1)Основное отличие абстрактного класса от интерфейса заключается во множественности,т.е. множественностью интерфейсов они компенсировали одиночность наследования класса.

282

2)У всех массивов имеется открытый,а не защищенный метод clone().Им можно пользоваться для создания нового массива,содержащего копии всех элементов,как в следующем примере кода:

int[] luckyNumbers = {2,3,5,7,1,13};

int[] cloned = (int[]) luckyNumbers.clone();

cloned[5] = 12;//не произойдет изменения массива luckyNumbers[5];

3)**Внутренним классом называется класс,определенный в другом классе**.**А зачем он вообще нужен?**На то имеются следующие причины:

* Объект внутреннего класса **имеет доступ к данным объекта,в котором он определен,включая закрытые данные.**
* **Внутренний класс** можно скрыть от других классов того же пакета
* **Анонимный внутренний класс** оказывается удобным в тех случаях,когда требуется определить обратный вызов в процессе выполнения программы,не прибегая к необходимости писать много кода.

4)Внутри внутренних классов не может быть static методов, а во внешнем классе не может быть static-методов,которые обращаются к внутреннему классу.

5) Car car = new Car();

Car.Door door = car.new Door();

6) public int getHeight()

{

if (height!=0)

return this.height;

else

return (int)(Car.this.height \* 0.8);

}

**Две ссылки this**

**5)ПУБЛИЧНЫЙ ВНУТРЕННИЙ КЛАСС-ТОЖЕ КЛАСС!!!**

**6)Очень важная особенность внутренних вложенных классов:**

public static void main(String[] args) {  
 */\*\*  
 \* Мы можем спокойно создавать  
 \* объекты класса Mouse, даже когда нет ни одного созданного объекта класса Zoo.  
 \* Вот как можно это сделать:  
 \*/  
 obj* = new Difficult\_Constructor.TEST();  
 System.*out*.println(*obj*);  
}

**Самое сложное здесь в том,что у вложенного статического класса не было конструктора,но при помощи этой конструкции можно создать объект без конструктора.**

А теперь здоровенный кусок кода,который покажет тебе,как это работает:

package Inner\_Classes;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.util.Date;  
  
public class TalkingClock {  
 private int interval;  
 private boolean beep;  
  
 */\*\*  
 \*  
 \** ***@param*** *interval интервал между сообщениями  
 \** ***@param*** *beep Истинно,если часы должны издавать звуковой сигнал  
 \*/* public TalkingClock(int interval, boolean beep) {  
 this.interval = interval;  
 this.beep = beep;  
 }  
  
 */\*\*  
 \* Смотри,что будет,если мы пробьем через рефлексию наш изначальный класс:  
 \* public class Inner\_Classes.TalkingClock  
 {  
 public Inner\_Classes.TalkingClock (int, boolean);  
  
 static boolean access$000(Inner\_Classes.TalkingClock);А что же это?Это наш внутренний класс проверяет значение поля beep  
 Т.е. компилятор на самом деле не особо различает внутренний класс и внешний.А зачем внутренние классы?  
 С ними сложнее напортачить.Т.е. автор книги говорит,что по сути,какой-нибудь хитроумный хакер найдет способ взломать  
 твой код и получить доступ к твоим данным,но вот обычному криворукому пользователю сделать это будет невозможно  
 Соответственно,использование внутренних классов предовтращает различного рода ошибки  
 public void start();  
  
 private int interval;  
 private boolean beep;  
 }  
 \*/* public void start()  
 {  
 ActionListener listener = new TimePrinter();*/\*\*  
 Аналогичным будет указание this.new TimePrinter(),хотя ссылка this тут лишняя,  
 такое объявление более общепринято  
 \*/* Timer t = new Timer(interval,listener);  
 t.start();  
 }  
 public class TimePrinter implements ActionListener {  
 */\*\*  
 \* Вот что будет,если вкинуть внутренний класс в рефлексию  
 \* public class Inner\_Classes.TalkingClock$TimePrinter  
 {  
 public Inner\_Classes.TalkingClock$TimePrinter (Inner\_Classes.TalkingClock);  
  
 public void actionPerformed(java.awt.event.ActionEvent);  
  
 final Inner\_Classes.TalkingClock this$0;//ссылка на внешинй класс,которая синтезируется компилятором  
 }  
 \*  
 \*/* @Override  
 public void actionPerformed(ActionEvent e) {  
 System.*out*.println("The time is: " + new Date());  
 if(beep) Toolkit.*getDefaultToolkit*().beep();  
 */\*\*  
 \* Смотри-ка,а вроде и приватное,а доступ все равно имеется.  
 \*/* }  
 }  
 }

