ПАКЕТЫ И ИНТЕРФЕЙСЫ

Пакеты. Определение пакета

- *Пакеты* — это контейнеры для классов, которые используются для сохранения пространства имен классов разделенным на именованные области.
- Например, вы имеете возможность создать класс с именем List, который можно хранить в вашем собственном пакете, не опасаясь, что он столкнется с некоторым другим классом с именем List, хранящимся где-то в другом месте.
- Пакеты сохраняются иерархическим способом и явно импортируются в определения новых классов.

Пакет и подпакет

- Разработчики Java включили в язык дополнительную конструкцию — пакеты (packages). Все классы Java распределяются по пакетам. Кроме классов пакеты могут включать в себя интерфейсы и вложенные подпакеты (subpackages). Образуется древовидная структура пакетов и подпакетов.
- Эта структура в точности отображается на структуру файловой системы. Все файлы с расширением class (содержащие байт-коды), образующие пакет, хранятся в одном каталоге файловой системы. Подпакеты собраны в подкаталоги этого каталога.
- Каждый пакет образует одно пространство имен (namespace). Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы.

Пакет и подпакет

- Таким образом, ни один класс, интерфейс или подпакет не может оказаться сразу в двух пакетах. Если надо использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: пакет.класс . Такое уточненное имя называется полным именем класса (fully qualified name).
- Все эти правила совпадают с правилами хранения файлов и подкаталогов в каталогах.
- Пакетами пользуются еще и для того, чтобы добавить к уже имеющимся правам доступа к членам класса private, protected и public еще один, "пакетный" уровень доступа.
- Если член класса не отмечен ни одним из модификаторов private, protected, public, то, по умолчанию, к нему осуществляется пакетный доступ (default access), а именно, к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают и доступ к классу целиком — если класс не помечен модификатором public , то все его члены, даже открытые, public , не будут видны из других пакетов.

Пакеты

- В общем случае исходный файл Java может содержать любую (или все) из следующих четырех внутренних частей:
- одиночный package-оператор (не обязательно);
- любое число import-операторов (не обязательно);
- одиночное объявление общего класса (требуется);
- любое число частных классов пакета (не обязательно).

Определение пакета

- Создать пакет очень легко: просто включите оператор package в начало исходного файла Java. Любые классы, объявленные в пределах того файла, будут принадлежать указанному пакету. Оператор package определяет пространство имен, в котором сохраняются классы. Если вы опускаете инструкцию package, имена класса помещаются в пакет по умолчанию (default package), который не имеет никакого имени. (Поэтому-то вы и не должны были волноваться относительно пакетов до настоящего времени.) В то время как пакет по умолчанию хорош для коротких примеров программ, он неадекватен для реальных приложений. В большинстве случаев вы сами будете определять пакет для своего кода.
- **Общая форма ИНСТРУКЦИИ** package

package *pkg;*

- Здесь *pkg* — имя пакета. Например, следующая инструкция создает пакет с именем MyPackage.

```
package MyPackage;
```

- Чтобы хранить пакеты, Java использует каталоги файловой системы. Например, class-файлы для любых классов, которые вы объявляете как часть пакета MyPackage, должны быть сохранены в каталоге с именем MyPackage.
- Помните, что регистр существенен, и имя каталога должно точно соответствовать имени пакета.
- Одну и ту же package-инструкцию могут включать несколько файлов. Она просто указывает, какому пакету принадлежат классы, определенные в файле. Это не исключает принадлежности других классов в других файлах к тому же самому пакету. Большинство реальных пакетов содержат много файлов.

Определение пакета

- Можно создавать *иерархию пакетов.* Для этого необходимо просто отделить каждое имя пакета от стоящего выше при помощи операции "точка". Общая форма инструкции многоуровневого пакета:

package *pkg1 [.pkg2[.pkg3]]* ;

- Иерархия пакетов должна быть отражена в файловой системе вашей системы разработки Java-программ. Например, пакет, объявленный как

package java.awt.image;

- должен быть сохранен в каталоге java/awt/image, java\awt\image или java:awt:image файловой системы UNIX, Windows или Macintosh, соответственно.
- Старайтесь тщательно выбирать имена пакетов.
- Нельзя переименовывать пакет без переименования каталога, в котором хранятся классы.

