ВЪТРЕШНИ КЛАСОВЕ

ЛЕКЦИОНЕН КУРС "ООП (JAVA)"

Distributed @learning center

СТРУКТУРА НА ЛЕКЦИЯТА

- Определение
- Отношение външен вътрешен клас
- Използване
- Примери
- Контролни рамки

ОПРЕДЕЛЕНИЕ ЗА ВЪТРЕШНИ КЛАСОВЕ

- Вътрешен клас
  - Възможно е да се постави дефиниция на клас в друг клас
- Добра възможност за:
  - Групиране на класове, които принадлежат логически един към друг
  - Контролиране видимостта на единия от класовете в другия

ОПРЕДЕЛЕНИЕ ЗА ВЪТРЕШНИ КЛАСОВЕ

- На пръв поглед вътрешните класове изглеждат като прост механизъм за скриване на кода
  - Поставяме ги вътре в други класове
  - Вътрешните класове правят повече от това
    - Те знаят за „заобикалящите“ ги класове
    - Могат да комуникират с тях
- Вътрешните класове са различни от композицията
- Необходимостта от вътрешни класове не винаги е очевидна

PARCEL1: ТИПИЧНА УПОТРЕБА

```
public class Parcel1 {
    class Contents {
       private int i=11;
       public int value() { return i; }
    }
    class Destination {
       private String label;
       Destination(String whereTo) {
          label = whereTo;
       }
       String readLabel() { return label; }
    }
    public void ship(String dest) {
       Contents c = new Contents();
       Destination d = new Destination(dest);
       System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
       Parcel1 p = new Parcel1();
       p.ship("България");
    }
}
```

PARCEL1: ТИПИЧНА УПОТРЕБА

```
public class Parcel1 {
    class Contents {
       private int i=11;
       public int value() { return i; }
    }
    class Destination {
       private String label;
       Destination(String whereTo) {
          label = whereTo;
       }
       String readLabel() { return label; }
    }
    public void ship(String dest) {
       Contents c = new Contents();
       Destination d = new Destination(dest);
       System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
       Parcel1 p = new Parcel1();
       p.ship("България");
    }
}
```

- Обикновено във въвеждащите курсове по програмиране на Java се изучава, че класовете могат да бъдат само public и „приятелски“
- Вътрешните класове могат да бъдат също private и protected
- Създаване на вътрешен клас: поставяме дефиницията на класа вътре в „обграждащия“ клас
- Използват методи както всеки друг клас

- Методът ship() точно както използването на другите методи
- Разлика – имената са вложени в Parcel1

PARCEL1: ТИПИЧНА УПОТРЕБА

```
public class Parcel1 {
    class Contents {
       private int i=11;
       public int value() { return i; }
    }
    class Destination {
       private String label;
       Destination(String whereTo) {
          label = whereTo;
       }
       String readLabel() { return label; }
    }
    public void ship(String dest) {
       Contents c = new Contents();
       Destination d = new Destination(dest);
       System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
       Parcel1 p = new Parcel1();
       p.ship("България");
    }
}
```

? Резултат

България

PARCEL2

```
public class Parcel2 {
        class Contents {
            private int i=11;
            public int value() { return i; }
        }
        class Destination {
            private String label;
            Destination(String whereTo) {
                label = whereTo;
            }
            String readLabel() { return label; }
        }
        public Destination to(String s) {
            return new Destination(s);
        }
        public Contents cont() {
            return new Contents();
        }
        public void ship(String dest) {
            Contents c = new Contents();
            Destination d = new Destination(dest);
            System.out.println(d.readLabel());
        }
```

```
public static void main(String[] args) {
        Parcel2 p = new Parcel2();
        p.ship("България");
        Parcel2 q = new Parcel2();
        Parcel2.Contents c = q.cont();
        Parcel2.Destination d = q.to("Малта");
    }
}
```

PARCEL2

```
public class Parcel2 {
    class Contents {
        private int i=11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents cont() {
        return new Contents();
    }
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
```

Често, един външен клас има методи, които връщат референция към вътрешен клас

```
public static void main(String[] args) {
        Parcel2 p = new Parcel2();
        p.ship("България");
        Parcel2 q = new Parcel2();
        Parcel2.Contents c = q.cont();
        Parcel2.Destination d = q.to("Малта");
    }
}
```

PARCEL2

```
public class Parcel2 {
    class Contents {
        private int i=11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents cont() {
        return new Contents();
    }
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
```

Ако искаме да създаваме обект от вътрешния клас където и да е (освен тялото на не-static метод на външния клас), трябва да се укаже типа на този обект като:

Име-външен-клас.Име-вътрешен-клас

```
public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("България");
    Parcel2 q = new Parcel2();
    Parcel2.Contents c = q.cont();
    Parcel2.Destination d = q.to("Малта");
  }
}
```

ВРЪЗКА С ВЪНШЕН КЛАС

- Досега изглежда, че вътрешните класове са само схема на скриване на имена и код
  - Която е полезна, но не е напълно убедителна
- Има обаче още една особеност
  - Когато създаваме вътрешен клас, обект от този вътрешен клас има връзка към обхващащия обект
  - Така без специална квалификация има достъп до членовете на този прилежащ обект
- Т.е. вътрешните класове имат права на достъп до всички елементи в прилежащия клас

ПРИМЕР

```
interface Selector {
   boolean end();
   Object current();
   void next();
}
public class Sequence {
   private Object[] items;
   private int next = 0;
   public Sequence(int size) { items = new Object[size]; }
   public void add(Object x) {
      if(next < items.length)
         items[next++] = x;
}
private class SequenceSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }
}
```

