Конспект лекций по Java. Занятие 8

В.Фесюнов

1. Модификаторы доступа при наследовании

На 3-ем занятии мы рассматривали описатели ограничения доступа (или, более кратко, модификаторы доступа). Несколько моментов в этом рассмотрении остались за рамками наших возможностей. Теперь, после знакомства с наследованием классов, мы можем доопределить их смысл. Поля и методы классов могут быть описаны как

- **public** — означает, что данный элемент доступен без каких-либо ограничений;
- **private** — доступ разрешен только из данного класса;
- **protected** — доступ разрешен из данного класса и из всех классов-потомков
- *без описателя* — доступ разрешен из всех классов данного пакета

Т.е. `public` -поля и методы доступны без ограничений во всех классах; `private` доступны только в данном классе и недоступны в порожденных классах; `protected` отличаются от `private` тем, что доступны в порожденных классах, а поля и методы без описателя доступны только в классах данного пакета вне зависимости от наследования.

2. Наследование классов — инструмент построения абстракций

```
(Небольшое теоретическое отступление)
```

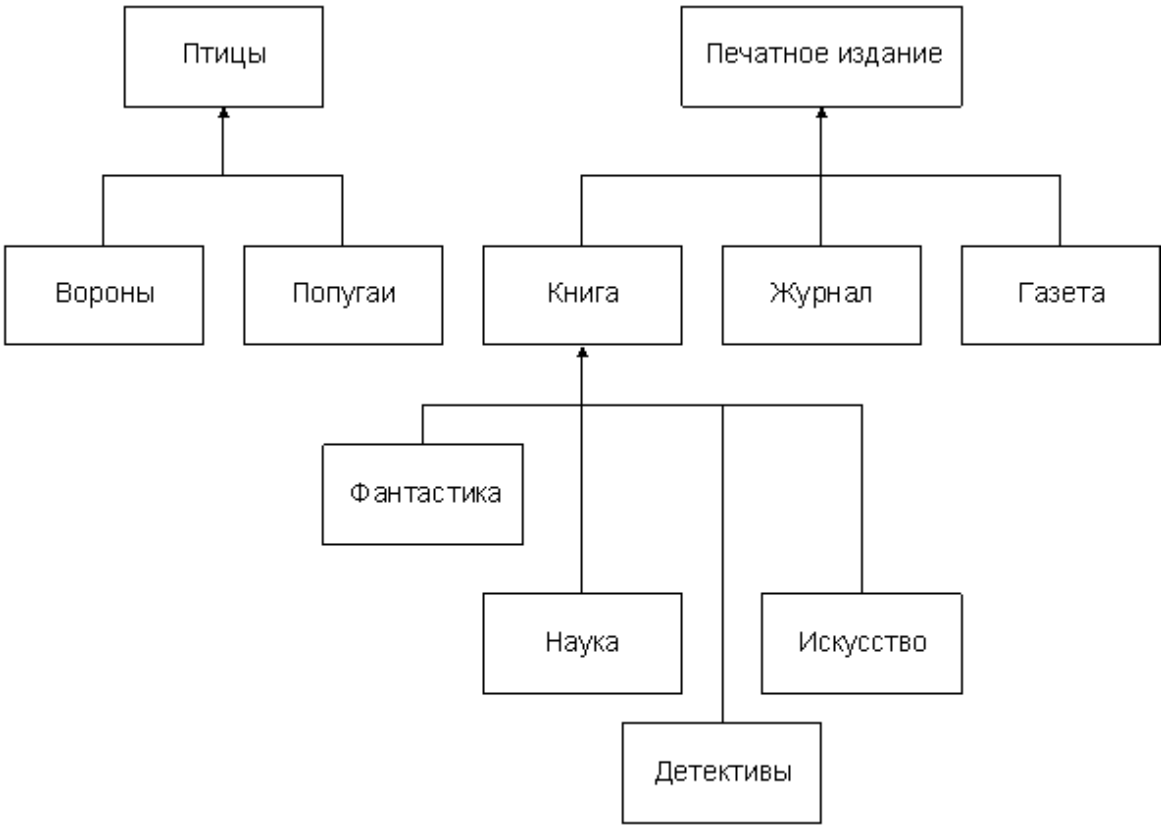
Вместе с другими категориями ООП (полиморфизм, инкапсуляция) наследование служит инструментом построения абстракций. Причем, приемы, применяемые в ООП, не совсем обычны для тех, кто ранее программировал без использования принципов ООП.

