Оглавление

Объектные переменные суперкласса и динамическое управление методами

Еще раз отметим важное и интересное свойство объектных переменных суперкласса. Они могут ссылаться на объекты подкласса.

Напомним, что объектная переменная — это переменная, значением которой является ссылка на объект соответствующего класса, то есть фактически та переменная, которую мы отождествляем с объектом. Объектная переменная объявляется так же, как обычная переменная базового типа, с той лишь разницей, что в качестве типа переменной указывается имя класса. Создается же объект с помощью оператора new и конструктора класса. Сказанное означает, что ссылку на объект подкласса (объект, созданный конструктором подкласса) можно присвоить в качестве значения объектной переменной суперкласса (в качестве типа переменной указав имя суперкласса).

Важное ограничение состоит в том, что через объектную переменную суперкласса можно ссылаться только на те члены подкласса, которые наследуются из суперкласса или переопределяются в подклассе.

```
//Объектная переменная суперкласса ссылается на объект подкласса
class ClassA{
double Re;
void set(double x){
Re=x;}
void show(){ System.out.println("Класс A:");
System.out.println("Поле Re: "+Re);}
}
class ClassB extends ClassA{
double Im;
void set(double x,double y){ Re=x;
Im=y;}
void show(){ System.out.println("Класс B:");
System.out.println("Поле Re: "+Re); System.out.println("Поле Im:
"+Im);}
}
```

```
class SuperRefs{
public static void main(String[] args){
ClassA objA;
ClassB objB=new ClassB(); objA=objB;
objB.set(1,5);
objB.show();
objA.set(-10);
objA.show();
}
}
```

В данном случае описывается суперкласс ClassA, на основе которого создается подкласс ClassB. В суперклассе ClassA объявлено поле double Re и методы set() и show(). Метод show() не имеет аргументов и выводит сообщение с названием класса (буквы-идентификатора класса) и значением поля Re. Метод set() имеет один аргумент, который присваивается в качестве значения полю Re.

Поле Re наследуется в классе ClassB. В этом классе также описывается поле double Im. Метод set() перегружается так, чтобы иметь два аргумента — значения полей Re и Im. Перегружается и метод show(), чтобы выводить на экран значения двух полей.

В главном методе программы командой ClassA objA объявляется объектная переменная objA класса ClassA. Командой ClassB objB=new ClassB() создается объект класса ClassB, и ссылка на этот объект присваивается в качестве значения объектной переменной objB класса ClassB. Затем командой objA=objB ссылка на тот же объект присваивается в качестве значения объектной переменной objA. Таким образом, в результате и объектная переменная objA, и объектная переменная objB ссылаются на один и тот же объект. То есть переменных две, а объект один. Тем не менее ссылка на объект через переменную objA является «ограниченной» — через нее можно обращаться не ко всем членам объекта класса ClassB.

Командой objB.set(1,5) полям Re и Im объекта присваиваются значения 1 и 5 соответственно. Командой objB.show() значения полей объекта выводятся на экран. Для этого вызывается версия метода show(), описанная в классе ClassB. Командой objA.set(-10) меняется значение поля Re. Для этого вызывается версия метода set(), описанная в классе ClassA и наследуемая в классе ClassB. Вызвать через объектную переменную objA версию метода set() с двумя аргументами не получится — эта версия не описана в классе ClassB, поэтому через объектную переменную суперкласса версия метода недоступна. Однако командой objA.show() можно вызвать переопределенный в классе ClassB метод show(). Результат выполнения программы следующий:

Класс B: Поле Re: 1.0
Поле Im: 5.0 Класс B:
Поле Re: -10.0
Поле Im: 5.0

Отметим также, что в силу отмеченных особенностей ссылки на объект подкласса через объектную переменную суперкласса через переменную objA можно обратиться к полю Re объекта подкласса, но нельзя обратиться к полю Im.

Хотя описанная возможность ссылаться на объекты подклассов через объектные переменные суперклассов может показаться не очень полезной, она открывает ряд перспективных технологий, в том числе и динамическое управление методами.