Sequence е просто масив от Object с фиксиран размер с клас около него

```
public Selector selector() {
     return new SequenceSelector();
}
   public static void main(String[] args) {
      Sequence sequence = new Sequence(10);
       for(int i = 0; i < 10; i++)
          sequence.add(Integer.toString(i));
       Selector selector = sequence.selector();
       while(!selector.end()) {
           System.out.print(selector.current() + " ");
           selector.next();
       }
   }
}
```

ПРИМЕР

```
interface Selector {
   boolean end();
   Object current();
   void next();
}
public class Sequence {
   private Object[] items;
   private int next = 0;
   public Sequence(int size) { items = new Object[size]; }
   public void add(Object x) {
      if(next < items.length)
         items[next++] = x;
}
private class SequenceSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }
}
```

Извикваме add() за добавяне нов обект в края на последователността (ако е останало място)

```
public Selector selector() {
     return new SequenceSelector();
}
   public static void main(String[] args) {
      Sequence sequence = new Sequence(10);
       for(int i = 0; i < 10; i++)
          sequence.add(Integer.toString(i));
       Selector selector = sequence.selector();
       while(!selector.end()) {
           System.out.print(selector.current() + " ");
           selector.next();
       }
   }
}
```

ПРИМЕР

```
interface Selector {
  boolean end();
  Object current();
  void next();
}
public class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size) { items = new Object[size];
  public void add(Object x) {
    if(next < items.length)
      items[next++] = x;
}
private class SequenceSelector implements Selector {
   private int i = 0;
   public boolean end() { return i == items.length; }
   public Object current() { return items[i]; }
   public void next() { if(i < items.length) i++; }
}
```

- Интерфейсът се използва за извличане обекти от Sequence - пример за Iterator
- Селектор позволява да видим дали сме в края, да получим достъп до текущия обект и да преминем към следващия обект в последователността
- Понеже Selector е интерфейс, други класове могат да го имплементират (по свой начин) и други методи могат да го възприемат като аргумент (за създаване на по-общ код)

```
  public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
      sequence.add(Integer.toString(i));
    Selector selector = sequence.selector();
    while(!selector.end()) {
      System.out.print(selector.current() + " ");
      selector.next();
    }
  }
}
```

ПРИМЕР

- SequenceSelector е private клас, осигуряващ функционалност на Selector
- SequenceSelector е вътрешен клас – при внимателен анализ ще забележим, че:
  - Методите end(), current() и next() реферират item, който не е част от SequenceSelector, а е частно поле в външния клас
  - Вътрешният клас обаче има достъп до методи и полета от обкръжаващия клас, сякаш ги притежава

```
interface Selector {
  boolean end();
  Object current();
  void next();
}
public class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size)
  public void add(Object x) {
    if(next < items.length)
      items[next++] = x;
  }
  private class SequenceSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }
  }
```

```
}
  public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
      sequence.add(Integer.toString(i));
    Selector selector = sequence.selector();
    while(!selector.end()) {
      System.out.print(selector.current() + " ");
      selector.next();
    }
  }
}
```

ПРИМЕР

```
interface Selector {
  boolean end();
  Object current();
  void next();
}
public class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size) { items = new Object[size]; }
  public void add(Object x) {
     if(next < items.length)
        items[next++] = x;
}
private class SequenceSelector implements Selector {
   private int i = 0;
   public boolean end() { return i == items.length; }
   public Object current() { return items[i]; }
   public void next() { if(i < items.length) i++; }
}
```

? Резултат

```
public Selector selector() {
    return new SequenceSelector();
}
  public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
     for(int i = 0; i < 10; i++)
        sequence.add(Integer.toString(i));
     Selector selector = sequence.selector();
     while(!selector.end()) {
        System.out.print(selector.current() + " ");
        selector.next();
     }
  }
}
```

0 1 2 3 4 5 6 7 8 9

ОБОБЩЕНИЕ

- Един вътрешен клас има автоматичен достъп до членовете на прилежащия външен клас
- Как може да се случи това?
  - Вътрешният клас "тайно" поддържа референция към конкретния обект на прилежащия външен клас, отговорен за създаването му
  - Когато реферираме член на външния клас, тази референция се използва за избора му

ОБОБЩЕНИЕ

- За щастие, компилаторът се грижи за всичко това, но се вижда, че обект от вътрешен клас може да бъде създаден само във връзка с обект от прилежащия външен клас
  - Изключение: статичен вътрешен клас
- Конструкцията на обект от вътрешен клас изисква референция към обекта от прилежащия клас
  - Компилаторът ще се "оплаче", ако не може да получи достъп до тази референция
- По-голямата част от времето се случва без намеса от страна на програмиста

.THIS

- Ако трябва да създадем референция към обекта от външния клас, използваме името на външния клас, последван от точка и this
- Получената референция е автоматично коректният тип, който е известен и проверен при компилирането
  - Пести оверхед по време на изпълнение

ПРИМЕР ЗА .THIS

```
public class DotThis {
   void f() {
      System.out.println("DotThis.f()");
   }

   public class Inner {
      public DotThis outer() {
            return DotThis.this;
            //A plain "this" would be Inner's "this"
       }
}

      public Inner inner() {
            return new Inner();
       }

       public static void main(String[] args) {
            DotThis dt = new DotThis();
            DotThis.Inner dti = dt.inner();
            dti.outer().f();
        }
}
```

DotThis.f()

.NEW

- Понякога искаме да накараме друг обект да създаде обект на един от вътрешните му класове
- За да направим това, трябва да предоставим референция към другия обект от външния клас в new-израз
  - Използвайки .new синтаксиса

ПРИМЕР ЗА .NEW

```
public class DotNew {
    public class Inner {  }

   public static void main(String[] args) {
      DotNew dn = new DotNew();
      DotNew.Inner dni = dn.new Inner();
   }
}
```

