Лекция 7.b

Вътрешни и анонимни класове

Част II

Основни теми

- **Вътрешни и анонимни класове-** синтаксис и приложения.,

- ***public*** и ***private*** конструктори, скриване на source кода на приложението
- обработка на събития с вътрешни и анонимни класове (***closure , callback*** *конструкции*)
- Задачи

7b.1 Наследственост при вътрешни класове

7b.2 Closure и Callback

7b.3 Създаване и изпълнение на потребителски дефинирани събития

7b.4 Анонимни вътрешни класове

7b.5 Вътрешни класове и обработка на събития

7b.6 Приложение на Система за управление на събития

7b.7 Текстови полета и примери за обработка на събитие ActionEvent

7b.8 Общи типове събития и съответните им интерфейс-и

7b.9 Схематично представяне на модела за обработка на събития в Java

Задачи

Литература:

Bruce Eckel "Thinking in Java", 2nd ed., Prentice Hall 2000
или българското й издание "Да мислим на Java" том 1 и 2, SoftPress, 2001

7b.1 Наследственост при вътрешни класове- примери

- **Конструкторът на вътрешен клас** трябва да има референция към името на външния клас. Затова, **при онаследяване от единствен вътрешен клас**, тази референция трябва да е явно записана в производния клас (example A)
- Вътрешните класове са **изцяло отделни същности**, всяко със собствена област от имена от това на външния клас (example B)
- Явно наследяване от вътрешен клас (example C)

```
// example A shows how to inherit
// an inner class only.

class WithInner {
  class Inner {} // inner class to
inherit
}
public class InheritInner  extends
WithInner.Inner {
  // extending the inner class
  //! InheritInner() {}
  // Won't compile
  InheritInner( WithInner wi) {
    wi.super(); // reference to the
outer class required!!
  }
  public static void main(String[]
args) {
    WithInner wi = new WithInner();
    InheritInner ii = new
InheritInner(wi);
  }
}
}
```

Използвайте синтаксис

```
enclosingClassReference.super();
```

В конструктора на производния клас.

Забележете:
InheritInner онаследява само вътрешния клас, not не и външния клас

Резюме

```
// example B shows how inner classes behave in inheritance
// An inner class cannot be overriden
// like a method.

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

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

public class BigEgg extends Egg {
  public class Yolk {
  // cannot override inner class Yolk in class Egg
    public Yolk() {
      System.out.println("BigEgg.Yolk() ");
    }
  }
  public static void main(String[] args) {
    new BigEgg();
  } // What is the output? Why?
}
```

Конструкторът по подразбиране `BigEgg()` Се генерира от компилаторът, и това извиква конструкторът по подразбиране на базовия клас `Egg()` което от своя страна извиква конструкторът по подразбиране `Egg.Yolk()`

а не конструкторът по подразбиране на `BigEgg.Yolk()`

The output is:

```
New Egg()
Egg.Yolk()
```

```
// example C shows explicit inheritance of inner classes
// Inheritance from the outer class and its inner classes
class Egg2 { // base class
  protected class Yolk { // inner class in the base class
    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 {// derived outer class
  public class Yolk extends Egg2.Yolk{
                                    //derived inner class
    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();// upcast to Egg2
    e2.g();// call the overridden version of f()
  }
}
```

BigEgg2.Yolk е производен на Egg2.Yolk и презарежда методите му.
Методът insertYolk( ) позволява BigEgg2 да преобразува един от обектите си до своите Yolk обекти в референцията y на Egg2.
Така, когато g( ) извиква y.f( ) презаредената версия на f( ) се използва.
Повторното извикване на Egg2.Yolk( ) е на базовия клас конструктор в BigEgg2.Yolk. Презаредената версия на f( ) се използва при извикване на g( ).

