Национальный Открытый Университет "ИНТУИТ": www.intuit.ru

Вадим Монахов

Лекция 8. Наследование: проблемы и альтернативы. Интерфейсы. Композиция

8.1. Проблемы множественного наследования классов. Интерфейсы

Достаточно часто требуется совмещать в объекте поведение, характерное для двух или более независимых иерархий. Но еще чаще требуется писать единый полиморфный код для объектов из таких иерархий в случае, когда эти объекты обладают схожим поведением. Как мы знаем, за поведение объектов отвечают методы. Это значит, что в полиморфном коде требуется для объектов из разных классов вызывать методы, имеющие одинаковую сигнатуру, но разную реализацию. *Унарное наследование*, которое мы изучали до сих пор, и при котором у класса может быть только один прародитель, не обеспечивает такой возможности. При унарном наследовании нельзя ввести переменную, которая бы могла ссылаться на экземпляры из разных иерархий, так как она должна иметь тип, совместимый с базовыми классами этих иерархий.

В C++ для решения данных проблем используется *множественное наследование*. Оно означает, что у класса может быть не один непосредственный прародитель, а два или более. В этом случае проблема совместимости с классами из разных иерархий решается путем создания класса, наследующего от необходимого *числа классов*-прародителей.

Но при множественном наследовании классов возникает ряд трудно *разрешимых* проблем, поэтому в *Java* оно не поддерживается. Основные причины, по которым произошел отказ от использования *множественного наследования* классов: *наследование* ненужных полей и методов, конфликты совпадающих имен из разных ветвей наследования.

В частности, так называемое ромбовидное *наследование*, когда у класса `A` наследники `B` и `C`, а от них наследуется *класс* `D`. Поэтому *класс* `D` получает поля и методы, имеющиеся в классе `A`, в удвоенном количестве – один комплект по линии родителя `B`, другой – по линии родителя `C`.

Конечно, в C++ имеются средства решать указанные проблемы. Например, проблемы ромбовидного наследования снимаются использованием так называемых виртуальных классов, благодаря чему при ромбовидном наследовании потомкам достается только один комплект членов класса `A`, и различения имен из разных ветвей класса с помощью имен классов. Но в результате получается заметное усложнение логики работы с классами и объектами, вызывающее *логические ошибки*, не отслеживаемые компилятором. Поэтому в *Java* используется *множественное наследование с помощью интерфейсов*, лишенное практически всех указанных проблем.

Интерфейсы являются важнейшими *элементами языков* программирования, применяемых как для написания полиморфного кода, так и для межпрограммного обмена. Концепция интерфейсов *Java* была первоначально заимствована из языка Objective-C, но в дальнейшем развивалась и дополнялась. В настоящее время она, фактически, стала основой объектного программирования в *Java*, заменив концепцию *множественного наследования*, используемую в C++.

Интерфейсы являются специальной разновидностью полностью *абстрактных классов*. То есть таких классов, в которых вообще нет реализованных методов – все методы абстрактные. Полей данных в них также нет, но можно задавать *константы* (неизменяемые переменные класса). *Класс* в *Java* должен быть наследником одного класса-родителя, и может быть наследником произвольного числа интерфейсов. Сами интерфейсы также могут наследоваться от интерфейсов, причем также с разрешением на *множественное наследование*.

Отсутствие в интерфейсах полей данных и реализованных методов снимает почти все проблемы *множественного наследования* и обеспечивает изящный инструмент для написания полиморфного кода. Например, то, что все коллекции обладают методами, перечисленными в разделе о коллекциях, обеспечивается тем, что их классы являются наследниками интерфейса `Collection`. Аналогично, все классы *итераторов* являются наследниками интерфейса `Iterator`, и т.д.

Декларация интерфейса очень похожа на декларацию класса:

```
МодификаторВидимости interface ИмяИнтерфейса
  extends ИмяИнтерфейса1, ИмяИнтерфейса2,..., ИмяИнтерфейсаN{
  декларация констант;
  декларация заголовков методов;
}
```

В качестве *модификатора видимости* может использоваться либо *слово* `public` – общая видимость, либо модификатор должен отсутствовать, в этом случае обеспечивается видимость по умолчанию – пакетная. В списке прародителей, расширением которых является данный *интерфейс*, указываются прародительские

интерфейсы `ИмяИнтерфейса1`, `ИмяИнтерфейса2` и так далее. Если *список* прародителей пуст, в отличие от классов, *интерфейс* не имеет прародителя.