4)Статические внутренние классы применяются тогда,когда доступ к объекту внутреннего класса не требуется

“— Пусть слово **static** тут не вводит тебя в заблуждение. Если переменная объявлена статической – то она существуют в единственно экземпляре – это верно. Но статический вложенный класс больше похож на статический метод в этом плане. Слово static перед объявлением класса указывает, что этот класс не хранит в себе ссылок на объекты внешнего класса, внутри которого объявлен.

— Ага. Обычные методы втихаря хранят ссылку на объект, а статические – нет. То же и со статическими классами, я прав, Элли?”

1) При создании объектов вложенного класса (как класс Mouse) вне внешнего класса-родителя, надо еще указывать через точку и имя внешнего класса.

Например так: Zoo.Mouse.

2) Класс Zoo.Mouse и его объекты имеют доступ к private static переменным и методам класса Zoo ( класс Mouse ведь тоже объявлен внутри класса Zoo).

На этом на сегодня все

Наследование внутренних классов

С наследованием вложенных классов нет никаких проблем,они реализуются так же,как и обычные

public class Car

{

public static class Door

{

}

}

public class LamborginiDoor extends Car.Door

{

}

А обычный внутренний класс содержит ссылку на свой внешний класс,поэтому наследовать надо следующим образом:

public class Car

{

public class Door

{

}

}

public class LamborginiDoor extends Car.Door

{

LamborginiDoor(Car car)

{

car.super();

}

}

!!!!!!

В конструктор класса Door нужно неявно передать объект класса Car, это делается с помощью специальной конструкции: «car.super()».

!!!!!!!!

Кстати, если ты попробуешь создать конструктор вообще без параметров, то программа просто не скомпилируется. Немного странно и непривычно, да?

ВАЖНЫЙ КУСОК КОДА:

public class Solution {  
 public class Building {  
 public class Hall {  
 private BigDecimal square;  
  
 public Hall(BigDecimal square) {  
 this.square = square;  
 }  
 }  
  
 public class Apartments {  
 }  
  
 }  
 public class Apt3Bedroom extends Solution.Building.Apartments{  
 public Apt3Bedroom(Building building){  
 building.super();  
 }  
  
 }  
 public class BigHall extends Solution.Building.Hall{  
 public BigHall(Building building,BigDecimal square){  
 building.super(square);  
 }  
 }  
  
 public static void main(String[] args) {  
  
 }  
}

LOCAL CLASSES

class Car

{

public ArrayListcreatePoliceCars(int count)

{

ArrayList result = new ArrayList();

class PoliceCar extends Car

{

int policeNumber;

PoliceCar(int policeNumber)

{

this.policeNumber = policeNumber;

}

}

for(int i=0; i<count; i++)

result.add(new PoliceCar(i));

return result;

}

}

Основное свойство локальных классов-они могу использовать локальные переменные,но только неизменяемые(помеченные как final).

А почему final?

Понял в чем штука? Внутренний класс не может изменить локальную переменную метода, т.к. к тому времени, когда будет исполняться код этого класса, мы уже можем вообще уйти из метода.

Теперь второй момент. Метод toString() использует переданную переменную. Для этого пришлось:

**А)** сохранить ее внутри сгенерированного класса

**Б)** добавить ее в конструктор.

— Понял. Классы, объявленные внутри метода, всегда работают с копией переменных.

— Именно!

— Тогда понятно, почему переменные должны быть final. И почему их нельзя менять. Если на самом деле ты работаешь с копией, а не с оригиналом, то пользователь не поймет, почему не может менять значение переменной, а значит, надо просто ему запретить ее менять.

Просто анонимный класс компилится в отдельный класс,и обрабатывается после.

У внутренних классов есть свой специальный инициализатор(конструкторы то добавлять нельзя)

class Car

{

public ArrayListcreatePoliceCars(int count)

{

ArrayList result = new ArrayList();

for(int i=0; i<count; i++)

{

final int number = i;

result.add(new Car()

{

int policeNumber;

{

policeNumber = number;

}

});

}

return result;

}

}