ПРЕОБРАЗУВАНЕ НАГОРЕ

- При необходимост от скриване Java предоставя перфектен механизъм
  - Позволяваме класът да е „приятелски“ – видим само в един пакет
  - Вместо да го създаваме като вътрешен клас
- Силата на вътрешните класове
  - Когато правим преобразуване нагоре
  - И особено към интерфейс

ПРИМЕР

```
public interface Destination {
      String readLabel();
}

public interface Contents {
      int value();
}
```

Интерфейсите автоматично правят своите членове public

```
class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}
```

```
public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("България");
        //Illegal -- can't access private class:
        //! Parcel4.PContents pc = p.new PContents();
    }
}
```

ПРИМЕР

```
class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}
```

Вътрешният клас е private – само Parcel4 има достъп до него
Нормалните класове са публични или приятелски

```
public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("България");
        //Illegal -- can't access private class:
        //! Parcel4.PContents pc = p.new PContents();
    }
}
```

ПРИМЕР

```
class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}
```

Вътрешният клас е protected – само Parcel4, класовете в пакета на Parcel4 и наследниците на Parcel4 имат достъп до него

```
public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("България");
        //Illegal -- can't access private class:
        //! Parcel4.PContents pc = p.new PContents();
    }
}
```

ПРИМЕР

```
class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Desti
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}
```

- Клиент-програмата има ограничено познание и достъп до вътрешните класове
- Не може да се прави преобразуване надолу до вътрешен private клас – нямаме достъп до името
- private вътрешните класове доставят начин на проектантите, напълно да премахнат всякакви зависимости от типа на кода и изцяло да скрият подробностите на имплементацията

```
public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("България");
        //Illegal -- can't access private class:
        //! Parcel4.PContents pc = p.new PContents();
    }
}
```

ПРИМЕР

```
class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}
```

? Резултат

България

```
public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("България");
        //Illegal -- can't access private class:
        //! Parcel4.PContents pc = p.new PContents();
    }
}
```

ИЗПОЛЗВАНЕ НА ВЪТРЕШНИ КЛАСОВЕ

- „Обикновени“ вътрешни класове – лесни за разбиране
- Съществуват други, по-неясни начини за използване на вътрешните класове - могат да се създават в рамките на:
  - Методи или
  - Произволни области на видимост
- Причини:
  - Имплементиране на интерфейси така, че да е възможно създаване и връщане на референции
  - Решаване на сложен процес – създаваме помощни класове, които подпомагат решението, но не искаме класът да е достъпен

ВЪЗМОЖНОСТИ

- Клас, дефиниран в метод
- Клас, дефиниран в област за видимост вътре в метод
- Анонимен клас, имплементиращ интерфейс
- Анонимен клас, разширяващ клас, който има конструктор, различен от подразбиране
- Анонимен клас, инициализиращ полета
- Анонимен клас, извършващ конструиране чрез инициализация на екземпляр

ДЕФИНИРАН В МЕТОД КЛАС

```
public class Parcel4 {
    public Destination dest(String s) {
        class PDestination  implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel() { return label; }
        }
      return new PDestination(s);
    }
    public static void main(String[] args) {
      Parcel4 p = new Parcel4();
      Destination d = p.dest("България");
  }
}
```

- Класът PDestination по-скоро част от метода dest()
- Недостъпен извън този метод
- Преобразуване нагоре във върнатия резултат
  - dest() не връща нищо друго освен референция към базовия клас Destination
- PDestination е валиден обект

КЛАС, ДЕФИНИРАН В ОБЛАСТ ВЪТРЕ В МЕТОД

```
public class Parcel5 {
      private void internalTracking(boolean b) {
          if(b) {
               class TrackingSlip {
                    private String id;
                    TrackingSlip(String s) {
                         id = s;
                    }
                    String getSlip() { return id; }
               }
               TrackingSlip ts = new TrackingSlip("slip");
               String s = ts.getSlip();
          }
          // Не може да се използва тук! Извън областта
          // на видимост е;
          //! TrackingSlip ts = new TrackingSlip("x");
      }
      public void track() { internalTracking(true); }
      public static void main(String[] args) {
      Parcel5 p = new Parcel5();
      p.track();
    }
}
```

- Класът TrackingSlip е вложен в областта на видимост на if конструкция
- Това не означава, че класът е условно създаден
- Компилира се заедно с всичко друго
- Той обаче не е достъпен извън областта на видимост, в която е дефиниран
- С изключение на това, класът изглежда като всеки друг обикновен клас

АНОНИМЕН КЛАС: ИМПЛЕМЕНТИРА ИНТЕРФЕЙС

```
public interface Destination {
      String readLabel();
}

public interface Contents {
      int value();
}
```

```
class MyContents implements Contents {
        private int i=11;
        public int value() { return i; }
}
return new MyContents();
```

```
public class Parcel6 {
       public Contents cont() {
            return new Contents() {
                 private int i=11;
                 public int value() { return i; }
            }; // В този случай се изисква ";"
       }
       public static void main(String[] args) {
            Parcel6 p = new Parcel6();
            Contents c = p.cont();
       }
}
```

- Методът cont() комбинира създаване на връщаната стойност с дефиниция на класа, който представя тази стойност
- Класът е анонимен (без име)
- Този синтаксис означава „създай обект на анонимен клас, наследен от Contents“
- Референцията, върната от new автоматично (чрез upcast) се предава на обръщение към Contents

АНОНИМНИ КЛАСОВЕ: КОНСТРУКТОР С АРГУМЕНТ

```
public class Wrapping {
      private int i;
      public Wrapping(int x) { i = x; }
      public int value() { return i; }
}
```

