object input) {
               // Аргумент split() используется для разбиения строки
               return Arrays.toString(((String)input).split(" "));
     }
}

public class Apply {
     public static void process(Processor p, Object s) {
               print("Используем Processor " + p.name());
               print(p.process(s));
     }
     public static String s =
               "Disagreement with beliefs is by definition incorrect";
     public static void main(String[] args) {
               process(new Upcase(), s);
               process(new Downcase(), s);
               process(new Splitter(), s);
     }
} /* Output:
Используем Processor Upcase
DISAGREEMENT WITH BELIEFS IS BY DEFINITION INCORRECT
Используем Processor Downcase
disagreement with beliefs is by definition incorrect
Используем Processor Splitter
[Disagreement, with, beliefs, is, by, definition, incorrect]
*///:~
```

Метод Apply.process() получает любую разновидность Processor, применяет ее к Object, а затем выводит результат. Метод split() является частью класса String. Он получает объект String, разбивает его на несколько фрагментов по ограничителям, определяемым переданным аргументом, и возвращает String[]. Здесь он используется как более компактный способ создания массива String.

Теперь предположим, что вы обнаружили некое семейство электронных фильтров, которые тоже было бы уместно использовать с методом Apply.process():

```
//: interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
     private static long counter;
```

```
      private final long id = counter++;
      public String toString() { return "Waveform " + id; }
} ///:~

//: interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
      public String name() {
                return getClass().getSimpleName();
      }
      public Waveform process(Waveform input) { return input; }
} ///:~

//: interfaces/filters/LowPass.java
package interfaces.filters;

public class LowPass extends Filter {
      double cutoff;
      public LowPass(double cutoff) { this.cutoff = cutoff; }
      public Waveform process(Waveform input) {
                return input; // Фиктивная обработка
      }
} ///:~

//: interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
      double cutoff;
      public HighPass(double cutoff) { this.cutoff = cutoff; }
      public Waveform process(Waveform input) { return input; }
} ///:~

//: interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
      double lowCutoff, highCutoff;
      public BandPass(double lowCut, double highCut) {
                lowCutoff = lowCut;
                highCutoff = highCut;
      }
      public Waveform process(Waveform input) { return input; }
} ///:~
```

Класс Filter содержит те же интерфейсные элементы, что и Processor, но, поскольку он не является производным от Processor (создатель класса Filter и не подозревал, что вы захотите использовать его как Processor), он не может использоваться с методом Apply.process(), хотя это выглядело бы вполне естественно. Логическая привязка между Apply.process() и Processor оказывается более сильной, чем реально необходимо, и это обстоятельство препятствует повторному использованию кода Apply.process(). Также обратите внимание, что входные и выходные данные относятся к типу Waveform.

Но, если преобразовать класс Processor в интерфейс, ограничения ослабляются и появляется возможность повторного использования Apply.process(). Обновленные версии Processor и Apply выглядят так:

```
//: interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;

public interface Processor {
     String name();
     Object process(Object input);
} ///:~

//: interfaces/interfaceprocessor/Apply.java
package interfaces.interfaceprocessor;
import static net.mindview.util.Print.*;

public class Apply {
     public static void process(Processor p, Object s) {
               print("Using Processor " + p.name());
               print(p.process(s));
     }
} ///:~
```

В первом варианте повторного использования кода клиентские программисты пишут свои классы с поддержкой интерфейса:

```
//: interfaces/interfaceprocessor/StringProcessor.java
package interfaces.interfaceprocessor;
import java.util.*;

public abstract class StringProcessor implements Processor{
     public String name() {
               return getClass().getSimpleName();
     }
     public abstract String process(Object input);
     public static String s =
               "If she weighs the same as a duck, she's made of wood";
     public static void main(String[] args) {
               Apply.process(new Upcase(), s);
               Apply.process(new Downcase(), s);
               Apply.process(new Splitter(), s);
     }
}

class Upcase extends StringProcessor {
     public String process(Object input) { // Ковариантный возвращаемый тип
               return ((String)input).toUpperCase();
  }
}

class Downcase extends StringProcessor {
     public String process(Object input) {
               return ((String)input).toLowerCase();
     }
}