Динамическое управление методами базируется на том, что выбор варианта перегруженного метода определяется не типом объектной ссылки, а типом объекта, причем на этапе не компиляции, а выполнения программы. С подобной ситуацией мы встречались в предыдущем примере, когда при ссылке на метод show() через объектную переменную objA суперкласса ClassA вызывалась переопределенная версия метода из подкласса ClassB, то есть версия, описанная в классе объекта, а не в классе объектной переменной. Рассмотрим еще один пример:

```
// Динамическое управление методами
class A{ void show(){
System.out.println("Класс A");}
}
class B extends A{ void show(){
System.out.println("Класс B");}
}
class C extends A{ void show(){
System.out.println("Класс C");}
}
class Dispatch{
public static void main(String args[]){
A a=new A();
B b=new B();
C c=new C();
A ref;
ref=a;
ref.show();
ref=b;
ref.show();
ref=c;
ref.show();
}
}
```

В классе A описан метод show(), действие которого сводится к выводу на экран сообщения Класс A. В каждом из классов B и C этот метод переопределяется. Версия метода show() из класса B выводит сообщение Класс B, а версия этого же метода из класса C — сообщение Класс C.

В главном методе программы создаются объекты a, b и c соответственно классов A, B и C, а также объявляется объектная переменная ref класса A. Далее этой объектной переменной последовательно в качестве значений присваиваются ссылки на объекты a, b и c (командами ref=a, ref=b и ref=c).

Поскольку класс A является суперклассом и для класса B, и для класса C, данные операции возможны. Причем после каждого такого присваивания через объектную переменную ref командой ref.show() вызывается метод show(). Результат выполнения программы имеет вид:

Класс A
Класс B
Класс C

Мы видим, что хотя формально во всех трех случаях команда вызова метода show() одна и та же (команда ref.show()), результат разный в зависимости от того, на какой объект в данный момент ссылается объектная переменная ref.

Абстрактные классы

В Java существуют такие понятия, как абстрактный метод и абстрактный класс.

Под абстрактным методом подразумевают метод, тело которого в классе не объявлено, а есть только сигнатура (тип результата, имя и список аргументов). Перед таким абстрактным методом указывается идентификатор abstract, а заканчивается описание сигнатуры метода в классе традиционно — точкой с запятой.

Класс, который содержит хотя бы один абстрактный метод, называется абстрактным. Описание абстрактного класса начинается с ключевого слова abstract.

Абстрактный класс в силу очевидных причин не может использоваться для создания объектов. Поэтому абстрактные классы являются суперклассами для подклассов. При этом в подклассе абстрактные методы абстрактного суперкласса должны быть определены в явном виде (иначе подкласс тоже будет абстрактным). Пример использования абстрактного класса приведен в листинге:

```
// Абстрактный суперкласс:
abstract class A{
// Абстрактный метод:
abstract void callme();
// Неабстрактный метод:
void callme2(){
System.out.println("Второй метод");}
}
// Подкласс:
class B extends A{
// Определение наследуемого абстрактного метода:
void callme(){
System.out.println("Первый метод");}
}

class AbstDemo{
```

```
public static void main(String args[]){
// Объект подкласса:
B obj=new B();
obj.callme();
obj.callme2();}
}
```

Пример достаточно простой: описывается абстрактный суперкласс A, на основе которого затем создается подкласс B. Суперкласс A содержит абстрактный метод call() и обычный (неабстрактный) метод callme2(). Оба метод наследуются в классе B. Но поскольку метод call() абстрактный, то он описан в классе B.

Методом call() выводится сообщение Первый метод, а методом callme2() — сообщение Второй метод. В главном методе программы создается объект подкласса и последовательно вызываются оба метода. В результате получаем сообщения:

Первый метод
Второй метод

Что касается практического использования абстрактных классов, то обычно они бывают полезны при создании сложных иерархий классов. В этом случае абстрактный класс, находящийся в вершине иерархии, служит своеобразным шаблоном, определяющим, что должно быть в подклассах. Конкретная же реализация методов выносится в подклассы. Такой подход, кроме прочего, нередко позволяет избежать ошибок, поскольку будь у суперкласса только неабстрактные наследуемые методы, было бы сложнее отслеживать процесс их переопределения в суперклассах. В то же время, если не определить в подклассе абстрактный метод, при компиляции появится ошибка.