Базовият клас изисква конструктор с аргумент

```
public class Parcel7 {
        public Wrapping wrap(int x) {
            // Извикване на базовия конструктор:
            return new Wrapping(x) {
                public int value() {
                    return super.value() * 47;
                }
            };
        }
        public static void main(String[] args) {
            Parcel7 p = new Parcel7();
            Wrapping w = p.wrap(10);
        }
}
```

- Предаваме съответния аргумент на конструктора на базовия клас x в new Wrapping(x)
- Анонимните класове не могат да имат конструктор - там ще извикаме super()

АНОНИМНИ КЛАСОВЕ: ИНИЦИАЛИЗАЦИЯ НА ОБЕКТ

```
public class Parcel8 {
      // Аргументът трябва да е final, за да се използва
      // вътре в анонимния вътрешен клас:
    public Destination dest( final String dest ) {
          return new Destination() {
              private String label = dest;
              public String readLabel() { return label; }
          };
     }
     public static void main(String[] args) {
         Parcel8 p = new Parcel8();
         Destination d = p.dest("България");
     }
}
```

Какво става, ако трябва да се извърши някаква инициализация на обект от анонимен вътрешен клас?

- Не можем да използваме конструктор – той липсва
- Инициализацията се извършва в мястото на дефиниране на полетата
- Ако за това използваме обект, дефиниран извън анонимния вътрешен клас, компилаторът изисква външния обект да е final

АНОНИМНИ КЛАСОВЕ: ДЕЙНОСТ С КОНСТРУКТОРИ

? Резултат

```
public class Parcel9 {
      public Destination dest(final String dest, final float price) {
          return new Destination() {
              private int cost;
               // Инициализиране на екземпляр за всеки обект:
              {
                 cost = Math.round( price );
                 if( cost > 100 )
                     System.out.println("Over budget!");
              }
              private String label = dest;
              public String readLabel() { return label; }
          };
      }
      public static void main(String[] args) {
      Parcel9 p = new Parcel9();
      Destination d = p.dest("България", 101.395f);
   }
}
```

Over budget!

Какво става, ако има необходимост от дейност с конструктори?

- С помощта на instance initialization може ефективно да създаваме конструктор за анонимен вътрешен клас
- Вътре в инициализатора на екземпляра има код, който не би могъл да се изпълни като част от инициализация на полето (if конструкция)
- Така задействан, един инициализатор на екземпляр е конструкторът на анонимен вътрешен клас

СТАТИЧНИ ВЪТРЕШНИ КЛАСОВЕ

- Ако няма необходимост от връзка между обект от вътрешен клас и обект от външен клас, тогава вътрешният клас може да бъде static
  - Обектите от обикновените вътрешни класове неявно поддържат референции към обекти от заобикалящите класове
  - Това не е вярно за static вътрешните класове
- Static вътрешен клас означава:
  - Няма нужда от обект от външен клас, за да създадем обект на static вътрешен клас
  - Не можем да осъществим достъп до обект на външен клас от обект от static вътрешен клас
- Не-static вътрешните класове не могат да имат static данни, static методи и static вътрешни класове

ПРИМЕР: PARCEL10

```
public class Parcel10 {
      private static class PContents implements Contents {
         private int i = 11;
         public int value() { return i; }
      }
      protected static class PDestination  implements Destination {
         private String label;
         private PDestination(String whereTo) {
            label = whereTo;
         }
         public String readLabel() { return label; }
         // Статичните вътрешни класове могат да съдържат други статични елементи:
         public static void f() { }
         static int x = 10;
         static class AnotherLevel {
            public static void f() { }
            static int x = 10;
         }
   }
   public static Destination dest(String s) { return new PDestination(s);  }
   public static Contents cont() {  return new PContents();    }
   public static void main(String[] args) {
        Contents c = cont();
        Destination d = dest("България");
   }
}
```

- В main() не е необходим обект на Parcel10
- Вместо него използваме нормалния синтаксис за извикване на static методи

ПРИМЕР: PARCEL11

```
public class Parcel11 {
        class Contents {
                private int i=11;
                public int value() { return i; }
        }
        class Destination {
                private String label;
                Destination(String whereTo) {
                   label = whereTo;
                }
                String readLabel() { return label; }
        }

        public static void main(String[] args) {
                Parcel11 p = new Parcel11();
                // За създаване на инстанция на вътрешен клас
                // трябва да се използва инстанция на външен клас:
                Parcel11.Contents c = p.new Contents();
                Parcel11.Destination d = p.new Destination("България");
        }
}
```

? Коментар

- Понякога един обект изисква от някой друг обект да създаде обект от своите вътрешни класове
- Трябва да се осигури обръщение към другия обект на външния клас в израза new

ВЛОЖЕНИ КЛАСОВЕ

```
class MNA {
      private void f() {}
      class A {
          private void g() {}
          public class B {
              void h() {
                  g();
                  f();
              }
          }
      }
}

public class MultiNestingAccess {
       public static void main(String[] args) {
            MNA mna = new MNA();
            MNA.A mnaa = mna.new A();
            MNA.A.B mnaab = mnaa.new B();
            mnaab.h();
       }
}
```

? Коментар на класа

- Няма значение колко дълбоко е вложен един вътрешен клас
- Може да има прозрачен достъп до всички членове на всички класове, в които е вложен

ЗАЩО ВЪТРЕШНИ КЛАСОВЕ?