Рассмотрим, например, как выглядят возможности наследования с первого взгляда. Наследование дает возможность построить простой класс, потом на его базе более сложный (добавив поля и методы), потом еще более сложный и т.д. Так это выглядит на первый взгляд. Но это, можно сказать, примитивный взгляд на возможности механизма наследования.

В ООП наследование обычно используется иначе и для других целей. Наследование классов — это мощный аппарат построения абстракций. При создании иерархии наследования классов в вершине иерархии расположены наиболее абстрактные сущности, которые все более конкретизируются и специализируются при движении вниз по дереву классов.

Для более предметного обсуждения рассмотрим несколько типичных примеров построения дерева наследования в стиле ООП.

Примеры



Замечание:

В визуальном представлении дерева наследования его изображают сверху вниз - от базовых классов к производным. На основе такого представления формируется и терминология (например, upcasting и downcasting, см. ниже).

Тип "Птицы" включает в себя свойства, характерные для всех птиц. А порожденные от него типы "Вороны" и "Попугаи" добавляют (каждый) какие-то свойства, характерные только для данных видов птиц.

Тип "Печатное издание" хранит наиболее общую информацию, характерную для всех печатных изданий. Например, каждое печатное издание имеет наименование. В типах, порожденных от него, добавляются свойства, характерные только для книг (например — список авторов), журналов (год, номер) или газет (дата).

Приведенные примеры демонстрируют не усложнение, а детализацию и

конкретизацию классов при наследовании. Именно это является основным общепринятым принципом при построении дерева наследования классов: наверху — наиболее общие, абстрактные понятия, и при движении вниз по дереву иерархии они детализируются.

Можно поставить такую абстрактную задачу.

Пусть требуется формировать множества, которые могут включать в себя как элементарные объекты, так и множества того же типа. Набор типов элементарных объектов фиксирован.

Правильное решение состоит в том, что в вершину иерархии мы помещаем класс для самого этого множества, а классы элементарных объектов мы строим как наследники класса множества. Детализация заключается в том, что эти классы — это множества из одного элемента.

В дальнейшем изучении мы постараемся помнить о том, что наследование — это механизм построения на основе более общих, более абстрактных классов новых классов — более специфических, более конкретных.

Продолжим знакомство с возможностями аппарата наследования классов.

3. Преобразования типов (классов) при наследовании

Аппарат наследования классов предусматривает возможности преобразования типов между суперклассом и подклассом. Преобразование типов в каком-то смысле является формальным. Сам объект при таком преобразовании не изменяется, *преобразование относится только к типу ссылки на объект* .

Рассмотрим это на примере.

Пример

```
class A {
   int x;
   . . .
}
```

```
class B extends A {
   int y;
   . . .
}

B b = new B();
A a = b;   // здесь происходит формальное преобразование типа: B => A
```

Различаются два вида преобразований типов — *upcasting* и *downcasting* . Повышающее преобразование (upcasting) — это преобразование от типа порожденного класса (от подкласса) к базовому (суперклассу). Такое преобразование допустимо всегда. На него нет никаких ограничений и для его проведения не требуется применять никаких дополнительных синтаксических конструкций (см. предыдущий пример). Это связано с тем, что объект подкласса всегда в себе содержит как свою часть объект суперкласса.