The output is:

```
// super()
Egg2.Yolk()
New Egg2()
// insertYolk(new Yolk());
Egg2.Yolk()
BigEgg2.Yolk()
// e2.g()
BigEgg2.Yolk.f()
```

7b.2 Closures и Callbacks

- *closure* е многократно изикван обект, който може да съхранява информацията в обхвата в който е създаден
- **Вътрешният клас** и **пример** за **closure**, понеже не само има достъп до всеки член на външния клас, но и поддържа референция до този клас с разрешение да манипулира членовете му
- При ***callback***, на даден обект дава на някой друг обект **частична информация**, която му **позволява да се обади обратно** на първия клас в последващ момент

Резюме

```
// Using inner classes for callbacks

interface Incrementable {
  void increment();
}

// Very simple to just implement the
interface:
class Callee1 implements Incrementable {
  private int i = 0;
  public void increment() {
    i++;
    System.out.println(i);
  }
}

// Some other way to implement()

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

Callee1 очевидно е най-простото решение за реализиране на interface

Нека сега class MyIncrement имплементира различна реализация на метод increment( ) и тя е несвързана с логиката на interface Incrementable.

```
// Using inner classes for callbacks

// If your class must implement increment() in
// some other way, you must use an inner class

class Callee2 extends MyIncrement {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Closure  implements Incrementable {
    public void increment() { incr(); }
  }
  Incrementable getCallbackReference() {
    return new Closure();
    //returns a reference to the inner
  }
}
```

Callee2 наследява **MyIncrement** с тази логически несвързана реализация на метод **increment( )** а искаме да реализираме също и метода **increment( )на interface Incrementable**.

Тогава, **increment( )** не може да се използва с **interface** by **Incrementable**, и е необходима да се използва вътрешен клас.

Заберлежете, че всичко в **Callee2** е ***private*** освен **getCallbackReference( )**

Вътрешният **class Closure** реализира **Incrementable** само за да даде "кукичка" ***за обратно връщане в*** **Callee2**.

Който получи референция към **Incrementable** може да извика логически издържания метод **increment( )** и нищо друго

Резюме

```
// Using inner classes for callbacks

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

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();
  }
}
```

class Caller взима за аргумент референция Incrementable в конструктора се (макар че, референция към callback обекта може да стане в произволен друг момент) и впоследствие използва тази референция за "call back" обратно извикване на обекта от class Callee

7b.3 Създаване и изпълнение на потребителски дефинирани събития

Event functionality is provided by **three interrelated elements**:

1. a class that provides event data,
2. an event interface,
3. the class that raises the event.
4. The class that consumes the event

7b.3 Създаване и изпълнение на потребителски дефинирани събития

Assume event name is `Action`:

1. A class that provides event data

`ActionEventArgs` or `ActionEvent`

2. An event interface

`ActionEventHandler` or `ActionListener`

3. The class that raises the event.

`ActionEventSource` or any other custom defined name

3. The class that consumes the event

`ActionEventConsumer` or any other custom defined name

Резюме

```
public interface ActionListener { // defines action event
// This is just a regular method so it can return something
// or take arguments if you like.
    public void actionPerformed (ActionEvent args) ;
}

public class ActionEvent {
// This is a class that defines the event arguments
// takes arguments as you like.
    private MyArg arg; // some arguments

    public ActionEvent (MyArg arg) {
       setMyArg(arg);
    }

    public MyArg getMyArg() {
     return arg; // return a copy of this.arg
     }

    private void setMyArg(MyArg value) {
     // validate and set arg;
     this.arg = value;
    }

}
```

Резюме

```
public class ActionEventSource  // event source
{
    private ActionListener ie;
    // ActionListener may be also public available!
    private boolean onAction;
    public EventSource(ActionListener event)
    {
    // Save the event object for later use.
    ie = event;
    // Nothing to report yet.
    onAction = false;
    }
    //...
    public void addActionListener(ActionListener al){
      ie = al;
    }
    public void doWork ()
    {
    // Check the predicate, which is set elsewhere.
    if (onAction )
        {
        // Signal the even by invoking the interface's method.
        // Create MyArg object and pass it to the event handler
        if (ie != null) // event is handled!!!
          ie.actionPerformed( new ActionEvent(new MyArg()));
          // the event is fired!
        }
    //...
    }
    // ...
}
```

Ако ActionListener ie е public, **то** ie **може да се инициализира директно в** CallMe **класовете на** ActionListener **обект, който е инстанция на вътрешен клас в клас** CallMe

Резюме