class Splitter extends StringProcessor {
```

```
        public String process(Object input) {
                return Arrays.toString(((String)input).split(" "));
        }
} /* Output:
Используем Processor Upcase
IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
Используем Processor Downcase
if she weighs the same as a duck, she's made of wood
Используем Processor Splitter
[If, she, weighs, the, same, as, a, duck,, she's, made, of, wood]
*///:~
```

Впрочем, довольно часто модификация тех классов, которые вы собираетесь использовать, невозможна. Например, в примере с электронными фильтрами библиотека была получена из внешнего источника. В таких ситуациях применяется паттерн «адаптер»: вы пишете код, который получает имеющийся интерфейс, и создаете тот интерфейс, который вам нужен:

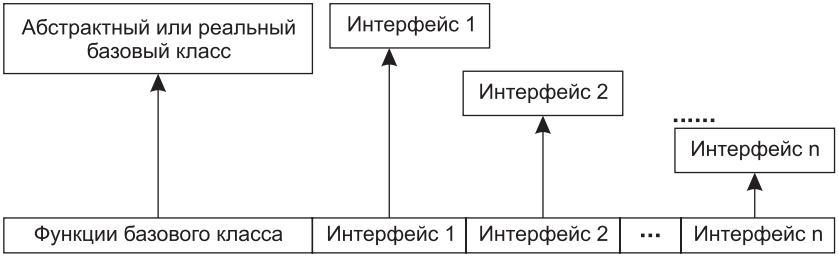
```
//: interfaces/interfaceprocessor/FilterProcessor.java
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
        Filter filter;
        public FilterAdapter(Filter filter) {
                this.filter = filter;
        }
        public String name() { return filter.name(); }
        public Waveform process(Object input) {
                return filter.process((Waveform)input);
        }
}

public class FilterProcessor {
        public static void main(String[] args) {
                Waveform w = new Waveform();
                Apply.process(new FilterAdapter(new LowPass(1.0)), w);
                Apply.process(new FilterAdapter(new HighPass(2.0)), w);
                Apply.process(
                        new FilterAdapter(new BandPass(3.0, 4.0)), w);
        }
} /* Output:
Используем Processor LowPass
Waveform 0
Используем Processor HighPass
Waveform 0
Используем Processor BandPass
Waveform 0
*///:~
```

Конструктор FilterAdapter получает исходный интерфейс (Filter) и создает объект с требуемым интерфейсом Processor. Также обратите внимание на применение делегирования в классе FilterAdapter.

Отделение интерфейса от реализации позволяет применять интерфейс к разным реализациям, а следовательно, расширяет возможности повторного использования кода.

«Множественное наследование» в Java

Так как интерфейс по определению не имеет реализации (то есть не обладает памятью для хранения данных), нет ничего, что могло бы помешать совмещению нескольких интерфейсов. Это очень полезная возможность, так как в некоторых ситуациях требуется выразить утверждение: «Икс является *и* А, *и* Б, *и* В одновременно». В C++ подобное совмещение интерфейсов нескольких классов называется *множественным наследованием*, и оно имеет ряд очень неприятных аспектов, поскольку каждый класс может иметь свою реализацию. В Java можно добиться аналогичного эффекта, но, поскольку реализацией обладает всего один класс, проблемы, возникающие при совмещении нескольких интерфейсов в C++, в Java принципиально невозможны:



При наследовании базовый класс вовсе не обязан быть абстрактным или «реальным» (без абстрактных методов). Если наследование *действительно* осуществляется не от интерфейса, то среди прямых «предков» класс может быть только один — все остальные должны быть интерфейсами. Имена интерфейсов перечисляются вслед за ключевым словом implements и разделяются запятыми. Интерфейсов может быть сколько угодно, причем к ним можно проводить восходящее преобразование. Следующий пример показывает, как создать новый класс на основе реального класса и нескольких интерфейсов:

```
//: interfaces/Adventure.java
// Использование нескольких интерфейсов.