- Защо Sun си създадоха толкова проблеми за да добавят тази фундаментална особеност на езика Java?
  - Обикновено вътрешните класове наследяват класове или имплементират интерфейси
  - Кодът във вътрешните класове манипулират обектите на външните класове, в които са създадени
  - Един вътрешен клас осигурява нещо като прозорец във външния клас

МОТИВАЦИЯ ЗА ВЪТРЕШНИ КЛАСОВЕ

- Какво различава вътрешен клас, прилагащ даден интерфейс, от външен клас, прилагащ същия интерфейс?
  - Не можем винаги да използваме удобството на интерфейсите – понякога се налага да работим с имплементации
- Най-убедителната причина за вътрешните класове
  - Всеки вътрешен клас може независимо да наследява от имплементация
  - Така, вътрешният клас не е ограничен от това, дали външният клас наследява от такава имплементация

МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ

- Един начин за интерпретиране на вътрешните класове
  - Завършване на проблема на множественото наследяване
  - Интерфейсите разрешават само част от проблемите
  - Вътрешните класове позволяват ефективно „наследяване на множествено приложение“
  - Позволяват ефективно наследяване на повече от един не-интерфейс

МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ НА ИНТЕРФЕЙСИ

```
interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
    B makeB() {
        // Анонимен вътрешен клас:
        return new B() {};
    }
}

public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        takesA(x);
        takesA(y);
        takesB(x);
        takesB(y.makeB());
    }
}
```

? Коментар на класа

- Имаме два интерфейса, които трябва да се добавят по някакъв начин в един клас
- Поради гъвкавостта на интерфейсите съществуват две възможности:
  - Самостоятелен клас
  - Вътрешен клас

МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ НА ИМПЛЕМЕНТАЦИИ

```
class C {}
abstract class D {}

class Z extends C {
      D makeD() { return new D() {}; }
}

public class MultiImplementation {
      static void takesC(C c) {}
      static void takesD(D d) {}
      public static void main(String[] args) {
          Z z = new Z();
          takesC(z);
          takesD(z.makeD());
      }
}
```

? Коментар на класа

- Ако, вместо интерфейси, имаме абстрактни или конкретни класове
- Вече сме ограничени
- Трябва да използваме вътрешни класове, ако един клас трябва да допълва другите два

ОБОБЩЕНИЕ

- Ако няма необходимост да решаваме проблема „наследяване на множествени приложения“, бихме могли да не използваме вътрешни класове
- Допълнителни възможности с вътрешни класове:
  - Вътрешният клас може да има множество инстанции, всяка една със своя информация за състоянието и която е независима от информацията в обекта от външния клас
  - В един самостоятелен външен клас може да има няколко вътрешни класове, всеки от които допълва същия интерфейс или наследява от същия клас по различен начин
  - Мястото на създаване на обекта на вътрешния клас не е свързано със създаването на обекта на външния клас
  - Няма потенциално объркваща ‚е‘-връзка с външния клас – това е отделен елемент

ЗАТВАРЯНЕ

- Едно затваряне (closure) е обект, който запазва информация за мястото, където е създаден
- В съответствие с тази дефиниция един вътрешен клас е обектно-ориентирано затваряне
  - Не съдържа информацията за външния клас (мястото, където е създаден)
  - Но, автоматично съдържа референция обратно към целия обект от външния клас
  - Посредством нея може да манипулира всички членове на външния клас, дори pravite

ОБРАТНА ВРЪЗКА

- Един от най-убедителните аргументи за включването на някакъв механизъм от указатели (пойнтери) в Java, беше разрешаване на обратна връзка (callbacks)
- При обратна връзка, на друг обект се предоставя информация, която му позволява да се върне обратно в оригиналния обект в някакъв по-късен момент
  - Това е много мощна концепция

ОБРАТНА ВРЪЗКА

- Ако обаче обратната връзка се изпълнява с помощта на указател (адрес) се разчита на програмиста да има коректно поведение и да не злоупотребява с указателя (адреса)
- Java обаче, предпочита да бъде по-внимателна с това правомощие, така че указатели (адреси) не са включени в езика
- В Java се предлага друго решение, което се нарича closure (затваряне)
  - Осигурява се от вътрешен клас
  - Добро решение - по-гъвкаво и много по-безопасно от указателите

ПРИМЕР

```
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
       System.out.println(i);
    }
}
```

```
interface Incrementable {
    void increment();
}
```

```
class MyIncrement {
    public void increment() {
      System.out.println("Other operation"); }
    static void f(MyIncrement mi) {
      mi.increment(); }
}
```

```
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
       super.increment();
       i++;
       System.out.println(i);
    }
    private class Closure implements Incrementable {
       public void increment() {
          Callee2.this.increment();
       }
     }
     Incrementable getCallbackReference() {
          return new Closure();
      }
}
```

```
class Caller {
      private Incrementable callbackReference;
      Caller(Incrementable cbh) {
           callbackReference = cbh;
      }
      void go() {
           callbackReference.increment(); }
}
```

ПРИМЕР

```
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        System.out.println(i);
    }
}
```

```
interface Incrementable {
    void increment();
}
```

```
class MyIncrement {
    public void increment() {
      System.out.println("Other operation");
    }
    static void f(MyIncrement mi) {
      mi.increment(); }
}
```

```
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        System.out.println(i);
    }
    private class Closure implements Incrementab
        public void increment() {
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
```

```
                callbackReference = cbh; }
        void go() {
            callbackReference.increment(); }
}
```

Примерът показва разграничение между имплементиране на интерфейс във външен клас и във вътрешен клас

ПРИМЕР

```
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        System.out.println(i);
    }
}
```

```
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        System.out.println(i);
    }
    private class Closure implements Incrementable {
        public void increment() {
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
```

Callee1 е очевидно опростено решение по отношение на кода

```
}
```

```
class MyIncrement {
    public void increment() {
      System.out.println("Other operation"); }
    static void f(MyIncrement mi) {
      mi.increment(); }
}
```

Callee2 наследява от MyIncrement – този клас има метод increment(), който прави нещо, несвързано с това, което се очаква от Incrementable интерфейса

```
    void go() {
        callbackReference.increment(); }
}
```