```
//class that handles the event
public class CallMe implements ActionListener
{
    private ActionEventSource en;
    // component that fires the event

    public CallMe ()
    {
    // hook the event handler to the event source.
      en = new ActionEventSource(this);

    }
    // Define the actual handler for the event.
    public void actionPerformed (EventArgs args)
    {
    // Wow!  Something really interesting must have occurred!
    // get args
    // and
    // Do something...
    }
    //...
}
```

7b.4 Анонимни вътрешни класове

- Конструкция за скриване на имена и организация на код
- Използва се за описване на събития изискващи малко код за описанието им


```
//: c08:Parcel6.java
// A method that returns an anonymous inner class.
public class Parcel6
 {
    public Contents cont()
    {
      return new Contents()
       {
         private int i = 11;
         public int value()
           {
              return i;
           }
       };
   // Semicolon required in this case
   }
   public static void main(String[] args)
     {
       Parcel6 p = new Parcel6();
       Contents c = p.cont();
     }
 }

// Contents is created using a default constructor
// This is required for the above to compile
interface Contents {
   int value();
}
/* or is required for the above to compile
class Contents {
  int value(){ return 0;}
}
*/
```

Метод *cont( )* комбинира връщане на стойност и дефиниране на клас, който описва връщаната стойност!

Допълнително, класът е анонимен *anonymous*– няма име.

По друг начин казано: "*създай обект от безименен клас който е производен на Contents.*"

Това е *съкратен запис* на:

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

```
//: c08:Parcel6.java
// A method that returns an anonymous inner class.
public class Parcel6
 {
    public Contents cont()
    {
      class MyContents implements Contents
      {
         private int i = 11;
         public int value() { return i; }

       }

     return new MyContents();
    }
   // Semicolon required in this case
   }
   public static void main(String[] args)
     {
       Parcel6 p = new Parcel6();
       Contents c = p.cont();
     }
 }

// Contents is created using a default constructor
// This is required for the above to compile
interface Contents {
   int value();
}
/* or is required for the above to compile
class Contents {
  int value(){ return 0;}
}
*/
```

Метод *cont( )* комбинира връщане на стойност и дефиниране на клас, който описва връщаната стойност!

Допълнително, класът е анонимен *anonymous*– няма име.

По друг начин казано: "*създай обект от безименен клас който е производен на Contents.*"

Това е *съкратен запис* на:

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

```
//: c08:Parcel7.java Using an argument
// The base class needs a constructor
// with an argument.
public class Parcel7
 {
  private int count = 2;
  public Wrapping wrap(int x, final String
dest)
   {
    // Base constructor call:
    return new Wrapping(x)
      {
        private String label = dest;
        public int value()
          {
           return super.value() * count;
          }
    }; // Semicolon required
  }
  public static void main(String[] args)
  {
   Parcel7 p = new Parcel7();
   Wrapping w = p.wrap(10);
  }
 }
 public class Wrapping
 {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
}
```

Анонимен клас с конструктор-предаваме аргумент на базовия коструктор, представен тук като x в new Wrapping(x).

Анонимният клас не може да има конструктор където да използвате както обикновено super( )

Ако искате да използвате обект в анонимен вътрешен клас, където обектът е реферирана с локална променлива, дефинирана извън този клас, то компилаторът изисква тази променлива да.

(примерно, dest е final в списъка от аргументи на метода дефиниращ анонимния клас!)

Анонимният клас има пълен достъп до данните на външния клас (пример с count !)

7b.4 Анонимни вътрешни класове

Effectively final local variable.

Anonymous classes can access **instance** and **`static`** variable of the outer class. This is called ***variable capture***. Instance and **`static`** variables may be **used and changed without restriction** in the body **of an anonymous class**

The use of **local variables**, however, is more **restricted**: **capture of local variables is not allowed unless they are *effectively final* *in* *JDK 8***, i.e. once a local variable is captured inside an anonymous class or inner class, its initial value cannot be changed even though it is not declared as **`final`**. Missing to declare an **effectively `final`** variable as **`final`** would not cause a compilation failure