interface CanFight {
     void fight();
}

interface CanSwim {
     void swim();
}

interface CanFly {
     void fly();
}

class ActionCharacter {
     public void fight() {}
}

class Hero extends ActionCharacter
     implements CanFight, CanSwim, CanFly {
     public void swim() {}
```

```
    public void fly() {}
}

public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Используем объект в качестве типа CanFight
        u(h); // Используем объект в качестве типа CanSwim
        v(h); // Используем объект в качестве типа CanFly
        w(h); // Используем объект в качестве ActionCharacter
    }
} ///:~
```

Мы видим, что класс Hero сочетает реальный класс ActionCharacter с интерфейсами CanFight, CanSwim и CanFly. При объединении реального класса с интерфейсами на первом месте должен стоять реальный класс, а за ним следуют интерфейсы (иначе компилятор выдаст ошибку).

Заметьте, что объявление метода fight() в интерфейсе CanFight совпадает с тем, что имеется в классе ActionCharacter, и поэтому в классе Hero *нет* определения метода fight(). Интерфейсы можно расширять, но при этом получается другой интерфейс. Необходимым условием для создания объектов нового типа является наличие всех определений. Хотя класс Hero не имеет явного определения метода fight(), это определение существует в классе ActionCharacter, что и делает возможным создание объектов класса Hero.

Класс Adventure содержит четыре метода, которые принимают в качестве аргументов разнообразные интерфейсы и реальный класс. Созданный объект Hero передается всем этим методам, а это значит, что выполняется восходящее преобразование объекта к каждому интерфейсу по очереди. Система интерфейсов Java спроектирована так, что она нормально работает без особых усилий со стороны программиста.

Помните, что главная причина введения в язык интерфейсов представлена в приведенном примере: это возможность выполнять восходящее преобразование к нескольким базовым типам. Вторая причина для использования интерфейсов совпадает с предназначением абстрактных классов: запретить программисту-клиенту создание объектов этого класса.

Возникает естественный вопрос: что лучше — интерфейс или абстрактный класс? Если можно создать базовый класс без определений методов и переменных-членов, выбирайте именно интерфейс, а не абстрактный класс. Вообще говоря, если известно, что нечто будет использоваться как базовый класс, первым делом постарайтесь сделать это «нечто» интерфейсом.

Расширение интерфейса через наследование

Наследование позволяет легко добавить в интерфейс объявления новых методов, а также совместить несколько интерфейсов в одном. В обоих случаях получается новый интерфейс, как показано в следующем примере:

```
//: interfaces/HorrorShow.java
// Расширение интерфейса с помощью наследования.

interface Monster {
     void menace();
}

interface DangerousMonster extends Monster {
     void destroy();
}

interface Lethal {
     void kill();
}

class DragonZilla implements DangerousMonster {
     public void menace() {}
     public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
     void drinkBlood();
}

class VeryBadVampire implements Vampire {
     public void menace() {}
     public void destroy() {}
     public void kill() {}
     public void drinkBlood() {}
}

public class HorrorShow {
     static void u(Monster b) { b.menace(); }
     static void v(DangerousMonster d) {
               d.menace();
               d.destroy();
     }
     static void w(Lethal l) { l.kill(); }
     public static void main(String[] args) {
               DangerousMonster barney = new DragonZilla();
               u(barney);
               v(barney);
               Vampire vlad = new VeryBadVampire();
               u(vlad);
               v(vlad);
               w(vlad);
     }
} ///:~
```

DangerousMonster представляет собой простое расширение Monster, в результате которого образуется новый интерфейс. Он реализуется классом DragonZilla.

Синтаксис, использованный в интерфейсе Vampire, работает *только* при наследовании интерфейсов. Обычно ключевое слово extends может использоваться всего с одним классом, но, так как интерфейс можно составить из нескольких других интерфейсов, extends подходит для написания нескольких имен

интерфейсов при создании нового интерфейса. Как нетрудно заметить, имена нескольких интерфейсов разделяются при этом запятыми.

Конфликты имен при совмещении интерфейсов

При реализации нескольких интерфейсов может возникнуть небольшая проблема. В только что рассмотренном примере интерфейс `CanFight` и класс `ActionCharacter` имеют идентичные методы `void fight()`. Хорошо, если методы полностью тождественны, но что, если они различаются по сигнатуре или типу возвращаемого значения? Рассмотрим такой пример:

```
//: interfaces/InterfaceCollision.java
package interfaces;

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
     public void f() {}
     public int f(int i) { return 1; } // перегружен
}

class C3 extends C implements I2 {
     public int f(int i) { return 1; } // перегружен
}

class C4 extends C implements I3 {
     // Идентичны, все нормально:
     public int f() { return 1; }
}

// Методы различаются только по типу возвращаемого значения:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

Трудность возникает из-за того, что переопределение, реализация и перегрузка образуют опасную «смесь». Кроме того, перегруженные методы не могут различаться только возвращаемыми значениями. Если убрать комментарий в двух последних строках программы, сообщение об ошибке разъясняет суть происходящего:

```
InterfaceCollision.java:23: f() в C не может
реализовать f() в I1; попытка использовать
несовместимые возвращаемые типы
обнаружено: int
требуется: void
InterfaceCollision.java:24: интерфейсы I3 и I1
несовместимы; оба определяют f(),
но с различными возвращаемыми типами
```

Использование одинаковых имен методов в интерфейсах, предназначенных для совмещения, обычно приводит к запутанному и трудному для чтения коду. Постарайтесь по возможности избегать таких ситуаций.

Интерфейсы как средство адаптации

Одной из самых убедительных причин для использования интерфейсов является возможность определения нескольких реализаций для одного интерфейса. В простых ситуациях такая схема принимает вид метода, который при вызове передается интерфейсу; от вас потребуется реализовать интерфейс и передать объект методу.

Соответственно, интерфейсы часто применяются в архитектурном паттерне *«Стратегия»*. Вы пишете метод, выполняющий несколько операций; при вызове метод получает интерфейс, который тоже указываете вы. Фактически вы говорите: «Мой метод может использоваться с любым объектом, удовлетворяющим моему интерфейсу». Метод становится более гибким и универсальным.

Например, конструктор класса Java SE5 Scanner получает интерфейс Readable. Анализ показывает, что Readable не является аргументом любого другого метода из стандартной библиотеки Java — этот интерфейс создавался исключительно для Scanner, чтобы его аргументы не ограничивались определенным классом. При таком подходе можно заставить Scanner работать с другими типами. Если вы хотите создать новый класс, который может использоваться со Scanner, реализуйте в нем интерфейс Readable:

```
//: interfaces/RandomWords.java
// Реализация интерфейса для выполнения требований метода
import java.nio.*;
import java.util.*;

public class RandomWords implements Readable {
      private static Random rand = new Random(47);
      private static final char[] capitals =
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
      private static final char[] lowers =
               "abcdefghijklmnopqrstuvwxyz".toCharArray();
      private static final char[] vowels =
               "aeiou".toCharArray();
      private int count;
      public RandomWords(int count) { this.count = count; }
      public int read(CharBuffer cb) {
               if(count-- == 0)
                        return -1; // Признак конца входных данных
               cb.append(capitals[rand.nextInt(capitals.length)]);
               for(int i = 0; i < 4; i++) {
                        cb.append(vowels[rand.nextInt(vowels.length)]);
                        cb.append(lowers[rand.nextInt(lowers.length)]);
               }
               cb.append(" ");
               return 10; // Количество присоединенных символов
      }
      public static void main(String[] args) {
               Scanner s = new Scanner(new RandomWords(10));
               while(s.hasNext())
                        System.out.println(s.next());
      }
} /* Output:
Yazeruyac
```

```
Fowenucor
Goeazimom
Raeuuacio
Nuoadesiw
Hageaikux
Ruqicibui
Numasetih
Kuuuuozog
Waqizeyoy
*///:~
```

Интерфейс Readable требует только присутствия метода read(). Метод read() либо добавляет данные в аргумент CharBuffer (это можно сделать несколькими способами; обращайтесь к документации CharBuffer), либо возвращает –1 при отсутствии входных данных.