Понижающее преобразование (downcasting) — это преобразование от суперкласса к подклассу. Такое преобразование имеет ряд ограничений. Во-первых, оно может задаваться только явно при помощи операции преобразования типов, например,

B b1 = (B)a;

Во-вторых, объект, подвергаемый преобразованию, реально должен быть того класса, к которому он преобразуется. Если это не так, то возникает исключение ClassCastException в процессе выполнения программы.

Все это выглядит странно, для тех, кто не знаком с ООП. Действительно, получается, что единственная допустимая ситуация, когда такое преобразование возможно, это когда мы построили объект класса B, где B вляется подклассом A, затем зачем-то преобразовали его к классу A, а потом, опять же непонятно для чего, преобразовали его обратно к классу B.

На самом деле это имеет смысл. Для класса A может быть создан программный код, выполняющий что-то полезное. Он имеет свои методы, и они предполагают работу с объектами класса A.

Потом те, кто желают воспользоваться этим программным кодом, строят подкласс B класса A. Но при работе с программным кодом, разработанным для класса A, приходится объекты класса B преобразовывать к классу A (upcasting), поскольку

программный код для класса A ничего не знает о классе B (и ему подобных).

Получив какие-то результаты от программного кода класса A, нужно опять вернуться к работе с классом B (downcasting). Это один из типичных сценариев, требующих преобразования типов как в одну, так и в другую сторону.

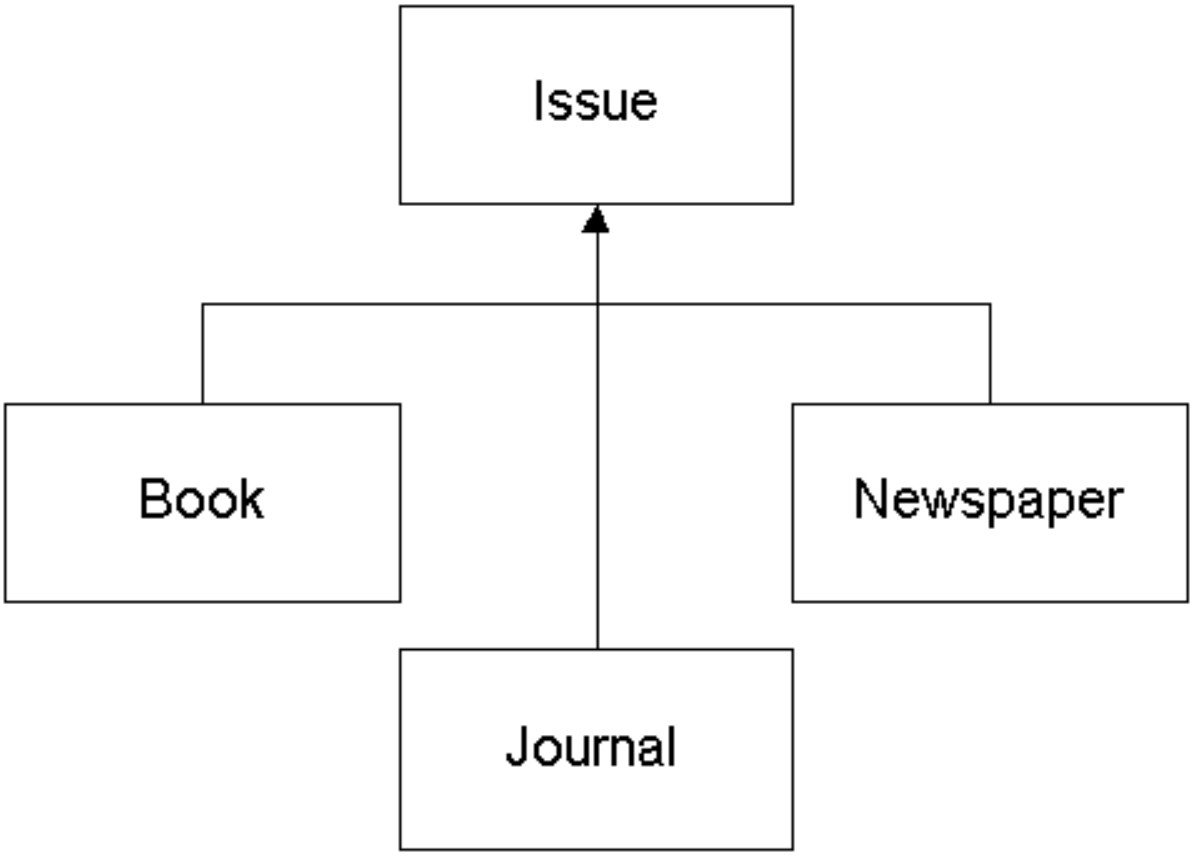
В Java для проверки типа объекта есть операция **instanceof** . Она часто применяется при понижающем (downcasting) преобразовании. Эта операция проверяет, имеет ли ее левый операнд класс, заданный правым операндом.

```
if ( a instanceof B )
    b1 = (B)a;
```