```
public class CaptureTest {
    private static int count = 5;// captured inside the annonymous class
    private String str = "Captured string";
    public void method(String dest) {//dest is effectively final
        int localVar = 5; //localVar is effectively final
        Wrapping wr = new Wrapping(local) {
            private String label = dest;
            public int value() {
                //dest = "";//// not allowed, dest variable is captured!!
                return super.value() * count;
            }
            public String toString(){
                return str;
            }
        }; // Semicolon required
        // localVar = 7;// not allowed, localVar variable is captured!!
        count  = 8; // allowed
        str    = "New captured string";//allowed
    }
}
class Wrapping {
    private int i;
    public Wrapping(int x) {
        i = x;
    }
    public int value() {
        return i;
    }
}
```

```
//: c08:Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class.

public class Parcel9 {
  public Destination dest(final String dest,
                          final float price)
  {
   return new Destination()
   {
      private int cost;
      // Instance initialization for each object:
      {
        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("Tanzania", 107b.395F);
    System.out.println(d.readlabel());
  }
}
```

Резюме

За инициализация на данни в анонимен клас се използва блок от код, вместо конструктор за общо ползване.

Тук

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

Какво се изпълнява и защо, ако

```
public class Destination {
 String readLabel()
    {
    return "SOS";
    }
}
```

7b.5 Вътрешни класове и обработка на събития

- *Application framework* е class или съвъкупност от класове създадени за решаване определена приложна задача.
- *Application framework* се използва като се наследи от един или повече от класове и се предефинират методите на наследените класове.
- *Control framework* е частен случай на *Application framework* в случай на задача изискваща реагиране на събития и се нарича **система, управлявана от събития**.
- Едно от най- важните й приложения important е в програмиране на graphical user interface (GUI), която изцяло управлявана от събития

7b.5 Вътрешни класове и обработка на събития

- **Пример**: Система за управление чиято **задача е да изпълнява събития**, които са готови “ready” за изпълнение.
- Под “ready” ще говорим в смисъл на **готовност за изпълнение по отношение на системното време**.
- Примерът дава **общ скелет за изграждане на такъв тип система** за управление.
- Използва се **abstract** class вместо **interface** за описване на произволно управляващо събитие поради нужда изпълнението да се синхронизира със системното време

Резюме

```
//: c08:controller:Event.java
// The common methods for any control event.
package c08.controller;

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();
}
```

7b.5 Вътрешни класове и обработка на събития

- Конструкторът на **class Event** капсулира времето на създаване на събитието с оглед създаване на определена наредба в изпълнението на събития **Event**
- Методът **ready( )** казва дали е време да се изпълни събитието. Допълнително, **ready( )** би могло да се предефинира в производен клас на **Event** и да се използва друг критерий за готовност

- Методът **action( )** е методът който се изпълнява след като обектът **Event** е **ready( )**, и **description( )** дава текстово описание на обектът **Event**.

7b.5 Вътрешни класове и обработка на събития

- Следващите класове съдържат същинската част на система за управление на събития. Първият клас **EventSet** е помощен – целта му е на дефинира **Event** обектите.
- **EventSet** примерно е контейнер за 100 **Event**s.
- **index** се използва да следи за свободна памет за следващо събитие,
- **next** реферира следващото събитие **Event** в списъка с метода **getNext( )**,
- Обектите **Event** се премахват от списъка чрез **removeCurrent( )**) след като се изпълнят и **getNext( )** може да открие “дупки” в списъка при преминаване към следващо събитие

Резюме