Для имен интерфейсов в *Java* нет специальных правил, за исключением того, что для них, как и для других *объектных типов*, имя принято начинать с заглавной буквы. Мы будем использовать для имен интерфейсов *префикс* `I` (от слова `Interface`), чтобы их имена легко отличать от имен классов.

Объявление *константы* осуществляется почти так же, как в классе:

```
МодификаторВидимости Тип ИмяКонстанты = значение;
```

В качестве необязательного *модификатора видимости* может использоваться *слово* `public`. Либо модификатор должен отсутствовать – но при этом *видимость также считается* **public**, *а не пакетной*. Еще одним отличием от декларации в классе является то, что при задании в интерфейсе все поля автоматически считаются окончательными (модификатор `final`), т.е. без *права* изменения, и к тому же являющимися *переменными класса* (модификатор `static`). Сами модификаторы `static` и `final` при этом ставить не надо.

Декларация метода в интерфейсе осуществляется очень похоже на декларацию *абстрактного метода* в классе – указывается только *заголовок метода*:

```
МодификаторВидимости Тип ИмяМетода(списокПараметров)
 throws списокИсключений;
```

В качестве *модификатора видимости*, как и в предыдущем случае, может использоваться либо *слово* `public`, либо модификатор должен отсутствовать. При этом *видимость также считается* **public**, *а не пакетной*. В списке исключений через запятую перечисляются типы *проверяемых исключений* (потомки `Exception`), которые может возбуждать метод. Часть `throws списокИсключений` является необязательной. При задании в интерфейсе все методы автоматически считаются общедоступными (`public`) абстрактными (`abstract`) методами объектов.

Пример задания интерфейса:

```
package figures_pkg;


public interface IScalable {
    public int getSize();
    public void setSize(int newSize);
}
```

Класс можно наследовать от одного *родительского класса* и от произвольного количества интерфейсов. Но вместо слова `extends` используется *зарезервированное слово* `implements` – *реализует*. Говорят, что *класс*-наследник интерфейса реализует соответствующий *интерфейс*, так как он обязан реализовать все его методы. Это гарантирует, что объекты для любых классов, наследующих некий *интерфейс*, могут вызывать методы этого интерфейса. Что позволяет писать полиморфный код для объектов из разных *иерархий классов*. При реализации возможно добавление новых полей и методов, как и при обычном наследовании. Поэтому можно считать, что это просто один из вариантов наследования, обладающий некоторыми особенностями.

Уточнение: в абстрактных классах, реализующих *интерфейс*, реализации методов может не быть – наследуется декларация *абстрактного метода* из интерфейса.

Замечание: *Интерфейс* также может реализовывать (implements) другой *интерфейс*, если в том при задании типа использовался *шаблон* (**generics**, template). Но эта тема выходит за пределы данного учебного пособия.

В наследнике класса, реализующего *интерфейс*, можно переопределить методы этого интерфейса. Повторное указание в списке родителей интерфейса, который уже был унаследован кем-то из прародителей, запрещено.

Реализации у сущности типа *интерфейс* не бывает, как и для *абстрактных классов*. То есть экземпляров интерфейсов не бывает. Но, как и для *абстрактных классов*, можно вводить переменные типа *интерфейс*. Эти переменные могут ссылаться на объекты, принадлежащие классам, реализующим соответствующий *интерфейс*. То есть классам-наследникам этого интерфейса.

Правила совместимости таковы: переменной типа *интерфейс* можно присваивать ссылку на *объект* любого класса, реализующего этот *интерфейс*.

С помощью переменной типа *интерфейс* разрешается вызывать только методы, декларированные в данном интерфейсе, а не любые методы данного объекта. Если, конечно, не использовать приведение типа.