ПРИМЕР

```
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        System.out.println(i);
    }
}
```

```
interface Incrementable {
    void increment();
}
```

```
class MyIncrement {
    public void increment() {
      System.out.println("Other operation"); }
    static void f(MyIncrement mi) {
      mi.increment(); }
}
```

```
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        System.out.println(i);
    }
    private class Closure implements Incrementable {
        public void increment() {
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
```

- Когато MyIncrement се наследи в Callee2, increment() не може да бъде препокрит за използване от Incrementable, така че сме принудени да предоставим отделна имплементация, като използваме вътрешен клас
- Когато създаваме вътрешен клас, не добавяме или променяме интерфейса на външния клас

ПРИМЕР

```
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        System.out.println(i);
    }
}
```

```
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        System.out.println(i);
    }
    private class Closure implements Incrementable {
        public void increment() {
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
```

- Всичко, с изключение на getCallbackReference () в Callee2 е private
- За да се даде възможност за връзка с външния свят, интерфейсът Incrementable е от съществено значение
- Тук можем да видим как интерфейсите позволяват пълно разделяне на интерфейс от имплементация.

```
class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) {
        callbackReference = cbh; }
    void go() {
        callbackReference.increment(); }
}
```

ПРИМЕР

```
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        System.out.println(i);
    }
}
```

```
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        System.out.println(i);
    }
    private class Closure implements Incrementable {
        public void increment() {
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
```

- Вътрешният клас Closure имплементира Incrementable, за да осигури "кука" обратно в Callee2
  - Тя е безопасна "кука"
- Всеки, който получава Incrementable референция, може, разбира се, да извика само increment () и няма друга възможност
  - За разлика от указател, който би позволил "да се вилнее"

```
class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) {
        callbackReference = cbh; }
    void go() {
        callbackReference.increment(); }
}
```

ПРИМЕР

- Стойността на обратното повикване (call back) е в неговата гъвкавост
- Можем динамично да решаваме какви методи ще извикваме по време на изпълнение
- Ползата от това ясно се вижда при графични потребителски интерфейси, където обратните повиквания се използват навсякъде за внедряване на GUI функционалност

```
class Callee1 imple
    private int i = 0;
    public void incr
        i++;
        System.out.pr
    }
}
```

```
class Callee2 exten
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        System.out.println(i);
    }
    priva
        pul
            C
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
```

- Caller получава Incrementable референция в своя конструктор
- Някога по-късно може да използва референцията за “call back" в класа Callee.

```
class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) {
        callbackReference = cbh;
    }
    void go() {
        callbackReference.increment();
    }
}
```

ПРИМЕР

```
public class Callbacks {
    public static void main(String[] args) {
        Callee1 c1 = new Callee1();
        Callee2 c2 = new Callee2();
        MyIncrement.f(c2);
        Caller caller1 = new Caller(c1);
        Caller caller2 = new Caller(c2.getCallbackReference());
        caller1.go();
        caller1.go();
        caller2.go();
        caller2.go();
    }
}
```

```
Other operation
1
1
2
Other operation
2
Other operation
3
```

НАСЛЕДЯВАНЕ ОТ ВЪТРЕШНИ КЛАСОВЕ

```
class WithInner {
    class Inner { }
}
public class InheritInner extends WithInner.Inner {
    // Няма да се компилира
    //! InheritInner() {}
    InheritInner(WithInner wi) {
        wi.super();
    }

    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
}
```

? Коментар на класа

- Понеже конструкторът на вътрешния клас трябва да се свърже с обръщение към заобикалящия обект на класа, нещата се усложняват, когато наследяваме вътрешен клас
- Проблем: „тайната“ референция на заобикалящия клас трябва да се инициализира
- В наследения клас вече няма обект по подразбиране, към който да се свърже
- Решение: Използва се синтаксис, осигуряващ явна асоциация

ПРЕДЕФИНИРАНЕ НА ВЪТРЕШНИ КЛАСОВЕ

- Ситуация
  - Създаваме вътрешен клас
  - Наследяваме от заобикалящия клас
  - Предефинираме вътрешния клас
- Възможно ли е да се предефинира вътрешен клас?
  - Изглежда мощна концепция

ПРИМЕР: BIGEGG

```
class Egg {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg.Yolk()");
        }
    }
    private Yolk y;
    public Egg() {
        System.out.println("New Egg()");
        y = new Yolk();
    }
}

public class BigEgg extends Egg {
    protected class Yolk {
        public Yolk() {
            System.out.println("BigEgg.Yolk()");
        }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
}
```

New Egg()
Egg.Yolk()

? Коментар на класа

- Конструкторът по подразбиране се синтезира автоматично от компилатора
  - Това извиква конструктора по подразбиране на базовия клас
- Не се използва „предефинираната“ версия на Yolk
  - Двата вътрешни класа са напълно самостоятелни елементи
  - Всеки със своето собствено пространство

ПРИМЕР: BIGEGG2

```
class Egg2 {
    protected class Yolk {
        public Yolk() {  System.out.println("Egg2.Yolk()");   }
        public void f() {  System.out.println("Egg2.Yolk().f()");  }
    }
    private Yolk y = new Yolk();
    public Egg2() {  System.out.println("New Egg2()");   }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}
public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() { System.out.println("BigEgg2.Yolk()");  }
        public void f() { System.out.println("BigEgg2.Yolk.f()");  }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
}
```

ПРИМЕР: BIGEGG2

? Коментар на класа

```
class Egg2 {
    protected class Yolk {
        public Yolk() {  System.out.println("Egg2.Yolk()");   }
        public void f() {  System.out.println("Egg2.Yolk().f()");
    }
    private Yolk y = new Yolk();
    public Egg2() {  System.out.println("New Egg2()");   }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}
public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() { System.out.println("BigEgg2.Yolk()");  }
        public void f() { System.out.println("BigEgg2.Yolk.f()")
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
}
```