Отметим еще одно немаловажное обстоятельство, которое касается наследования вообще. В некоторых случаях необходимо защитить метод от возможного переопределения в подклассе. Для этого при описании метода в его сигнатуре указывается ключевое слово final. Если это ключевое слово включить в сигнатуру класса, этот класс будет защищен от наследования — на его основе нельзя будет создать подкласс. Третий способ использования ключевого слова final касается описания полей (переменных). В этом случае оно означает запрет на изменение значения поля, то есть фактически означает определение константы.

Интерфейсы

Достаточно часто требуется совмещать в объекте поведение, характерное для двух или более независимых иерархий. Но еще чаще необходимо 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 для объектов из разных классов нужно 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 могла бы ссылаться на экземпляры из разных иерархий, т. к. она должна иметь тип, совместимый с базовыми классами этих иерархий.

В C++ для решения таких задач используется *множественное наследование*. Оно означает, что у класса может быть не один непосредственный прародитель, а два или более. В этом случае совместимость с классами из разных иерархий обеспечивается созданием класса, наследующего от необходимого числа классов-прародителей.

Но при множественном наследовании классов возникает ряд трудноразрешимых проблем, поэтому в Java оно не поддерживается. Основные причины отказа от множественного наследования классов: наследование ненужных полей и методов, а также конфликты совпадающих имен из разных ветвей наследования.

В частности, существует так называемое *ромбовидное наследование*, когда у класса A имеются 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 для преодоления указанных трудностей. Например, возможность различать имена полей и методов, доставшихся от разных прародителей, с помощью дополнительного указания имени класса, из которого берется поле или метод.

А проблемы ромбовидного наследования в значительной степени снимаются применением так называемых виртуальных классов, благодаря чему при ромбовидном наследовании потомкам достается только один комплект членов класса A. Но в результате заметно усложняется логика работы с классами и объектами, что вызывает логические ошибки, не отслеживаемые компилятором. Поэтому в Java реализовано *множественное наследование с помощью интерфейсов*, лишенное практически всех указанных недостатков.

Интерфейсы — важнейшие элементы языков программирования, применяемые 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 самостоятельно. В настоящее время она фактически стала основой объектного программирования в Java, заменив концепцию множественного наследования, используемого в C++.

Интерфейсы являются специальной разновидностью полностью абстрактных классов (т. е. 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 тоже 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, все коллекции обладают методами add, addAll, clear, contains и рядом других, это обеспечивается тем, что их классы являются наследниками интерфейса Collection. Аналогично, все классы итераторов — наследники интерфейса Iterator и т. д.

Имеется еще одно применение интерфейсов, которое появилось в JDK 5, — это использование их в качестве метки, маркера (marker). Можно пометить ряд методов с помощью конструкции @ИмяИнтерфейса, чтобы компилятор мог их обнаружить и задействовать в определенных целях. Например, методы, помеченные спецификатором @Action, где Action — интерфейс, заданный в пакете application, могут играть роль обработчиков событий для объекта action.

Декларация интерфейса очень похожа на декларацию класса:

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

У интерфейсов может быть только два варианта видимости. Если в качестве модификатора видимости задано слово public, то интерфейс общедоступный. Если же модификатор отсутствует, то интерфейсу обеспечивается видимость по умолчанию — пакетная. В списке прародителей, расширением которых является данный интерфейс, указываются прародительские интерфейсы ИмяИнтерфейса1, ИмяИнтерфейса2 и т. д. Если этот список пуст, то интерфейс не имеет прародителя.

Для имен интерфейсов в Java нет специальных правил, за исключением того, что их имена (как и других объектных типов) принято начинать с заглавной буквы.

Объявление константы осуществляется почти так же, как в классе:

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

Необязательным модификатором видимости может быть слово public. Либо модификатор должен отсутствовать, но при этом видимость также считается public, а не пакетной. Еще одно отличие от декларации в классе: при задании полей в интерфейсе все они автоматически считаются окончательными (модификатор final), т. е. без права изменения, да к тому же являющимися переменными класса (модификатор static). Сами модификаторы static и final при этом ставить не нужно.