```
// Along with Event, the generic: Controller.java
// framework for all control systems:
package c08.controller;
// This is just a way to hold Event objects.
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; // (In real life, throw exception)
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
      // See if it has looped to the beginning:
      if(start == next) looped = true;
      // If it loops past start, the list
      // is empty:
      if((next == (start + 1) % events.length)
         && looped)
        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() {
    events[next] = null;
  }
}
```

Резюме

```
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();
      }
    }
  }
}
```

7b.5 Вътрешни класове и обработка на събития

- Забележете **`removeCurrent( )`** инициализира референцията към събитието като **`null`**. Така ресурсите заемани от това събитие се освобождават.
- **`Controller`** осъществява истинската работа по управление на събитията
- Използва контейнера **EventSet** за съхраняване на обектите **`Event`** като метод **`addEvent( )`** позволява д асе добавят нови събития към списъка
- Най- важен е методът **`run( )`, който обхожда списъка в EventSet** търсейки за обект **`Event`** готов **`ready( )`** да се изпълни. Когато обект **`Event`** е **`ready( )`**, се изпълнява **`action( )`** , отпечатва се **`description( )`,** и обектът **`Event`** се премахва от списъка

7b.5 Вътрешни класове и обработка на събития

- **Най- важното дотук е абстрактната дефиниция за това какво един обект `Event` прави.**
- **Основен принцип в моделирането “*отделяне на нещата, които са неизменни от тези, които се променят*.”**
- **<u>На практика</u> изразяваме различни действия със създаване на различни производни класове на `class Event`.**

7b.5 Вътрешни класове и обработка на събития

Това е ролята на вътрешните класове – те позволяват:

- **Да се създаде изцяло нова реализация на система за управление в един единствен клас като се капсулира в нея всичко необходимо за нейната реализация**
- **Използваме вътрешните класове за изразяване на различните видове действия на `action()` за решаване на задачата. В допълнение използваме `private` вътрешни класове, чиято имплементация остава скрита.**
- **Вътрешните класове осигуряват достъп до членовете на външния клас и кодът остава логически компактен.**

7b.6 Приложение на Система за управление на събития

Това е ролята на вътрешните класове – те позволяват:

- **Да се създаде изцяло нова реализация на система за управление в един единствен клас като се капсулира в нея всичко необходимо за нейната реализация**
- **Използваме вътрешните класове за изразяване на различните видове действия на action( ) за решаване на задачата. В допълнение използваме private вътрешни класове, чиято имплементация остава скрита.**
- **Вътрешните класове осигуряват достъп до членовете на външния клас и кодът остава логически компактен.**

7b.6 Приложение на Система за управление на събития

Да разгледаме частно приложение на система за управление на събития в Парник , където действията са: включване на светлина, вода, включване на термотат, включване на звуков сигнал, и рестартиране на системата.

Вътрешните класове реализират различна версия на един и същ базов клас, `Event`, като това става в един единствен клас. Всяко едно събитие се дефинира във вътрешен клас, който онаследява от `class Event` и предефинине на метод `action()`.

- **Както в общия случай на приложна система class `GreenhouseControls` е производен на клас `Controller`**

Начало на класът за управление на събития в парник

Пример за дефиниране на събитие за включване на светлина

```
//: c08:GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import c08.controller.*;

public class GreenhouseControls
                    extends Controller
{
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";

  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String description() {
      return "Light is on";
    }
  }
```

Резюме

```
private class LightOff extends Event {
   public LightOff(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here to
     // physically turn off the light.
     light = false;
   }
   public String description() {
     return "Light is off";
   }
 }
 private class WaterOn extends Event {
   public WaterOn(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     water = true;
   }
   public String description() {
     return "Greenhouse water is on";
   }
 }
```

Резюме

```
private class LightOff extends Event {
   public LightOff(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here to
     // physically turn off the light.
     light = false;
   }
   public String description() {
     return "Light is off";
   }
 }
 private class WaterOn extends Event {
   public WaterOn(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     water = true;
   }
   public String description() {
     return "Greenhouse water is on";
   }
 }
```

Резюме