Основное назначение переменных *интерфейсного типа* – вызов с их помощью методов, продекларированных в соответствующем интерфейсе. Если такой переменной назначена *ссылка* на *объект*, можно гарантировать, что из этого объекта разрешено вызывать эти методы, независимо от того, какому классу принадлежит *объект*. Ситуация очень похожа с полиморфизмом на основе виртуальных и *динамических методов* объектов. Но гарантией работоспособности служит не одинаковость *сигнатуры методов* в одной иерархии, а одинаковость *сигнатуры методов* в разных иерархиях – благодаря совпадению с декларацией одного и того же интерфейса. Обязательно, чтобы методы были методами объектов – *полиморфизм* на основе методов класса невозможен, так как для вызовов этих методов используется *статическое связывание* (на этапе компиляции).

8.2. Отличия интерфейсов от классов. Проблемы наследования интерфейсов

Не бывает экземпляров типа интерфейс, то есть экземпляров интерфейсов, реализующих тип интерфейс.

Список элементов интерфейса может включать только методы и константы. Поля данных использовать нельзя.

Элементы интерфейса всегда имеют тип видимости `public` (в том числе без явного указания). Не разрешено использовать модификаторы видимости кроме `public`.

В интерфейсах не бывает конструкторов и *деструкторов*.

Методы не могут иметь модификаторов `abstract` (хотя и являются абстрактными по умолчанию), `static`, *`native`*, `synchronized`, `final`, `private`, `protected`.

Интерфейс, как и класс, наследует все методы прародителя, однако только на уровне абстракций, без реализации методов. То есть интерфейс наследует только обязательность реализации этих методов в классе, поддерживающем этот интерфейс.

Наследование через интерфейсы может быть множественным. В декларации интерфейса можно указать, что интерфейс наследуется от одного или нескольких прародительских интерфейсов.

Реализация интерфейса может быть только в классе, при этом, если он не является абстрактным, то должен реализовать все методы интерфейса.

Наследование класса от интерфейсов также может быть множественным.

Кроме указанных отличий имеется еще одно, связанное с проблемой *множественного наследования*. В наследуемых классом интерфейсах могут содержаться методы, имеющие одинаковую сигнатуру (хотя, возможно, и отличающиеся контрактом). Как выбрать в классе или при вызове из объекта тот или иной из этих методов? Ведь, в отличие от *перегруженных методов*, *компилятор* не сможет их различить. Такая ситуация называется конфликтом имен.

Аналогичная ситуация может возникнуть и с константами, хотя, в отличие от методов, реально этого почти никогда не происходит.

Для использования констант из разных интерфейсов решением является квалификация имени *константы* именем соответствующего интерфейса – все аналогично *разрешению конфликта* имен в случае пакетов. Например:

```
public interface I1 {
    Double PI=3.14;
}

public interface I2 {
    Double PI=3.1415;
}

class C1 implements I1,I2 {
  void m1(){
    System.out.println("I1.PI="+ I1.PI);
    System.out.println("I2.PI="+ I2.PI);
  };
}
```

Но для методов такой способ различения имен запрещен. Поэтому наследовать от двух и более интерфейсов методы, имеющие совпадающие сигнатуры, но различающиеся контракты, нельзя. Если сигнатуры различаются, проблем нет, и используется *перегрузка*. Если контракты совпадают или совместимы, в классе можно реализовать один метод, который будет реализацией для всех интерфейсов, в которых продекларирован метод с таким контрактом.

Методы, объявленные в интерфейсе, могут возбуждать *проверяемые исключения*. Контракты методов считаются совместимыми в случае, когда контракты отличаются только типом возбуждаемых исключительных ситуаций, причем классы этих исключений лежат в одной иерархии. При этом в классе, реализующем *интерфейс*, метод должен быть объявлен как возбуждающий совместимый с интерфейсом *тип исключения* – то есть либо такой же, как в прародительском интерфейсе, либо являющийся наследником этого типа.
Подведем некоторый итог:

В том случае, если *класс* `A2` на уровне абстракций ведет себя так же, как *класс* `A1`, но кроме того обладает дополнительными особенностями поведения, следует использовать *наследование*. То есть считать *класс* `A2` наследником класса `A1`. Действует правило "`A2` есть `A1`" (A2 is a A1).

Если для нескольких классов `A1,B1,…` из разных иерархий можно на уровне абстракций выделить общность поведения, следует задать *интерфейс* `I`, который описывает эти абстракции поведения. А классы задать как наследующие этот *интерфейс*. Таким образом, действует правило "`A1`, `B1`,… есть `I`" (A1, B1,… is a I).