Декларация метода в интерфейсе очень похожа 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, можно использовать слово public. При отсутствии модификатора видимость также считается public, а не пакетной. В списке исключений через запятую перечисляют типы проверяемых исключений (потомки Exception), которые может возбуждать метод. Часть throws необязательная. При задании в интерфейсе все методы автоматически считаются общедоступными (public) и абстрактными (abstract).

Следующий листинг иллюстрирует пример задания интерфейса:

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

Практическое использование интерфейса подразумевает его реализацию. Эта процедура напоминает наследование абстрактных классов. Реализуется интерфейс в классе. Класс, который реализует интерфейс, должен содержать описание всех методов интерфейса. Методы интерфейса при реализации описываются как открытые. Один и тот же класс может реализовать одновременно несколько интерфейсов, равно как один и тот же интерфейс может реализовываться несколькими классами.

Для реализации интерфейса в классе в сигнатуре заголовка класса указывается инструкция implements(). С учетом того, что реализующий интерфейс класс может одновременно наследовать еще и суперкласс, общий синтаксис объявления класса, который наследует суперкласс и реализует несколько интерфейсов, имеет следующий вид:

```
class имя [extends суперкласс] implements
интерфейс1,интерфейс2,...{
// тело класса
}
```

Если имеет место наследование классов, то после имени класса через ключевое слово extends указывается имя наследуемого суперкласса, затем идет ключевое слово implements. После ключевого слова implements через запятую перечисляются реализуемые в классе интерфейсы, а дальше все стандартно — указывается непосредственно тело класса. Напомним также, что перед ключевым словом class может размещаться ключевое слово public, определяющее уровень доступа класса.

В листинге приведен пример программы, в которой используется интерфейс:

```
// Интерфейс:
interface MyMath{
// Сигнатура метода:
double Sinus(double x);
```

```
// Константа:
double PI=3.14159265358979;
}
// Класс реализует интерфейс:
class MyClass implements MyMath{
// Реализация метода (вычисление синуса):
public double Sinus(double x){
int i,n=1000;
double z=0,q=x;
for(i=1;i<=n;i++){
z+=q;
q*=(-1)*x*x/(2*i)/(2*i+1);}
return z;}
}
class MyMathDemo{
public static void main(String args[]){
MyClass obj=new MyClass();
// Использование константы:
double z=MyClass.PI/6;
// Вызов метода:
System.out.println("sin("+z+")="+obj.Sinus(z));}
}
```

В программе использован интерфейс MyMath, а также классы MyClass и MyMathDemo.

Класс MyClass реализует интерфейс MyMath. Класс MyMathDemo содержит главный метод программы.

В интерфейсе MyMath объявлен метод (только сигнатура) с названием Sinus().

Метод имеет один аргумент типа double и возвращает результат того же типа.

Кроме того, в интерфейсе объявлена константа PI типа double. Это приближенное значение числа π. Обращаем внимание, что хотя поле не содержит ни идентификатора final, ни идентификатора static в своем описании, оно является статической константой. Хотя это и не обязательно, названия полей интерфейсов принято записывать в верхнем регистре.

Класс MyClass, как отмечалось, реализует интерфейс MyMath. Поскольку в интерфейсе объявлен всего один метод, его и следует описать в классе MyClass. В данном случае использован ряд Тейлора для синуса:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

В главном методе программы создается объект obj класса MyClass и объявляется переменная z типа double. Этой переменной присваивается значение π/6. При этом используется константа PI, объявленная в интерфейсе MyMath. Константа статическая и наследуется в классе MyClass, поэтому

ссылка на нее в главном методе программы выглядит как MyClass.PI. В самом же классе к этому полю можно обращаться просто по имени, то есть как PI. Далее для аргумента z вычисляется значение синуса и результат выводится на экран. В итоге получаем сообщение:

sin(0.5235987755982984)=0.4999999999999995

Это достаточно близкий результат к точному значению 1/2.