Использование CLASSPATH

- Размещением корня любой иерархии пакетов в файловой системе компьютера управляет специальная переменная окружения CLASSPATH.
- При сохранении всех классы в одном и том же неименованном пакете, который используется по умолчанию, позволяет просто компилировать исходный код и запускать интерпретатор Java, указывая (в качестве его параметра) имя класса на командной строке.
- Данный механизм работал, потому что заданный по умолчанию текущий рабочий каталог обычно указывается в переменной окружения CLASSPATH, определяемой для исполнительной (run-time) системы Java по умолчанию.
- Однако все становится не так просто, когда включаются пакеты.

Использование CLASSPATH

- Предположим, что вы создаете класс с именем packTest в пакете с именем test. Так как ваша структура каталогов должна соответствовать вашим пакетам, вы создаете каталог с именем test и размещаете исходный файл **PackTest.java** внутри этого каталога.
- Затем вы назначаете test текущим каталогом и компилируете **PackTest.java.** Это приводит к сохранению результата компиляции (файла PackTest.class) в каталоге test, как это и должно быть.
- Когда вы попробуете выполнить этот файл с помощью интерпретатора Java, то он выведет сообщение об ошибке "can't find class PackTest" (не возможно найти класс PackTest).
- Это происходит потому, что класс теперь сохранен в пакете с именем test. Вы больше не можете обратиться к нему просто как к PackTest.

Использование CLASSPATH

- Вы должны обратиться к классу, перечисляя иерархию его пакетов и разделяя пакеты точками. Этот класс должен теперь назваться test. PackTest. Однако если вы попробуете использовать test.PackTest, то будете все еще получать сообщение об ошибке "can't find class test/PackTest" (не возможно найти класс test/PackTest).
- Причина того, что вы все еще принимаете сообщение об ошибках, скрыта в вашей переменной CLASSPATH. Вершину иерархии классов устанавливает CLASSPATH. Проблема в том, что, хотя вы сами находитесь непосредственно в каталоге test, в CLASSPATH ни он, ни, возможно, вершина иерархии классов, не установлены.
- В этот момент у вас имеется две возможности: перейти по каталогам вверх на один уровень и испытать команду Java test.PackTest или добавить вершину иерархии классов в переменную окружения CLASSPATH. Тогда вы будете способны использовать команду Java test.PackTest из любого каталога, и Java найдет правильный class-файл.
- Например, если вы работаете над вашим исходным кодом в каталоге C:\myjava, то установите в CLASSPATH следующие пути:

```
.;C:\myjava;C:\java\classes
```

Защита и управление доступом

- Пакеты добавляют еще одно измерение к управлению доступом. Java обеспечивает достаточно уровней защиты, чтобы допустить хорошо разветвленный контроль над видимостью переменных и методов в пределах классов, подклассов и пакетов.
- Классы и пакеты, с одной стороны, обеспечивают инкапсуляцию, а с другой — поддерживают пространство имен и области видимости переменных и методов.
- Пакеты действуют как контейнеры для классов и других зависимых пакетов. Классы действуют как контейнеры для данных и кода. Класс — самый мелкий модуль абстракции языка Java.
- Из-за взаимодействия между классами и пакетами, Java адресует четыре категории видимости для элементов класса:
- подклассы в том же пакете;
- неподклассы в том же пакете;
- подклассы в различных пакетах;
- классы, которые не находятся в том же пакете и не являются подклассами.