Множественное наследование в *Java* может быть двух видов:

- Только от интерфейсов, без наследования реализации.
- От класса и от интерфейсов, с наследованием реализации от прародительского класса.

Если *класс*-прародитель унаследовал какой-либо *интерфейс*, все его потомки также будут наследовать этот *интерфейс*.

8.3. Пример на использование интерфейсов

Приведем пример *абстрактного класса*, являющегося наследником `Figure`, и реализующего указанный выше *интерфейс* `IScalable`:

```
package figures_pkg;

public abstract class ScalableFigure extends Figure implements IScalable {
    private int size;

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
        this.size=size;
    }
}
```

В качестве наследника приведем код класса `Circle`:

```
package figures_pkg;
import java.awt.*;

public class Circle extends ScalableFigure {

    public Circle(Graphics g,Color bgColor, int r){
        setGraphics(g);
        setBgColor(bgColor);
        setSize(r);
    }

    public Circle(Graphics g,Color bgColor){
        setGraphics(g);
        setBgColor(bgColor);
        setSize( (int)Math.round(Math.random()*40) );
    }
```

```
    public void show(){
        Color oldC=getGraphics().getColor();
        getGraphics().setColor(Color.BLACK);
        getGraphics().drawOval(getX(),getY(),getSize(),getSize());
        getGraphics().setColor(oldC);
    }

    public void hide(){
        Color oldC=getGraphics().getColor();
        getGraphics().setColor(getBgColor());
        getGraphics().drawOval(getX(),getY(),getSize(),getSize());
        getGraphics().setColor(oldC);
    }
};
```

Приведем пример наследования интерфейсом от интерфейса:

```
package figures_pkg;

public interface IStretchable extends IScalable{
    double getAspectRatio();
    void setAspectRatio(double aspectRatio);
    int getWidth();
    void setWidth(int width);
    int getHeight();
    void setHeight(int height);
}
```

Интерфейс `IScalable` описывает методы объекта, способного менять свой размер (*size*). При этом *отношение* ширины к высоте (`AspectRatio` – *отношение* сторон) у фигуры не меняется. *Интерфейс* `IStretchable` описывает методы объекта, способного менять не только свой размер, но и "растягиваться" – изменять *отношение* ширины к высоте (`AspectRatio`).

К интерфейсам применимы как оператор `instanceof` , так и *приведение типов*. Например, фрагмент кода для изменения случайным образом размера объекта с помощью интерфейса `IScalable` может выглядеть так:

```
Object object;
...
object= Circle(...);//конструктор создает окружность
...
if(object instanceof IScalable){
    ((IScalable) object).setSize( (int)(Math.random()*80) );
}
```

Все очень похоже на использование класса `ScalableFigure` :

```
Figure figure;
...
figure = Circle(...);//конструктор создает окружность
...
if( figure instanceof IScalable){
    figure.hide();
    ((IScalable)figure).setSize((int)(Math.random()*80));
```

```
    figure.show();
}
```

Но если во втором случае *переменная* `figure` имеет тип `Figure`, то есть связанный с ней *объект* обязан быть фигурой, то на переменную `object` такое ограничение не распространяется. Зато фигуру можно скрывать и показывать, а для переменной типа `Object` это можно делать только после проверки, что `object` является экземпляром (то есть `instanceof`) класса `Figure`.

Аналогичный код можно написать в случае использования переменной типа `IScalable`:

```
IScalable scalableObj;
...
scalableObj = Circle(...);//конструктор создает окружность
...
scalableObj.setSize((int)(Math.random()*80));
```

Заметим, что *присваивание* `Object object= Circle(...)` разрешено, так как `Circle` – наследник `Object`. Аналогично, *присваивание* `Figure figure = Circle(...)` разрешено, так как `Circle` – наследник `Figure`. И, наконец, *присваивание* `scalableObj =Circle(...)` разрешено, так как `Circle` – наследник `IScalable`.

При замене в коде `Circle(...)` на `Dot(...)` мы бы получили правильный код в первых двух случаях, а вот *присваивание* `scalableObj = Dot (...);` вызвало бы ошибку компиляции, так как *класс* `Dot` не реализует *интерфейс* `IScalable`, то есть не является его потомком.