Интерфейсные ссылки

При создании объектов класса в качестве типа объектной переменной может указываться имя реализованного в классе интерфейса. Другими словами, если класс реализует интерфейс, то ссылку на объект этого класса можно присвоить интерфейсной переменной — переменной, в качестве типа которой указано имя соответствующего интерфейса. Ситуация очень напоминает ту, что рассматривалась при наследовании, когда объектная переменная суперкласса ссылалась на объект подкласса. Как и в случае с объектными ссылками суперкласса, через интерфейсную ссылку можно сослаться не на все члены объекта реализующего интерфейс класса. Доступны только те методы, которые объявлены в соответствующем интерфейсе. С учетом того, что класс может реализовать несколько интерфейсов, а один и тот же интерфейс может быть реализован в разных классах, ситуация представляется достаточно пикантной.

В листинге приведен пример программы, в которой используются интерфейсные ссылки:

```
// Интерфейс:
interface Base{
int F(int n);
}
// Класс A реализует интерфейс Base:
class A implements Base{
// Двойной факториал числа:
public int F(int n){
if(n==1||n==2) return n;
else return n*F(n-2);}
}
// Класс B реализует интерфейс Base:
class B implements Base{
// Факториал числа:
public int F(int n){
if(n<1) return 1;
else return n*F(n-1);}
}
class ImplDemo{
public static void main(String args[]){
// Интерфейсные переменные и создание объектов:
Base refA=new A();
```

```
Base refB=new B();
// Объектные переменные и создание объектов:
A objA=new A();
B objB=new B();
// Проверка работы методов:
System.out.println("1: "+refA.F(5));
System.out.println("2: "+refB.F(5));
System.out.println("3: "+objA.F(5));
System.out.println("4: "+objB.F(5));
// Изменение интерфейсных ссылок:
refA=objB;
refB=objA;
// Проверка результата:
System.out.println("5: "+refA.F(5));
System.out.println("6: "+refB.F(5));}
}
```

В интерфейсе Base объявлен всего один метод с названием F(), целочисленным аргументом и целочисленным результатом. Классы A и B реализуют интерфейс Base, причем каждый по-своему. В классе A метод F() описан так, что им возвращается в качестве результата двойной факториал от целочисленного аргумента (напомним, что по определению двойной факториал числа n есть произведение натуральных чисел до этого числа включительно «через два», то есть $n!! = n(n-2)(n-4)...$). В классе B методом F() вычисляется факториал числа аргумента метода (произведение натуральных чисел от 1 до числа n включительно, то есть $n! = n(n-1)(n-2)...1$). При описании метода F() в обоих классах использована рекурсия.

В главном методе программы командами Base refA=new A() и Base refB=new B() создаются два объекта классов A и B, причем ссылки на эти объекты записываются в интерфейсные переменные refA и refB. В качестве типа этих переменных указано имя интерфейса Base, а сами объекты создаются вызовом конструкторов соответствующих классов.

Затем создаются еще два объекта классов A и B, и ссылки на них записываются в объектные переменные objA и objB соответственно. Для этого используются команды A objA=new A() и B objB=new B().

После этого с помощью объектных и интерфейсных переменных несколько раз вызывается метод F() с аргументом 5. Отметим, что 5!!=15 и 5!=120. Поэтому если вызывается версия метода, описанная в классе A, результатом является число 15, а для версии метода, описанной в классе B, результат есть число 120.

В частности, при вызове метода через переменные objA и refA вызывается версия метода, описанная в классе A, а при вызове метода через переменные objB и refB — версия, описанная в классе B.

После этого командами refA=objB и refB=objA ссылки «меняются местами»: интерфейсная переменная refA ссылается на объект класса B, а

интерфейсная переменная refB — на объект класса A. После этого инструкцией refA.F(5) вызывается версия метода F() из класса B, а инструкцией refB.F(5) — версия метода F(), описанная в классе A. В результате выполнения программы получаем следующее:

```
1: 15
2: 120
3: 15
4: 120
5: 120
```