Защита и управление доступом

Доступ к членам классов

	Private	**Без модификатора**	**Protected**	**Public**
Тот же класс	Yes	Yes	Yes	Yes
Подкласс того же пакета	No	Yes	Yes	Yes
Неподкласс того же пакета	No	Yes	Yes	Yes
Другой подкласс пакета	No	No	Yes	Yes
Другой неподкласс пакета	No	No	No	Yes

Импорт пакетов

- В неименованном пакете умолчания нет классов ядра Java, все стандартные классы хранятся в нескольких *именованных* пакетах.
- Чтобы обеспечить видимость некоторых классов или полных пакетов Java, используется оператор import. После импортирования на класс можно ссылаться прямо, используя только его имя.
- В исходном файле Java оператор import следует немедленно после оператора и package (если он используется) и перед любыми определениями класса. Общая форма оператора import:

import *pkgl[.pkg2].(classname | *);

Здесь ***pkgi*** — имя пакета верхнего уровня, ***pkg2*** — имя подчиненного пакета внутри внешнего пакета (имена разделяются точкой). Наконец, вы определяете или явное имя класса, или звездочку (*), которая указывает, что компилятор Java должен импортировать полный пакет.

Следующий кодовый фрагмент показывает использование обеих форм:

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

Интерфейсы

- Создатели языка Java поступили радикально — запретили множественное наследование вообще. При расширении класса после слова extends можно написать только одно имя суперкласса. С помощью уточнения super можно обратиться только к членам непосредственного суперкласса.
- Но что делать, если все-таки при порождении надо использовать несколько предков?
- В таких случаях используется еще одна конструкция языка Java— интерфейс.
- **Интерфейс** (interface), в отличие от класса, содержит только константы и заголовки методов, без их реализации.
- Интерфейсы размещаются в тех же пакетах и подпакетах, что и классы, и компилируются тоже в class-файлы.

Интерфейсы. Определение интерфейса

- С помощью ключевого слова interface Java позволяет полностью отделить интерфейс от его реализации.
- Используя интерфейс, можно определить набор методов, которые могут быть реализованы одним или несколькими классами.
- Сам интерфейс в действительности не определяет никакой реализации.
- Хотя интерфейсы подобны абстрактным классам, они имеют дополнительную возможность: класс может реализовывать больше одного интерфейса.
- В противоположность этому класс может наследовать только один суперкласс (абстрактный или другой).

Интерфейсы

- Интерфейсы разработаны для поддержки динамического вызова методов во время выполнения.
- Обычно для вызова метода одного класса из другого нужно, чтобы оба класса присутствовали во время компиляции и компилятор Java мог проверить совместимость сигнатур методов.
- В иерархической же многоуровневой системе, где функциональные возможности обеспечиваются длинными цепочками связанных в иерархию классов, этот механизм неизбежно используется все большим и большим числом подклассов.
- Интерфейсы исключают определение метода или набора методов из иерархии наследования.
- Так как интерфейсы находятся вне иерархии классов, то для несвязанных в этой иерархии классов появляется иная, более эффективная возможность реализации интерфейсов.

Определение интерфейса

- Общая форма интерфейса выглядит так:

access interface name {
return-type *method-name1(parameter-list) ;*
return-type *method-name2(parameter-list)* ;
type *final-varname1* = value;
type *final-varname2* = value;
return-type *method-nameN (parameter-list) ;*
type *final-vaxnameN* = value;

access — спецификатор доступа (или public или не используется). Если никакой спецификатор доступа не включен, тогда используется доступ по умолчанию, и интерфейс доступен только другим членам пакета, в котором он объявлен.

При объявлении с public интерфейс может использоваться любым другим кодом, паше — имя интерфейса, им может быть любой допустимый идентификатор.

Объявленные методы не имеют тел. Они заканчиваются точкой с запятой после списка параметров.

Каждый класс, который включает интерфейс, должен реализовать все его методы.

Пример определения интерфейса

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

- Здесь объявлен простой интерфейс, содержащий один метод с именем callback(), который имеет единственный целый параметр.

Реализация интерфейсов

- Общая форма класса, который включает implements предложение, выглядит примерно так:

access class classname [extends superclass]

[implements _interface_ [,_interface_...]] { // тело-класса }

Здесь *access* — спецификатор доступа (public или не используется). Если класс реализует более одного интерфейса, они разделяются запятой. Если класс реализует два интерфейса, которые объявляют один и тот же метод, то клиенты любого интерфейса будут использовать один и тот же метод.

Методы, которые реализуют интерфейс, должны быть объявлены как public.