8.4. Композиция как альтернатива множественному наследованию

Как уже говорилось ранее, *наследование* относится к одному из важных аспектов, присущих объектам – поведению. Причем оно относится не к самим объектам, а к классам. Но имеется и другой аспект, присущий объектам – внутреннее устройство. При наследовании этот аспект скорее скрывается, чем подчеркивается: наследники должны быть устроены так, чтобы отличие в их устройстве не сказывалось на абстракциях их поведения.

Композиция – это описание объекта как состоящего из других объектов (*отношение агрегации*, или включения как составной части) или находящегося с ними в отношении ассоциации (объединения независимых объектов). Если *наследование* характеризуется отношением "is-a" ("это есть", "является"), то *композиция* характеризуется отношением "has-a" ("имеет в своем составе", "состоит из") и "use-a" ("использует").

Важность использования композиции связана с тем, что она позволяет *объединять отдельные части в единую более сложную систему*. Причем описание и *испытание* работоспособности отдельных частей можно делать независимо от других частей, а тем более от всей сложной системы. Таким образом, *композиция* – это *объединение* частей в единую систему.

В качестве примера *агрегации* можно привести классический пример – автомобиль. Он *состоит из* корпуса, колес, двигателя, карбюратора, топливного бака и т.д. Каждая из этих частей, в свою *очередь*, состоит из более простых деталей. И так далее, до того уровня, когда деталь можно считать единым целым, не включающий в себя другие объекты.

Шофер также является неотъемлемой частью автомобиля, но вряд ли можно считать, что автомобиль состоит из шофера и других частей. Но можно говорить, что у автомобиля обязательно должен быть шофер. Либо говорить, что шофер *использует* автомобиль. *Отношение* объекта "автомобиль" и объекта "шофер" гораздо слабее, чем *агрегация*, но все-таки весьма сильное – это *композиция* в узком смысле этого слова.

И, наконец, *отношение* автомобиля с находящимися в нем сумками или другими посторонними предметами – это *ассоциация*. То есть *отношение* независимых предметов, которые на некоторое время образовали единую систему. В таких случаях говорят, что автомобиль *используют* для того, чтобы отвезти предметы по нужному адресу.

С точки зрения программирования на *Java композиция* любого вида – это наличие в объекте поля *ссылочного типа*. Вид композиции определяется условиями создания связанного с этой ссылочной переменной объекта и изменения этой ссылки. Если такой вспомогательный *объект* создается одновременно с главным объектом и "умирает" вместе с ним – это *агрегация*. В противном случае это или *композиция* в узком смысле слова, или *ассоциация*.

Композиция во многих случаях может служить альтернативой *множественному наследованию*, причем именно в тех ситуациях, когда *наследование* интерфейсов "не работает". Это бывает в случаях, когда надо унаследовать от двух или более классов их поля и методы.

Приведем пример. Пусть у нас имеются классы `Car` ("Автомобиль") , *класс* `Driver` ("Шофер") и *класс* `Speed` ("Скорость"). И пусть это совершенно независимые классы. Зададим *класс* `MovingCar` ("движущийся автомобиль") как

```
public class MovingCar extends Car{
  Driver driver;
  Speed speed;
…}
```

Особенностью объектов `MovingCar` будет то, что они включают в себя не только особенности *поведения автомобиля*, но и все особенности объектов типа `Driver` и `Speed` . Например, автомобиль "знает" своего водителя: если у нас имеется *объект* `movingCar` , то `movingCar.driver` обеспечит *доступ* к объекту "водитель" (если, конечно, *ссылка* не равна `null`). В результате чего можно будет пользоваться общедоступными (и только!) методами этого объекта. То же относится к полю `speed` . И нам не надо строить гибридный *класс*-монстр, в котором от родителей `Car` , `Driver` и `Speed` унаследовано по механизму *множественного наследования* нечто вроде машино-кентавра, где шофера скрестили с автомобилем. Или заниматься реализацией в классе-наследнике интерфейсов, описывающих взаимодействие автомобиля с шофером и измерение/задание скорости.