Рассмотрим пример, в котором один класс реализует несколько интерфейсов:

```
// Первый интерфейс:
interface One{
void setOne(int n);
}
// Второй интерфейс:
interface Two{
void setTwo(int n);
}
// Суперкласс:
class ClassA{
int number;
void show(){
System.out.println("Поле number: "+number);}
}
// Подкласс наследует суперкласс и реализует интерфейсы:
class ClassB extends ClassA implements One,Two{
int value;
// Метод первого интерфейса:
public void setOne(int n){
number=n;}
// Метод второго интерфейса:
public void setTwo(int n){
value=n;}
// Переопределение метода суперкласса:
void show(){
super.show();
System.out.println("Поле value: "+value);}
}
class MoreImplDemo{
public static void main(String[] args){
// Интерфейсные переменные:
One ref1;
Two ref2;
// Создание объекта:
ClassB obj=new ClassB();
```

```
// Интерфейсные ссылки:
ref1=obj;
ref2=obj;
// Вызов методов:
ref1.setOne(10);
ref2.setTwo(-50);
// Проверка результата:
obj.show();}
}
```

Результат выполнения этой программы имеет вид:

Поле number: 10

Поле value: -50

Кратко поясним основные этапы реализации алгоритма. Итак, имеются два интерфейса One и Two, которые реализуются классом ClassB. Кроме того, класс ClassB наследует класс ClassA. В каждом из интерфейсов объявлено по одному методу: в интерфейсе One метод setOne(), а в интерфейсе Two метод setTwo(). Оба метода не возвращают результат и имеют один целочисленный аргумент.

У суперкласса ClassA объявлено поле int number и определен метод show(), который выводит значение поля на экран. При наследовании в подклассе ClassB этот метод переопределяется так, что выводит значения двух полей: наследуемого из суперкласса поля number и поля int value, описанного непосредственно в подклассе.

В классе ClassB методы setOne() и setTwo() реализованы так, что первый метод присваивает значение полю number, второй — полю value.

В главном методе программы создаются две интерфейсные переменные: переменная ref1 типа One и переменная ref2 типа Two. Кроме того, создается объект obj класса ClassB. В качестве значений интерфейсным переменным присваиваются ссылки на объект obj. Это возможно, поскольку класс ClassB реализует интерфейсы One и Two. Однако в силу того же обстоятельства через переменную ref1 можно получить доступ только к методу setOne(), а через переменную ref2 — только к методу setTwo(). Командами ref1.setOne(10) и ref2.setTwo(-50) полям объекта obj присваиваются значения, а командой obj.show() значения полей выводятся на экран.

Расширение интерфейсов

Подобно классам, один интерфейс может наследовать другой интерфейс. В этом случае говорят о расширении интерфейса. Как и при наследовании классов, при расширении интерфейсов указывается ключевое слово extends. Синтаксис реализации расширения интерфейса фактически такой же, как и синтаксис реализации наследования классов:

```
interface имя1 extends имя2{
```

```
// тело интерфейса
}
```

Листинг. Расширение интерфейса

```
// Интерфейс:
interface BaseA{
int FunA(int n);
}
// Расширение интерфейса:
interface BaseB extends BaseA{
int FunB(int n);
}
// Реализация интерфейса:
class MyClass implements BaseB{
public int FunA(int n){
if(n<1) return 1;
else return n*FunA(n-1);}
public int FunB(int n){
if(n==1||n==2) return n;
else return n*FunB(n-2);}
}
class ImplExtDemo{
public static void main(String args[]){
MyClass obj=new MyClass();
System.out.println("1: "+obj.FunA(5));
System.out.println("2: "+obj.FunB(5));}
}
```

В результате выполнения этой программы получаем:

1: 120

2: 15

Что касается самого программного кода, то он достаточно прост. Интерфейс BaseA содержит объявление метода FunA(). У метода целочисленный аргумент и результат метода — тоже целое число. Интерфейс BaseB расширяет (наследует) интерфейс BaseA. Непосредственно в интерфейсе BaseB объявлен метод FunB() с целочисленным результатом и целочисленным аргументом. Учитывая наследуемый из интерфейса BaseA метод FunA(), интерфейс BaseB содержит сигнатуры двух методов FunA() и FunB(). Поэтому в классе MyClass, который реализует интерфейс BaseB, необходимо описать оба эти метода. Метод FunA() описывается как возвращающий в качестве значения факториал числа-аргумента метода, а метод FunB() — как возвращающий в качестве значения двойной факториал числа. В главном методе программы создается объект obj класса MyClass и последовательно вызываются методы FunA() и FunB().