Рассмотрим несколько более содержательный пример, в котором применяются оба вида преобразований, а также операция instanceof.

Пример

Рассмотрим иерархию классов (Issue — печатное издание, Book — книга, Newspaper - газета, Journal — журнал).



Пусть классы Issue и Book реализованы след. образом:

```
public class Issue {
    String name;
    public Issue(String name) {
        this.name = name;
    }
    public void printName(PrintStream out) {
         out.println("Наименование:");
         out.println(name);
    }
    . . .
}

public class Book extends Issue {
```

```
    String authors;
    public Book(String name, String authors) {
        super(name);
        this.authors = authors;
    }

    public void printAuthors(PrintStream out) {
         out.println("Авторы:");
         out.println(authors);
    }
    . . .
}
```

где-то в программе присутствует такой фрагмент

```
Issue[] catalog = new Issue[] {
     new Journal("Play Boy"),
     new Newspaper("Спид Инфо"),
     new Book("Война и мир", "Л.Толстой"), };
. . .
for(int i = 0; i < catalog.length; i++) {
     if ( catalog[i] instanceof Book )
          ((Book) catalog[i]).printAuthors(System.out);
     catalog[i].printName(System.out);
}
```

Рассмотрим его более детально. Здесь порождается каталог (массив печатных изданий), причем каждое из печатных изданий каталога может быть как книгой, так и газетой или журналом. При построении массива выполняется приведение к базовому типу (upcasting). Далее в цикле мы печатаем информацию из каталога. Причем, для книг кроме наименования печатается еще и список авторов. Для этого с использованием операции instanceof проверяется тип печатного издания, а при самой печати списка авторов элемент каталога преобразуется к типу Book. Если этого не сделать, транслятор выдаст ошибку, т.к. метод printAuthors(...) есть только в классе Book, но не в классе Issue.

4. Полиморфизм

В ООП применяется понятие *полиморфизм* .

- Полиморфизм в ООП означает возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования.

Понятие полиморфизма, в свою очередь, опирается на два других понятия: совместное использование (*overloading*) и переопределение (*overriding*).

Рассмотрим их подробнее.

Термин *overloading* можно перевести как перегрузку, доопределение, совместное использование. Мы будем использовать перевод *совместное использование* . Под совместным использованием понимают использование одноименных методов с различным набором параметров. При вызове метода в зависимости от набора параметров выбирается требуемый метод. При этом одноименные методы могут быть как в составе одного класса, так и в разных классах, связанных отношением наследования. Это *статический* полиморфизм методов классов. Примеры совместного использования мы уже встречали ранее. Приведем еще несколько примеров.

```
class X {

   int f() {
      . . .
   }

   void f(int k) {
      . . .
   }
...
}
```

В классе X есть два метода f(...), но с разными типами возвращаемого значения и разными наборами параметров. Тип возвращаемого значения не является определяющим фактором при совместном использовании — при вызове метода транслятору нужно определить, какой из одноименных методов вызывать, а тип возвращаемого значения, в общем случае, не позволяет сделать это однозначно. Поэтому нельзя описать в рамках одного класса два метода с одинаковым набором

параметров и разными типами возвращаемых значений.

```
class Base {

    int f(int k) {
      . . .
    }
    . . .
}

class Derived extends Base {

    int f(String s, int k) {
      . . .
    }
    . . .
}
```

В данном примере представлено совместное использование при наследовании. Класс Derived имеет два метода f(...). Один он наследует от класса Base, другой описан в самом классе Derived.

