Программирование на языке Java. Пакеты и интерфейсы

Картузов А.В.

Пакет (package)—это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы можете создать класс List, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть если бы кто-нибудь еще создал класс с именем List.

Интерфейс—это явно указанная спецификация набора методов, которые должны быть представлены в классе, который реализует эту спецификацию. Реализация же этих методов в интерфейсе отсутствует. Подобно абстрактным классам интерфейсы обладают замечательным дополнительным свойством—их можно многократно наследовать. Конкретный класс может быть наследником лишь одного суперкласса, но зато в нем может быть реализовано неограниченное число интерфейсов.

1. Пакеты

Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакеты—это механизм, который служит как для работы с пространством имен, так и для ограничения видимости. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор в наших примерах использовали только одну. Ниже приведена общая форма исходного файла Java.

- одиночный оператор package (необязателен)
- любое количество операторов import (необязательны)
- одиночное объявление открытого (public) класса

- любое количество закрытых (private) классов пакета (необязательны)

1.1. Оператор package

Первое, что может появиться в исходном файле Java—это оператор package, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. Пакеты задают набор раздельных пространств имен, в которых хранятся имена классов. Если оператор package не указан, классы попадают в безымянное пространство имен, используемое по умолчанию. Если вы объявляете класс, как принадлежащий определенному пакету, например,

```
package java.awt.image;
```

то и исходный код этого класса должен храниться в каталоге java/awt/image.

Замечание:

Каталог, который транслятор Java будет рассматривать, как корневой для иерархии пакетов, можно задавать с помощью переменной окружения CLASSPATH. С помощью этой переменной можно задать несколько корневых каталогов для иерархии пакетов (через ; как в обычном PATH).

1.2. Трансляция классов в пакетах

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Представьте себе, что вы написали класс с именем PackTest в пакете test. Вы создаете каталог test, помещаете в этот каталог файл PackTest.Java и транслируете. Пока—все в порядке. Однако при попытке запустить его вы получаете от интерпретатора сообщение <can't find class PackTest> (<Не могу найти класс PackTest>). Ваш новый класс теперь хранится в пакете с именем test, так что теперь надо указывать всю иерархию пакетов, разделяя их имена точками—test.PackTest. Кроме того Вам надо либо подняться на уровень выше в иерархии каталогов и снова набрать <java test.PackTest>, либо внести в переменную CLASSPATH каталог, который является вершиной иерархии разрабатываемых вами классов.

1.3. Оператор import

После оператора package, но до любого определения классов в исходном Java-файле, может присутствовать список операторов import. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора import такова:

```
import пакет1 [.пакет2].(имякласса|*);
```

Здесь пакет1—имя пакета верхнего уровня, пакет2—это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса. В приведенном ниже фрагменте кода показаны обе формы использования оператора import :

```
import java.util.Date
import java.io.*;
```

Замечание:

Но использовать без нужды форму записи оператора import с использованием звездочки не рекомендуется, т.к. это может значительно увеличить время трансляции кода (на скорость работы и размер программы это не влияет).

Все встроенные в Java классы, которые входят в комплект поставки, хранятся в пакете с именем java. Базовые функции языка хранятся во вложенном пакете java.lang. Весь этот пакет автоматически импортируется транслятором во все программы. Это эквивалентно размещению в начале каждой программы оператора

```
import java.lang.*;
```

Если в двух пакетах, подключаемых с помощью формы оператора im-port со звездочкой, есть классы с одинаковыми именами, однако вы их не используете, транслятор не отреагирует. А вот при попытке использовать такой класс, вы сразу получите сообщение об ошибке, и вам придется переписать операторы import, чтобы

явно указать, класс какого пакета вы имеете ввиду.

```
class MyDate extends Java.util.Date { }
```

1.4. Ограничение доступа

Java предоставляет несколько уровней защиты, обеспечивающих возможность тонкой настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать еще с четырьмя категориями видимости между элементами классов :

- Подклассы в том же пакете.
- Не подклассы в том же пакете.
- Подклассы в различных пакетах.
- Классы, которые не являются подклассами и не входят в тот же пакет.

В языке Java имеется три уровня доступа, определяемых ключевыми словами: private (закрытый), public (открытый) и protected (защищенный), которые употребляются в различных комбинациях. Содержимое ячеек таблицы определяет доступность переменной с данной комбинацией модификаторов (столбец) из указанного места (строка).

	private	модификатор отсутствует	private protected	protected	public
тот же класс	да	да	Да	да	да
подкласс в том же пакете	нет	да	Да	да	да
независимый класс в том же пакете	нет	да	Нет	да	да
подкласс в другом пакете	нет	нет	Да	да	да
независимый класс в другом пакете	нет	нет	Нет	нет	да

На первый взгляд все это может показаться чрезмерно сложным, но есть несколько

правил, которые помогут вам разобраться. Элемент, объявленный public, доступен из любого места. Все, что объявлено private, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, вам нужно объявить такой элемент protected. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того, находятся ли они в данном пакете или нет—используйте комбинацию private protected.