Отличия интерфейсов от классов.

Сформулируем основные отличительные черты интерфейсов:

- не бывает экземпляров типа интерфейс, т. е. экземпляров интерфейсов, реализующих тип интерфейса (что очевидно, если вспомнить, что интерфейс является разновидностью полностью абстрактного класса, а у абстрактных классов экземпляров не бывает);
- список элементов интерфейса может включать только методы и константы, поля данных недопустимы;
- элементы интерфейса всегда имеют тип видимости public (в том числе без явного указания). Не разрешены модификаторы видимости, отличные от public;
- в интерфейсах не бывает конструкторов и деструкторов;
- методы, продекларированные в интерфейсе, являются абстрактными по умолчанию, но разрешается явным образом записывать модификатор abstract; методы не могут иметь модификаторы static, native, synchronized, final, private, protected;
- интерфейс, как и класс, наследует все методы прародителя, однако только на уровне абстракций, без реализации методов (т. е. интерфейс наследует лишь обязательность реализации этих методов в классе, поддерживающем данный интерфейс);
- наследование через интерфейсы может быть множественным. В заголовке интерфейса можно указать, что интерфейс наследуется от одного или нескольких прародительских интерфейсов;
- реализация интерфейса может быть только в классе; если при этом класс не абстрактный, он 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. 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.

При использовании констант из разных интерфейсов необходима квалификация имени константы именем соответствующего интерфейса (аналогично разрешению конфликта имен в случае пакетов). Пример приведен в листинге:

```
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 нельзя наследовать методы, имеющие совпадающие сигнатуры. Если сигнатуры различаются, то проблем нет, и используется перегрузка.

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 (т. е. считать класс A2 наследником класса A1). Действует правило:

"A2 есть A1" (A2 is A1).

Если для нескольких классов A1, B1,... из разных иерархий можно на уровне абстракций выделить общность поведения, то 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 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, то все его потомки также будут наследовать этот интерфейс.

Композиция

Как уже говорилось ранее, наследование затрагивает один из важных аспектов, присущих объектам, — поведение. Причем оно относится не к самим объектам, а к классам. Но имеется и другой аспект, присущий объектам, — внутреннее устройство. При наследовании это обстоятельство скорее скрывается, нежели подчеркивается: наследники должны быть устроены так, чтобы отличие в их устройстве не сказывалось на абстракциях их поведения.

Композиция — это такое устройство объекта, когда он либо состоит из других объектов, предназначенных для того, чтобы быть его частями (отношение агрегации или включения), либо находится с ними в отношении ассоциации (использования независимых объектов).

Таким образом, композиция — это объединение частей в единую систему.

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 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 независимо от всей сложной системы.

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 т. д. до того уровня, когда деталь можно считать единым целым, не включающим в себя другие объекты.

Шофер также является неотъемлемой частью автомобиля, но вряд ли можно считать, что автомобиль состоит из шофера и других частей. Однако можно сказать, что у автомобиля обязательно должен быть шофер, либо нужно 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 (т. е. 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. Так бывает, когда необходимо унаследовать от двух или более классов их поля и методы.

Приведем пример.

Пусть у нас имеются: класс Car (Автомобиль), класс Driver (Шофер) и класс Speed (Скорость), и пусть это совершенно независимые классы. Зададим класс MovingCar (Движущийся автомобиль):

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

Отличительной чертой 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 придется строить гибридный класс-монстр, в котором от родителей Car, Driver и Speed унаследованы поля и методы по механизму множественного наследования (нечто вроде машино-кентавра, где шофера скрестили с автомобилем). Причем в случае композиции не нужно заниматься реализацией в классе-наследнике интерфейсов, описывающих взаимодействие автомобиля с шофером и измерение/задание скорости.

Но у композиции имеется заметный недостаток: для получившегося класса существует ограничение при использовании полиморфизма, т. к. 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 такой код будет функционировать для соответствующих полей movingCar.driver и movingCar.speed, это не всегда помогает (например, если объект должен помещаться в список). Тем не менее, часто композиция — гораздо более удачное решение, чем множественное наследование.

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