Допустим, у нас имеется класс, не реализующий интерфейс Readable, — как заставить его работать с Scanner? Перед вами пример класса, генерирующего вещественные числа:

```
//: interfaces/RandomDoubles.java
import java.util.*;

public class RandomDoubles {
      private static Random rand = new Random(47);
      public double next() { return rand.nextDouble(); }
      public static void main(String[] args) {
               RandomDoubles rd = new RandomDoubles();
               for(int i = 0; i < 7; i ++)
                        System.out.print(rd.next() + " ");
      }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:~
```

Мы снова можем воспользоваться схемой адаптера, но на этот раз адаптируемый класс создается наследованием и реализацией интерфейса Readable. Псевдомножественное наследование, обеспечиваемое ключевым словом interface, позволяет создать новый класс, который одновременно является и RandomDoubles, и Readable:

```
//: interfaces/AdaptedRandomDoubles.java
// Создание адаптера посредством наследования
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles extends RandomDoubles
implements Readable {
      private int count;
      public AdaptedRandomDoubles(int count) {
               this.count = count;
      }
      public int read(CharBuffer cb) {
               if(count-- == 0)
                        return -1;
               String result = Double.toString(next()) + " ";
```

продолжение ⇩

```
                cb.append(result);
                return result.length();
        }
        public static void main(String[] args) {
                Scanner s = new Scanner(new AdaptedRandomDoubles(7));
                while(s.hasNextDouble())
                        System.out.print(s.nextDouble() + " ");
        }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:~
```

Так как интерфейсы можно добавлять подобным образом только к существующим классам, это означает, что любой класс может быть адаптирован для метода, получающего интерфейс. В этом заключается преимущество интерфейсов перед классами.

Поля в интерфейсах

Так как все поля, помещаемые в интерфейсе, автоматически являются статическими (static) и неизменными (final), объявление interface хорошо подходит для создания групп постоянных значений. До выхода Java SE5 только так можно было имитировать перечисляемый тип enum из языков C и C++:

```
//: interfaces/Months.java
// Использование интерфейсов для создания групп констант.
package interfaces;

public interface Months {
      int
                JANUARY = 1, FEBRUARY = 2, MARCH = 3,
                APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
                AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
                NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

Отметьте стиль Java — использование только заглавных букв (с разделением слов подчеркиванием) для полей со спецификаторами static и final, которым присваиваются фиксированные значения на месте описания. Поля интерфейса автоматически являются открытыми (public), поэтому нет необходимости явно указывать это.

В Java SE5 появилось гораздо более мощное и удобное ключевое слово enum, поэтому надобность в применении интерфейсов для определения констант отпала. Впрочем, эта старая идиома еще может встречаться в некоторых старых программах.

Инициализация полей интерфейсов

Поля, определяемые в интерфейсах, не могут быть «пустыми константами», но могут инициализироваться не-константными выражениями. Например:

```
//: interfaces/RandVals.java
// Инициализация полей интерфейсов
// не-константными выражениями.
import java.util.*;

public interface RandVals {
      Random RAND = new Random(47);
      int RANDOM_INT = RAND.nextInt(10);
      long RANDOM_LONG = RAND.nextLong() * 10;
      float RANDOM_FLOAT = RAND.nextLong() * 10;
      double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} ///:~
```

Так как поля являются статическими, они инициализируются при первой загрузке класса, которая происходит при первом обращении к любому из полей интерфейса. Простой тест:

```
//: interfaces/TestRandVals.java
import static net.mindview.util.Print.*;

public class TestRandVals {
      public static void main(String[] args) {
               print(RandVals.RANDOM_INT);
               print(RandVals.RANDOM_LONG);
               print(RandVals.RANDOM_FLOAT);
               print(RandVals.RANDOM_DOUBLE);
      }
} /* Output:
8
-32032247016559954
-8.5939291E18
5.779976127815049
*///:~
```

Конечно, поля не являются частью интерфейса. Данные хранятся в статической области памяти, отведенной для данного интерфейса.

Вложенные интерфейсы

Интерфейсы могут вкладываться в классы и в другие интерфейсы[1]. При этом обнаруживается несколько весьма интересных особенностей:

```
//: interfaces/nesting/NestingInterfaces.java
package interfaces.nesting;

class A {
      interface B {
               void f();
      }
      public class BImp implements B {
               public void f() {}
      }
      private class BImp2 implements B {
```

продолжение ⇨

[1] Благодарю Мартина Даннера за то, что он задал этот вопрос на семинаре.