Ниже приведен довольно длинный пример, в котором представлены все допустимые комбинации модификаторов уровня доступа. В исходном коде первого пакета определяется три класса: Protection, Derived и SamePackage. В первом из этих классов определено пять целых переменных—по одной на каждую из возможных комбинаций уровня доступа. Переменной n приписан уровень доступа по умолчанию, n_pri—уровень private, n_pro—protected, n_pripro—private protected и n_pub—public. Во всех остальных классах мы пытаемся использовать переменные первого класса. Те строки кода, которые из-за ограничения доступа привели бы к ошибкам при трансляции, закомментированы с помощью однострочных комментариев (//)—перед каждой указано, откуда доступ при такой комбинации модификаторов был бы возможен. Второй класс—Derived—является подклассом класса Protection и расположен в том же пакете p1. Поэтому ему доступны все перечисленные переменные за исключением n_pri. Третий класс, SamePackage, расположен в том же пакете, но при этом не является подклассом Protection. По этой причине для него недоступна не только переменная n_pri, но и n_pripro, уровень доступа которой—private protected.

```
package p1;
    public class Protection {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        private protected int n_pripro = 4;
        public int n_pub = 5;
        public Protection() {
            System.out.println("base constructor");
```

```
            System.out.println("n="+n);
            System.out.println("n_pri="+n_pri);
            System.out.println("n_pro="+n_pro);
            System.out.println("n_pripro="+n_pripro);
            System.out.println("n_pub="+n_pub);
            }
        }

class Derived extends Protection {
    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);
        // только в классе
        // System.out.println("n_pri="+n_pri);
        System.out.println("n_pro="+n_pro);
        System.out.println("n_pripro="+n_pripro);
        System.out.println("n_pub="+n_pub);
        }
    }

class  SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n="+p.n);
        // только в классе
        // System.out.println("n_pri="+p.n_pri);
        System.out.println("n_pro="+p.n_pro);
        // только в классе и подклассе
        // System.out.println("n_pripro="+p.n_pripro):
        System.out.println("n_pub="+p.n_pub):
        }
    }
```

2. Интерфейсы

Интерфейсы Java созданы для поддержки динамического выбора (resolution) методов во время выполнения программы. Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет переменных представителей, а в объявлениях методов

отсутствует реализация. Класс может иметь любое количество интерфейсов. Все, что нужно сделать—это реализовать в классе полный набор методов всех интерфейсов. Сигнатуры таких методов класса должны точно совпадать с сигнатурами методов реализуемого в этом классе интерфейса. Интерфейсы обладают своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает возможность реализовать один и тот же интерфейс в различных классах, никак не связанных по линии иерархии классового наследования. Именно в этом и проявляется главная сила интерфейсов. Интерфейсы являются аналогом механизма множественного наследования в C++, но использовать их намного легче.

2.1. Оператор interface

Определение интерфейса сходно с определением класса, отличие состоит в том, что в интерфейсе отсутствуют объявления данных и конструкторов. Общая форма интерфейса приведена ниже:

```
interface имя {
    тип_результата имя_метода1(список параметров);
    тип имя_final1-переменной = значение;
    }
```

Обратите внимание—у объявляемых в интерфейсе методов отсутствуют операторы тела. Объявление методов завершается символом ; (точка с запятой). В интерфейсе можно объявлять и переменные, при этом они неявно объявляются final—переменными. Это означает, что класс реализации не может изменять их значения. Кроме того, при объявлении переменных в интерфейсе их обязательно нужно инициализировать константными значениями. Ниже приведен пример определения интерфейса, содержащего единственный метод с именем callback и одним параметром типа int.

```
interface Callback {
    void callback(int param);
    }
```

2.2. Оператор implements

Оператор implements—это дополнение к определению класса, реализующего некоторый интерфейс(ы).

```
class имя_класса [extends суперкласс]
[implements интерфейс0 [, интерфейс1...]] { тело класса }
```

Если в классе реализуется несколько интерфейсов, то их имена разделяются запятыми. Ниже приведен пример класса, в котором реализуется определенный нами интерфейс:

```
class Client implements Callback {
    void callback(int p) {
    System.out.println("callback called with " + p);
    }
}
```

В очередном примере метод callback интерфейса, определенного ранее, вызывается через переменную—ссылку на интерфейс:

```
class TestIface {
    public static void main(String args[]) {
        Callback c = new client();
        c.callback(42);
        }
    }
```

Ниже приведен результат работы программы:

```
C:\> Java TestIface
callback called with 42
```

2.3. Переменные в интерфейсах

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант. В том случае, когда вы реализуете в классе какойлибо интерфейс, все имена переменных этого интерфейса будут видимы в классе как константы. Это аналогично использованию файлов-заголовков для задания в C и C++ констант с помощью директив #define или ключевого слова const в Pascal / Delphi.

Если интерфейс не включает в себя методы, то любой класс, объявляемый реализацией

этого интерфейса, может вообще ничего не реализовывать. Для импорта констант в пространство имен класса предпочтительнее использовать переменные с модификатором final. В приведенном ниже примере проиллюстрировано использование интерфейса для совместно используемых констант.

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
    }

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)  return NO; // 30%
        else if (prob < 60) return YES; // 30%
        else if (prob < 75) return LATER; // 15%
        else if (prob < 98) return SOON; // 13%
        else return NEVER; // 2%
        }
    }

class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
```

```
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.priniln("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
            }
        }

public static void main(String args[]) {
    Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.askO);
        answer(q.ask());
        }
    }
```

Обратите внимание на то, что результаты при разных запусках программы отличаются, поскольку в ней используется класс генерации случайных чисел Random пакета java.util. Описание этого пакета приведено в главе 12.

```
C:\> Java AskMe
    Later
    Scon
    No
    Yes
```

3. Использование пакетов

Теперь вы обладаете полной информацией для создания собственных пакетов классов. Легко понимаемые интерфейсы позволят другим программистам использовать ваш код для самых различных целей. Инструменты, которые вы приобрели, изучив эту и предыдущую главы, должны вам помочь при разработке любых объектно-ориентированных приложений. В дальнейшем вы познакомитесь с

некоторыми важными специфическими свойствами Java, которые представлены в виде классов в пакете java.lang. В трех последующих главах вы освоите работу с текстовыми строками, параллельное программирование и обработку исключительных ситуаций.