- Възможно е явно наследяване от вътрешния клас
- BigEgg2.Yolk явно разширява Egg2.Yolk и предефинира своите методи
- Методът insertYolk() позволява на BigEgg2 да присвои един от своите собствени Yolk обекти на у референцията в Egg2, така че когато g() извика у.f() се използва предефинираната версия на f()

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

ПРИМЕР: BIGEGG2

? Резултат

```
class Egg2 {
    protected class Yolk {
        public Yolk() {  System.out.println("Egg2.Yolk()");  }
        public void f() {  System.out.println("Egg2.Yolk().f()");  }
    }
    private Yolk y = new Yolk();
    public Egg2() {  System.out.println("New Egg2()");  }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}
public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() {  System.out.println("BigEgg2.Yolk()");  }
        public void f() {  System.out.println("BigEgg2.Yolk.f()");  }
    }
    public BigEgg2() {  insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
}
```

Външното и вътрешното наследяване са независими едно от друго

Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()

ПРИЛОЖНИ РАМКИ

- Приложни рамки (ПР)(Application frameworks)
  - Клас или набор от класове, създадени за решаване на определен тип задачи
- Механизъм за прилагане на ПР
  - Наследяване на един или повече класове
  - Предефиниране на определени методи
    - Кодът на предефинираните методи адаптира общото решение (осигурено от ПР) за решаване на специфичната задача

КОНТРОЛНИ РАМКИ

- Контролни рамки (КР) (Control frameworks)
  - Специален тип ПР
  - Работят със събития
    - Познати като event-driven системи
    - Напр. GUI почти изцяло управляван от събития
  - Изпълнява събития, винаги когато станат „готови“
    - Под „готови“ могат да се разбират различни неща
    - Напр., времето на часовника
- Рамките не съдържат специфична информация за това, което контролират

ПОДХОД

- Спецификация (описание) на контролното събитие
  - Обикновено интерфейс или абстрактен клас
  - Клас “Event”
- Действителната контролна рамка, която управлява и задейства събитията
  - Клас “Controller”
- Приложение на рамката
  - В нашия пример клас “GreenhouseControls”– контрол функциите на една оранжерия

СПЕЦИФИКАЦИЯ НА СЪБИТИЯ

? Коментар на класа

```
abstract public class Event {

    private long evtTime;

    public Event(long eventTime) {
        evtTime = eventTime;
    }

    public boolean ready() {
        return System.currentTimeMillis() >= evtTime;
    }

    abstract public void action();
    abstract public String description();
}
```

Конструктор:
- Поставя времето, когато искаме да се стартира събитие

ready():
- Кога трябва да стартираме събитие
- Може да се предефинира в наследен клас
- Тогава Event може да бъде нещо различно от времето

action():
- Изпълнява се, когато настъпва събитието

description():
- Информация за събитието

КОНТРОЛНА РАМКА

- Обикновено се реализира в две части:
  - Структура за съхраняване на обекти от тип “Event”
  - Същинската контролна рамка
- В нашия случай – двете части се реализират в два класа
  - Клас “EventSet”
  - Клас “Controller”

СЪХРАНЯВАНЕ СЪБИТИЙНИ ОБЕКТИ

```
class EventSet {
    private Event[] events = new Event[ 100 ];
    private int index = 0;
    private int next = 0;
    public void add( Event e ) {
        if( index >= events.length )
            return; // В реалността генерира изключение
        events[ index++ ] = e;
    }
    public Event getNext() {
        boolean looped = false;
        int start = next;
        do {
            next = (next + 1) % events.length;
            if( start == next ) looped = true;
            if(( next == (start + 1) % events.length )
                                        && looped )
            return null;
        } while( events[ next ] == null );
        return events[ next ];
    }
    public void removeCurrent() {
        events[ next ] = null;
    }
}
```

? Коментар на класа

Съхраняват се 100 случайни събития

index: Използва се за следене на свободното пространство
next: Използва се за търсене на следващото събитие в списъка

getNext: Връща следващото събитие, като среща дупки в списъка, докато се придвижва в него

removeCurrent: Установява, че събитието не се използва повече

КОНТРОЛНА РАМКА

Коментар на класа

```
public class Controller {
    private EventSet es = new EventSet();
    public void addEvent(Event c) { es.add(c); }
    public void run() {
        Event e;
        while((e = es.getNext()) != null) {
            if(e.ready()) {
                e.action();
                System.out.println(e.description());
                es.removeCurrent();
            }
        }
    }
}
```

Мястото, където се извършва истинската работа

addEvent(): Позволява добавяне на нови събития

run(): Обхожда в цикъл EventSet, като търси Event обект, който е готов за изпълнение
За всеки такъв:

- Извиква се съответният action()
- Извежда се description()
- Отстранява събитието от списъка

ИЗПОЛЗВАНЕ НА ВЪТРЕШНИ КЛАСОВЕ

- В примера се вижда, че не знаем какво точно прави един Event
- Централен въпрос на разработването на рамки
  - Разделянето на „статичното“ (непроменяемото) от „динамичното“ (променяемото)
- Тук влизат в действие вътрешните класове
  - Цялата реализация на едно приложение, използващото КР, е в самостоятелен клас
    - Обхваща всичко, специфично за конкретното приложение
    - Вътрешните класове се използват за реализиране на различните видове action()
    - Вътрешните класове са private – приложението е напълно скрито и може да се променя безнаказано
  - Вътрешните класове предпазват приложението от това да стане неудобно за работа и поддръжка
    - Лесно можем да осъществим достъп до всеки член на външния клас