```
private class WaterOff extends Event {
   public WaterOff(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     water = false;
   }
   public String description() {
     return "Greenhouse water is off";
   }
 }
 private class ThermostatNight extends Event {
   public ThermostatNight(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     thermostat = "Night";
   }
   public String description() {
     return "Thermostat on night setting";
   }
 }
 private class ThermostatDay extends Event {
   public ThermostatDay(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     thermostat = "Day";
   }
   public String description() {
     return "Thermostat on day setting";
   }
 }
```

Резюме

```
// An example of an action() that inserts a
// new one of itself into the event list:
private int rings;

private class Bell extends Event {
  public Bell(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Ring every 2 seconds, 'rings' times:
    System.out.println("Bing!");
    if(--rings > 0)
      addEvent(new Bell(
        System.currentTimeMillis() + 2000));
  }
  public String description() {
    return "Ring bell";
  }
}
```

```
private class Restart extends Event {
   public Restart(long eventTime) {
     super(eventTime);
   }
   public void action() {
     long tm = System.currentTimeMillis();
     // Instead of hard-wiring, you could parse
     // configuration information from a text
     // file here:
     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));
     // Can even add a Restart object!
     addEvent(new Restart(tm + 20000));
   }
   public String description() {
     return "Restarting system";
   }
 }
```

Пример за генериране "пакет" от всички събития

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

Резюме

Край на класът за управление на събития в парник

Стартиране на системата за управление на събитията

7b.7 Текстови полета и примери за обработка на събитие ActionEvent

- **class** `JTextComponent`
  - **Базов клас на class** `JTextField`
  - **Базов клас на class** `JPasswordField`
    - **Добавя** `echo` **символ (звезда) за скриване на текста при печатането му в текстовото поле**
  - **Позволява на потребителя да въвежда текст, когато компонентата получи фокус.**
  - **Поражда събитието** `ActionEvent`**, което се обработва с метод** `actionPerformed()`
  - **За еднообразие, всеки клиентски клас, който обработва** `ActionEvent` **трябва да реализира метод** `actionPerformed()` **на** `interface ActionListener`

Outline

TextFieldFrame .java

(1 от 3)

```
1  // Fig. 11.9: TextFieldFrame.java
2  // Demonstrating the JTextField class implementing ActionEvent interface.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame implements ActionListener
12 {
13    private JTextField textField1; // text field with set size
14    private JTextField textField2; // text field constructed with text
15    private JTextField textField3; // text field with text and size
16    private JPasswordField passwordField; // password field with text
17
18    // TextFieldFrame constructor adds JTextFields to JFrame
19    public TextFieldFrame()
20    {
21       super( "Testing JTextField and JPasswordField" );
22       setLayout( new FlowLayout() ); // set frame layout
23
24       // construct textfield with 10 columns
25       textField1 = new JTextField( 10 );
26       add( textField1 ); // add textField1 to JFrame
27
```

Реализира интерфейс за обработка на събитието ActionEvent

Създава обект JTextField

Outline

TextFieldFrame .java

(2 от 3)

```
28      // construct textfield with default text
29      textField2 = new JTextField( "Enter text here" );
30      add( textField2 ); // add textField2 to JFrame
31
32      // construct textfield with default text and 21 columns
33      textField3 = new JTextField( "Uneditable text field", 21 );
34      textField3.setEditable( false ); // disable editing
35      add( textField3 ); // add textField3 to JFrame
36
37      // construct passwordfield with default text
38      passwordField = new JPasswordField( "Hidden text" );
39      add( passwordField ); // add passwordField to JFrame
40
41      // register event handlers
42
43      textField1.addActionListener( this );
44      textField2.addActionListener( this  );
45      textField3.addActionListener( this );
46      passwordField.addActionListener( this  );
47   } // end TextFieldFrame constructor
48
49
50
51
52   // process text field events
53   public void actionPerformed( ActionEvent event )
54      {
55         String string = ""; // declare string to display
56
```

Създава обект `JTextField`

Създава обект `JTextField` без да позволява

Създава обект `JPasswordField`

Реализира обработка на `actionPerformed` в текущия обект `this` (регистрира метод за обработка на събитие към Swing компонетите)

Дефинира метод `actionPerformed` за обработка на събитието