Понятие *overloading* нужно отличать от понятия *overriding* (задавливание, подавление, переопределение). При переопределении (overriding) методов речь идет только о паре классов — базовом и порожденном. В порожденном классе определяется метод полностью идентичный как по имени, так и по набору параметров тому, что есть в базовом.

Пример

```
class A {
   int x;
   int f(int a) {
      return a+x;
   }
   . . .
}
```

```
class B extends A {
   int y;

   int f(int s) {
       return s*x;
    }
   . . .
}

B b = new B();
A a = b;         // здесь происходит формальное преобразование типа: B => A
int c = a.f(10); // ??? какой из f(...) будет вызван ???
```

Здесь самым интересным моментом является последняя строка. В ней "a" формально имеет тип A, но фактически ссылается на объект класса B. Возникает вопрос, какой из двух совершенно одинаково описанных методов f() будет вызван. Ответ на этот вопрос — B.f().

В Java (как и в других объектно-ориентрованных языках) выполняется вызов метода данного объекта с учетом того, что объект может быть не того же класса, что и ссылка, указывающая на него. Т.е. выполняется вызов метода того класса, к которому реально относится объект.

Это — *динамический* полиморфизм методов. Он называется *поздним связыванием* (dynamic binding, late binding, run-time binding). В C++ соответствующий механизм называется механизмом виртуальных функций.

Рассмотрим содержательный пример использования возможностей, которые дает переопределение методов и позднее связывание. Реализуем классы Issue и Book иначе.

```
public class Issue {
    String name;
    public Issue(String name) {
        this.name = name;
    }
    public void print(PrintStream out) {
         out.println("Наименование:");
         out.println(name);
```

```
    }
    . . .
}

public class Book extends Issue {
    String authors;
    public Book(String name, String authors) {
        super(name);
        this.authors = authors;
    }

    public void print(PrintStream out) {
         out.println("Авторы:");
         out.println(authors);
         super.print(out);       // явный вызов метода базового класса
    }
    . . .
}
```

и переделаем фрагмент, обеспечивающий печать нашего каталога.

```
Issue[] catalog = new Issue[] {
     new Journal("Play Boy"),
     new Newspaper("Спид Инфо"),
     new Book("Война и мир", "Л.Толстой"), };
. . .
for(int i = 0; i < catalog.length; i++) {
     catalog[i].print(System.out);
}
```

В классах Issue и Book вместо двух методов printName(...) и printAuthors(...) теперь один метод print(..). В классе Book метод print(...) переопределяет одноименный метод класса Issue.

- При написании метода print(...) в Book для сокращения кода использован прием явного вызова метода базового класса с использованием ключевого слова **super** . Эту возможность мы рассматривали ранее.

Теперь при печати каталога мы можем не делать специальную проверку для Book.

Нужный метод print(...) класса Book будет вызван автоматически благодаря механизму позднего связывания.

5. Ключевое слово final (Отступление)

В Java есть ключевое слово **final** , используемое как описатель полей, переменных, параметров и методов.

В применении к полям, переменным и параметрам оно означает, что их значение не может быть изменено. Поле или переменная с описателем final должны получить значение при описании, параметр просто не может быть изменен внутри тела метода.

final double pi = 3.14;

Описатель final в сочетании с описателем static позволяют создать константы, т.е. поля, неизменные во всей программе. Так pi логичнее было бы описать так.

static final double pi = 3.14;

Если нужно запретить переопределение (overriding) метода во всех порожденных классах, то этот метод можно описать как final.