Пример класса, который реализует интерфейс

- class Client implements Callback {

```
// Реализация Callback-интерфейса
public void callback(int p) {
System.out.println
("callback вызван с аргументом “+p);
} }
```

- Обратите внимание, что callback() объявлен со спецификатором доступа public.

Реализации доступа через интерфейсные ссылки

- Можно объявлять переменные как объектные ссылки, которые используют интерфейсный тип, а не тип класса.
- В такой переменной можно сохранять всякий экземпляр любого класса, который реализует объявленный интерфейс.
- Когда вы вызываете метод через ссылку такого рода, будет вызываться его правильная версия, основанная на актуальном экземпляре интерфейса. Выполняемый метод отыскивается динамически (во время выполнения), что позволяет создавать классы позже кода, который вызывает их методы.
- Кодом вызова можно управлять через интерфейс, ничего не зная об объекте вызова.

Пример вызова метода

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

Вывод этой программы:

```
callback вызван с аргументом 42
```

- Обратите внимание, что переменной c, объявленной с типом интерфейса Callback, был назначен экземпляр класса client.
- Хотя разрешается использовать с для обращения к методу callback(), она не может обращаться к любым другим членам класса client. Переменная интерфейсной ссылки обладает знаниями только методов, объявленных в соответствующей интерфейсной декларации.

Частичные реализации

- Если класс включает интерфейс, но полностью не реализует методы, определенные этим интерфейсом, то этот класс должен быть объявлен как abstract (абстрактный).

```
abstract class Incomplete implements Callback {
int a, b;
void show() {
System.out.println(a + " " + b) ;
}
// ...
```

- Здесь класс incomplete не реализует callback() и должен быть объявлен как абстрактный.
- Любой класс, который наследует incomplete, должен реализовать callback() или объявить себя как abstract.

Применения интерфейсов

- Стек может иметь фиксированный размер, или быть "растущим".
- Стек может также содержаться в массиве, связном списке, двоичном дереве и т. д.
- Независимо от того, как стек реализован, интерфейс стека остается тем же самым.
- Методы push() и pop() определяют интерфейс к стеку независимо от подробностей реализации.
- Ниже показан интерфейс, определяющий целый стек.

```
// Определение интерфейса целого стека,
interface IntStack {
void push(int item);        // запомнить элемент
int pop();    // извлечь элемент
}
```

Переменные в интерфейсах

- Можно использовать интерфейсы для импорта разделяемых констант во множественные классы просто объявлением интерфейса, который содержит переменные, инициализированные желательными значениями. Когда вы включаете этот интерфейс в класс (т. е. "реализуете" интерфейс), все имена указанных переменных окажутся в области их видимости как константы.
- Если интерфейс не содержит методов, то любой класс, который включает такой интерфейс, фактически не реализует ничего.

```
import java.util.Random;
interface SharedConstants {
int N0=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.nextDoubleO ) ;
 if (prob < 30)
return NO;  // 60%
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("Нет"); break;
case YES:
System.out.println("Да"); break;
case MAYBE:
System.out.println("Возможно"); break;
case LATER:
System.out.println("Позже"); break;
case SOON:
System.out.println("Вскоре"); break;
case NEVER:
System.out.println("Никогда"); break; } }

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

Расширение интерфейсов

- Один интерфейс может наследовать другой при помощи ключевого слова extends.
- Синтаксис — тот же самый, что для наследования классов.
- Когда класс реализует интерфейс, который наследует другой интерфейс, первый должен обеспечить реализацию для всех методов, определенных в цепочке наследования интерфейса.

Расширение интерфейсов (пример)

```
// Один интерфейс расширяет другой,
interface A {
void meth1();
void meth2 (); }
// В теперь включает meth1() и meth2() , а сам он добавляет meth3().
interface B extends A {
void metn3(); }
// Этот класс должен реализовать все из А и В.
class MyClass implements B {
public void meth1() {
System.out.println("Реализует meth1().");
}
public void meth2() {
System.out.println("Реализует meth2()."); }
public void meth3() {
System.out.println("Реализует meth3()."); } }

class IFExtend {
public static void main (String arg[].) { MyClass ob = new MyClass ();
ob.meth1();
ob.meth2();
ob.meth3(); } }
```

Спасибо за внимание!