```
                public void f() {}
        }
        public interface C {
                void f();
        }
        class CImp implements C {
                public void f() {}
        }
        private class CImp2 implements C {
                public void f() {}
        }
        private interface D {
                void f();
        }
        private class DImp implements D {
                public void f() {}
        }
        public class DImp2 implements D {
                public void f() {}
        }
        public D getD() { return new DImp2(); }
        private D dRef;
        public void receiveD(D d) {
                dRef = d;
                dRef.f();
        }
}

interface E {
        interface G {
                void f();
        }
        // Избыточное объявление public:
        public interface H {
                void f();
        }
        void g();
        // Не может быть private внутри интерфейса:
        //! private interface I {}
}

public class NestingInterfaces {
        public class BImp implements A.B {
                public void f() {}
        }
        class CImp implements A.C {
                public void f() {}
        }
        // Private-интерфейс не может быть реализован нигде,
        //  кроме как внутри класса, где он был определен:
        //! class DImp implements A.D {
        //! public void f() {}
        //! }
        class EImp implements E {
                public void g() {}
        }
        class EGImp implements E.G {
                public void f() {}
```

```
        }
        class EImp2 implements E {
                public void g() {}
                class EG implements E.G {
                        public void f() {}
                }
        }
        public static void main(String[] args) {
                A a = new A();
                // Нет доступа к A.D:
                //! A.D ad = a.getD();
                // Не возвращает ничего, кроме A.D:
                //! A.DImp2 di2 = a.getD();
                // Член интерфейса недоступен:
                //! a.getD().f();
                // Только другой класс A может использовать getD():
                A a2 = new A();
                a2.receiveD(a.getD());
        }
} ///:~
```

Синтаксис вложения интерфейса в класс достаточно очевиден. Вложенные интерфейсы, как и обычные, могут иметь «пакетную» или открытую (public) видимость.

Любопытная подробность: интерфейсы могут быть объявлены закрытыми (private), как видно на примере A.D (используется тот же синтаксис описания, что и для вложенных классов). Для чего нужен закрытый вложенный интерфейс? Может показаться, что такой интерфейс реализуется только в виде закрытого (private) вложенного класса, подобного DImp, но A.DImp2 показывает, что он также может иметь форму открытого (public) класса. Тем не менее класс A.DImp2 «замкнут» сам на себя. Факт реализации private-интерфейса не может упоминаться в программе, поэтому реализация такого интерфейса — просто способ принудительного определения методов этого интерфейса без добавления информации о дополнительном типе (то есть восходящее преобразование становится невозможным).

Метод getD() усугубляет сложности, связанные с private-интерфейсом, — это открытый (public) метод, возвращающий ссылку на закрытый (private) интерфейс. Что можно сделать с возвращаемым значением этого метода? В методе main() мы видим несколько попыток использовать это возвращаемое значение, и все они оказались неудачными. Заставить метод работать можно только одним способом — передать возвращаемое значение некоторому объекту, которому разрешено его использование (в нашем случае это еще один объект A, у которого имеется необходимый метод receiveD()).

Интерфейс E показывает, что интерфейсы могут быть вложены друг в друга. Впрочем, правила для интерфейсов — в особенности то, что все элементы интерфейса должны быть открытыми (public), — здесь строго соблюдаются, поэтому интерфейс, вложенный внутрь другого интерфейса, автоматически объявляется открытым и его нельзя сделать закрытым (private).

Пример NestingInterfaces демонстрирует разнообразные способы реализации вложенных интерфейсов. Особо стоит отметить тот факт, что при реализации

интерфейса вы не обязаны реализовывать вложенные в него интерфейсы. Также закрытые (private) интерфейсы нельзя реализовать за пределами классов, в которых они описываются.

Интерфейсы и фабрики

Предполагается, что интерфейс предоставляет своего рода «шлюз» к нескольким альтернативным реализациям. Типичным способом получения объектов, соответствующих интерфейсу, является паттерн *«фабрика»*. Вместо того, чтобы вызывать конструктор напрямую, вы вызываете метод объекта-фабрики, который предоставляет реализацию интерфейса — в этом случае программный код теоретически отделяется от реализации интерфейса, благодаря чему становится возможной совершенно прозрачная замена реализации. Следующий пример демонстрирует типичную структуру фабрики:

```
//: interfaces/Factories.java
import static net.mindview.util.Print.*;

interface Service {
     void method1();
     void method2();
}

interface ServiceFactory {
     Service getService();
}

class Implementation1 implements Service {
     Implementation1() {} // Доступ в пределах пакета
     public void method1() {print("Implementation1 method1");}
     public void method2() {print("Implementation1 method2");}
}

class Implementation1Factory implements ServiceFactory {
     public Service getService() {
              return new Implementation1();
     }
}

class Implementation2 implements Service {
     Implementation2() {} // Доступ в пределах пакета
     public void method1() {print("Implementation2 method1");}
     public void method2() {print("Implementation2 method2");}
}

class Implementation2Factory implements ServiceFactory {
     public Service getService() {
              return new Implementation2();
     }
}

public class Factories {
     public static void serviceConsumer(ServiceFactory fact) {
              Service s = fact.getService();
```

```
                s.method1();
                s.method2();
        }
        public static void main(String[] args) {
                serviceConsumer(new Implementation1Factory());
                // Реализации полностью взаимозаменяемы
                serviceConsumer(new Implementation2Factory());
        }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~
```

Без применения фабрики вам пришлось бы где-то указать точный тип создаваемого объекта Service, чтобы он мог вызвать подходящий конструктор.

Но зачем вводить лишний уровень абстракции? Данный паттерн часто применяется при создании библиотек. Допустим, вы создаете систему для игр, которая позволяла бы играть в шашки и шахматы на одной доске:

```
//: interfaces/Games.java
// Игровая библиотека с использованием фабрики
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
        private int moves = 0;
        private static final int MOVES = 3;
        public boolean move() {
                print("Checkers move " + moves);
                return ++moves != MOVES;
        }
}

class CheckersFactory implements GameFactory {
        public Game getGame() { return new Checkers(); }
}

class Chess implements Game {
        private int moves = 0;
        private static final int MOVES = 4;
        public boolean move() {
                print("Chess move " + moves);
                return ++moves != MOVES;
        }
}

class ChessFactory implements GameFactory {
        public Game getGame() { return new Chess(); }
}

public class Games {
        public static void playGame(GameFactory factory) {
                Game s = factory.getGame();
```

продолжение ⇩

```
                while(s.move())
                        ;
        }
        public static void main(String[] args) {
                playGame(new CheckersFactory());
                playGame(new ChessFactory());
        }
} /* Output:
Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
*///:~
```

Если класс Games представляет сложный блок кода, такое решение позволит повторно использовать его для разных типов игр.

В следующей главе будет представлен более элегантный способ реализации фабрик на базе анонимных внутренних классов.

Резюме

После первого знакомства интерфейсы выглядят так хорошо, что может показаться, будто им всегда следует отдавать предпочтение перед реальными классами. Конечно, в любой ситуации, когда вы создаете класс, вместо него можно создать интерфейс и фабрику.

Многие программисты поддались этому искушению. Тем не менее любая абстракция должна быть мотивирована реальной потребностью. Основное назначение интерфейсов — возможность переработки реализации в случае необходимости, а не введение лишнего уровня абстракции вместе с дополнительными сложностями. Дополнительные сложности могут оказаться довольно существенными. А представьте, что кто-то будет вынужден разбираться в вашем коде и в конечном итоге поймет, что интерфейсы были добавлены «на всякий случай», без веских причин — в таких ситуациях все проектирование, которое выполнялось данным разработчиком, начинает выглядеть довольно сомнительно.

В общем случае рекомендуется *отдавать предпочтение классам перед интерфейсами*. Начните с классов, а если необходимость интерфейсов станет очевидной — переработайте архитектуру своего проекта. Интерфейсы — замечательный инструмент, но ими нередко злоупотребляют.