ПРИМЕР

ПРИМЕР: ОРАНЖЕРИЯ

- Приложение на КР: контрол функциите на оранжерия
- Различни действия:
  - Включване и изключване на светлина
  - Пускане и спиране на вода и отопление
  - Звукови сигнали
  - Рестартиране на системата
- КР е проектирана за лесна изолация на този различен код
  - Вътрешните класове позволяват, в рамките на самостоятелен клас, да имаме множество наследени версии на един и същ базов клас (Event)
  - За всеки тип действие се наследява нов Event вътрешен клас
  - Контролният код се записва вътре в action()

КЛАС "GREENHOUSECONTROLS"

```
public class GreenhouseControls  extends Controller {
    private boolean light = false;
    private boolean water = false;
    private String thermostat = "Day";
    private int rings;
    private class LightOn extends Event {
      public LightOn(long eventTime) {
        super(eventTime);
      }
      public void action() {
       // Поставете кода за контрол на хардуера тук,
       // за да включите светлината
        light = true;
      }
      public String description() {
        return "Light is on";
      }
    }
    private class LightOff extends Event {
      public LightOff(long eventTime) {
        super(eventTime);
      }
      public void action() {
       // Поставете кода за контрол на хардуера тук,
       // за да изключите светлината
        light = false;
       }
       public String description() {
         return "Light is off";
       }
     }
```

? Коментар на класа

light, water, thermostat, rings:

- Принадлежат на външния клас
- Въпреки това вътрешните класове имат достъп до тях

7 вътрешни класове:

- Наследяват Event
- За обработване на различните събития
- Повечето action() методи включват специфичен хардуерен контрол
- Повечето Event класове изглеждат подобни (изключение Bell и Restart)
- За светлина - LightOn, LightOff

СЪБИТИЯ

```
private class WaterOn extends Event {
      public WaterOn(long eventTime) {
         super(eventTime);
      }
      public void action() {
        // Поставете кода за контрол на хардуера тук
          water = true;
      }
      public String description() {
         return "Greenhouse water is on";
       }
}
private class WaterOff extends Event {
       public WaterOff(long eventTime) {
            super(eventTime);
        }
        public void action() {
          // Поставете кода за контрол на хардуера тук
            water = false;
        }
        public String description() {
            return "Greenhouse water is off";
         }
}
```

7 вътрешни класове:

- Наследяват Event
- За обработване на различните събития
- Повечето action() методи включват специфичен хардуерен контрол
- Повечето Event класове изглеждат подобни (изключение Bell и Restart)
- За вода - WaterOn, WaterOff

СЪБИТИЯ

```
private class ThermostatNight extends Event {
        public ThermostatNight(long eventTime) {
              super(eventTime);
        }
        public void action() {
            // Поставете кода за контрол на хардуера тук
              thermostat = "Night";
        }
        public String description() {
             return "Thermostat on night setting";
        }
}
private class ThermostatDay extends Event {
        public ThermostatDay(long eventTime) {
              super(eventTime);
        }
        public void action() {
           // Поставете кода за контрол на хардуера тук
             thermostat = "Day";
        }
        public String description() {
             return "Thermostat on day setting";
        }
}
```

7 вътрешни класове:

- Наследяват Event
- За обработване на различните събития
- Повечето action() методи включват специфичен хардуерен контрол
- Повечето Event класове изглеждат подобни (изключение Bell и Restart)
- За температура - ThermostatNight, ThermostatDay

СЪБИТИЯ

```
private class Bell extends Event {
      public Bell(long eventTime) {
          super(eventTime);
       }
       public void action() {
         // Звънете на всеки две секунди, времена
         // на "звъненето":
           System.out.println("Bing!");
        if(--rings > 0)
            addEvent(new Bell (
                System.currentTimeMillis() + 2000));
       }
       public String description() {
            return "Ring bell";
       }
}
```

7 вътрешни класове:

- Наследяват Event
- За обработване на различните събития
- Повечето action() методи включват специфичен хардуерен контрол
- Повечето Event класове изглеждат подобни (изключение Bell и Restart)
- За звукови сигнали - Bell

РЕСТАРТИРАНЕ

```
private class Restart extends Event {
        public Restart(long eventTime) {
                super(eventTime);
        }
        public void action() {
                long tm = System.currentTimeMillis();
                // Вместо да задавате хардуера твърдо, бихте
                // могли да получите информация за
                // конфигурацията от текстов файл така:
                rings = 5;
                addEvent(new ThermostatNight(tm));
                addEvent(new LightOn(tm + 1000));
                addEvent(new LightOff(tm + 2000));
                addEvent(new WaterOn(tm + 3000));
                addEvent(new WaterOff(tm + 8000));
                addEvent(new Bell(tm + 9000));
                addEvent(new ThermostatDay(tm + 10000));
                // Можем дори да добавим обект Restart!
                addEvent(new Restart(tm + 20000));
        }
        public String description() {
                return "Restarting system";
        }
}
```

Restart:

- Инициализиране на системата
- Доставя съответните събитийни обекти
- По-гъвкаво решение е вместо кодиране на събитията да се чете от файл

ИЗВИКВАНЕ НА ПРИЛОЖЕНИЕТО

```
  public static void main(String[] args) {
     GreenhouseControls gc = new GreenhouseControls();
      long tm = System.currentTimeMillis();
      gc.addEvent( gc.new Restart( tm ));
      gc.run();
   }
}
```

```
Restarting system
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Greenhouse water is off
Bing!
Ring bell
Thermostat on day setting
Bing!
Ring bell
Bing!
Ring bell
Bing!
Ring bell
Bing!
Ring bell
Restarting system
…
```

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ: “ВЪТРЕШНИ КЛАСОВЕ”