Но у композиции имеется заметный недостаток: для получившегося класса имеется существенное ограничение при использовании полиморфизма. Ведь он не является наследником классов `Driver` и `Speed` . Поэтому полиморфный код, написанный для объектов типа `Driver` и `Speed` , для объектов типа `MovingCar` работать не будет. И хотя он будет работать для соответствующих полей `movingCar.driver` и `movingCar.speed` , это не всегда помогает. Например, если *объект* должен помещаться в *список*. Тем не менее часто использование композиции является гораздо более удачным решением, чем *множественное наследование*.

Таким образом, сочетание *множественного наследования* интерфейсов и композиции в подавляющем большинстве случаев является полноценной альтернативой *множественному наследованию* классов.

Краткие итоги

Интерфейсы используются для написания полиморфного кода для классов, лежащих в различных, никак не связанных друг с другом иерархиях.

Интерфейсы описываются аналогично абстрактным классам. Так же, как абстрактные классы, они не могут иметь экземпляров. Но, в отличие от *абстрактных классов*, интерфейсы не могут иметь полей данных (за исключением констант), а также реализации никаких своих методов.

Интерфейс определяет методы, которые должны быть реализованы классом-наследником этого интерфейса.

Хотя экземпляров типа интерфейс не бывает, могут существовать переменные типа интерфейс. Такая переменная – это ссылка. Она дает возможность ссылаться на объект, чей класс реализует данный интерфейс.

С помощью переменной типа интерфейс разрешается вызывать только методы, декларированные в данном интерфейсе, а не любые методы данного объекта.

Композиция – это описание объекта как состоящего из других объектов (отношение *агрегации*, или включения как составной части) или находящегося с ними в отношении ассоциации (объединения независимых объектов). Композиция позволяет объединять отдельные части в единую более сложную систему.

Наследование характеризуется отношением "is-a" ("это есть", "является"), а композиция – отношением "has-a" ("имеет в своем составе", "состоит из") и "use-a" ("использует").

Сочетание *множественного наследования* интерфейсов и композиции в подавляющем большинстве случаев является полноценной альтернативой *множественному наследованию* классов.

Типичные ошибки:

После настройки ссылки, хранящейся в переменной типа интерфейс, на объект какого-либо класса, реализующего этот интерфейс, пытаются вызвать поле или метод этого объекта, не объявленные в интерфейсе. Для такого вызова требуется приведение типа, причем до него рекомендуется проверить соответствие объекта этому типу.

Задания

Написать реализацию класса `Square` – наследника `ScalableFigure` . Класс должен располагаться в пакете `AdditionalFigures` .

В пакете `AdditionalFigures` задать интерфейс `IScalable` .

В качестве класса, реализующего этот интерфейс, написать абстрактный класс `StretchableFigure`. Класс должен располагаться в пакете `AdditionalFigures`.

Написать реализацию класса `Rectangle` – наследника `StretchableFigure`. Класс должен располагаться в пакете `AdditionalFigures`.

Написать приложение, в котором в зависимости от выбранной радиокнопки создается и отрисовывается на панели в произвольном месте, не выходящем за пределы панели, точка, окружность, квадрат или прямоугольник. По нажатию на кнопки **"Создать объект"**, **"show"**, **"hide"**, **"moveTo"** должны выполняться соответствующие методы для последнего созданного объекта.

Усложнить копию данного приложения, добавив на форму компонент с прокручивающимся или выпадающим списком с именами объектов. Имя объекта должно состоять из имени, соответствующего типу, и порядкового номера (`dot1`, `circle3` и т.п.). По нажатию на кнопки **"Создать объект"**, **"show"**, **"hide"**, **"moveTo"** должны выполняться соответствующие методы для объекта, выделенного в списке.

Добавить кнопки **"Изменить размер"** и **"Растянуть объект"**. В случае, если объект поддерживает интерфейс `ScalableFigure`, по нажатию первой из них он должен менять размер. Если он поддерживает интерфейс , по нажатию второй он должен растягиваться или сплющиваться в зависимости от значения в соответствующем пункте ввода.

В пакете `AdditionalFigures` написать интерфейс `IBordered`, обеспечивающий поддержку методов, необходимых для рисования границы (`border`) заданной ширины и цвета вокруг графического объекта. Реализовать этот интерфейс в классах `BorderedCircle`, `BorderedSquare`, `BorderedRectangle`.

Внимание! Если Вы увидите ошибку на нашем сайте, выделите её и нажмите Ctrl+Enter.