Кроме того, ключевое слово final может применяться к классам. Это означает, что данный класс не может быть унаследован другим классом.

6. Абстрактные классы

Класс может представлять собой как бы заготовку, в которой часть методов реализована, а часть — нет. В этом случае в описании класса перед словом class должен стоять описатель **abstact** и при описании нереализованных методов тоже должен использоваться этот описатель.

Пример

```
public abstract class D {
   . . .
   int g1(int s) {
       . . .
```

```
    }

    public abstract void g2(String str);

    . . .
}
```

В классе D метод g1 — это обычный метод, g2 — абстрактный, он содержит только заголовок, но не содержит реализации.

Как видно из примера, тело абстрактного метода отсутствует, сразу после заголовка метода стоит точка с запятой.

Абстрактный класс не может использоваться непосредственно для порождения объектов. Для этого необходимо, используя этот класс как базовый, породить другой класс, в котором нужно определить все абстрактные методы. Тогда можно будет создавать объекты.

С другой стороны не запрещено описывать переменные абстрактного класса. Просто им нужно присваивать ссылки на объекты неабстрактных классов.

- В этой ситуации всегда применяется upcasting.

Возвращаясь к примеру с печатными изданиями, можно отметить, что класс Issue можно было бы реализовать как абстрактный, определив но не реализовав в нем метод print(...). Тогда во всех порожденных классах (Book, Journal, Newspaper) пришлось бы реализовать метод print(...). Фрагмент, печатающий каталог, при этом остался бы прежним.

7. Интерфейсы

Понятие интерфейса чем-то похоже на абстрактный класс. Интерфейс — это полностью абстрактный класс, не содержащий никаких полей, кроме констант (static final - поля).

- Терминология. Класс *наследует* другой класс, но, класс *удовлетворяет* интерфейсу, класс *реализует, выполняет* интерфейс.

Существует, однако, серьезное отличие интерфейсов от классов вообще и от

абстрактных классов, в частности. Интерфейсы допускают множественное наследование. Т.е. один класс может удовлетворять нескольким интерфейсам сразу.

- Это связано с тем, что интерфейсы не порождают проблем с множественным наследованием, поскольку они не содержат полей.

Синтаксис:

```
public interface XXX {
   . . .
   int f(String s);
}
```

Это описание интерфейса XXX. Внутри скобок могут находиться только описания методов (без реализации) и описания констант (static final — полей). В данном случае интерфейс XXX содержит, в частности метод f(...).

```
public class R implements Serializable, XXX {
    . . .
}
```

Класс R реализует интерфейсы Serializable и XXX.

Внутри класса, реализующего некоторый интерфейс, должны быть реализованы все методы, описанные в этом интерфейсе. Поскольку XXX имеет метод f(...), то в классе R он должен быть реализован:

```
public class R implements Serializable, XXX {
    . . .
   public int f(String s) {
      ...
   }
...
}
```

Обратите внимание, что в интерфейсе f(...) описан без описателя public, а в классе R с описателем public. Дело в том, что все методы интерфейса по умолчанию считаются

public, так что этот описатель там можно опустить. А в классе R мы обязаны его использовать явно.

Еще одним общим моментом интерфейсов и абстрактных классов является то, что хотя и нельзя создавать объекты интерфейсов, но можно описывать переменные типа интерфейсов.

- Интерфейсы широко используются при написании различных стандартов. Стандарт определяет, в частности, набор каких-то интерфейсов. И, кроме того, он содержит вербальное описание семантики этих интерфейсов. После этого, прикладные разработчики могут писать программы, используя интерфейсы стандарта. А фирмы-разработчики могут разрабатывать конкретные реализации этих стандартов. При внедрении (deployment) прикладного программного обеспечения можно взять продукт любой фирмы-разработчика, реализующий данный стандарт (на практике, конечно, все несколько сложнее).

В последующем изучении мы редко будем строить абстрактные классы и интерфейсы, но очень часто будем использовать таковые из стандартной библиотеки Java.