```
57          // user pressed Enter in JTextField textField1
58          if ( event.getSource() == textField1 )
59             string = String.format( "textField1: %s",
60                event.getActionCommand() );
61
62          // user pressed Enter in JTextField textField2
63          else if ( event.getSource() == textField2 )
64             string = String.format( "textField2: %s",
65                event.getActionCommand() );
66
67          // user pressed Enter in JTextField textField3
68          else if ( event.getSource() == textField3 )
69             string = String.format( "textField3: %s",
70                event.getActionCommand() );
71
72          // user pressed Enter in JTextField passwordF
73          else if ( event.getSource() == passwordField )
74             string = String.format( "passwordField: %s",
75                new String( passwordField.getPassword() ) );
76
77          // display JTextField content
78          JOptionPane.showMessageDialog( null, string );
79       } // end method actionPerformed
80    } // end private inner class TextFieldHandler
81 } // end class TextFieldFrame
```

Проверява дали събитието е породено от textField1

Прочита текст въведен в тестовото поле

Проверява дали събитието е породено от textField2

Прочита текст въведен в тестовото

Проверява дали събитието е породено от textField3

Прочита текст въведен в тестовото поле

Проверява дали събитието е породено от passwordField

Прочита паролата въведена в passwordField

TextFieldFrame.java

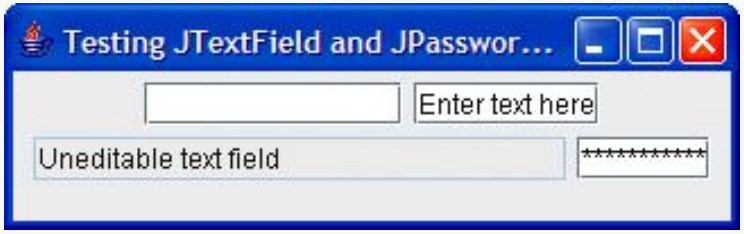
(3 от 3)

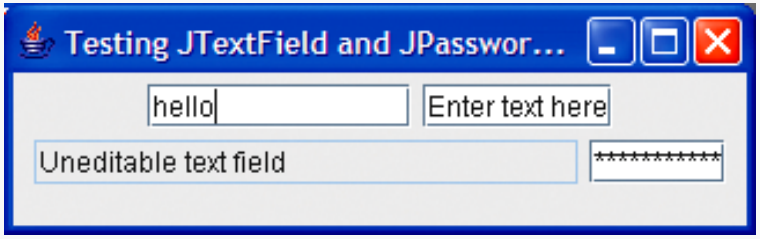
Outline

TextFieldTest .java

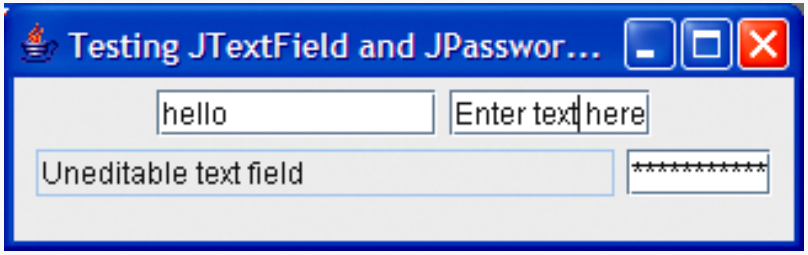
(1 от 2)

```
1  // Fig. 11.10: TextFieldTest.java
2  // Testing TextFieldFrame.
3  import javax.swing.JFrame;
4
5  public class TextFieldTest
6  {
7     public static void main( String args[] )
8     {
9        TextFieldFrame textFieldFrame = new TextFieldFrame();
10       textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textFieldFrame.setSize( 325, 100 ); // set frame size
12       textFieldFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class TextFieldTest
```



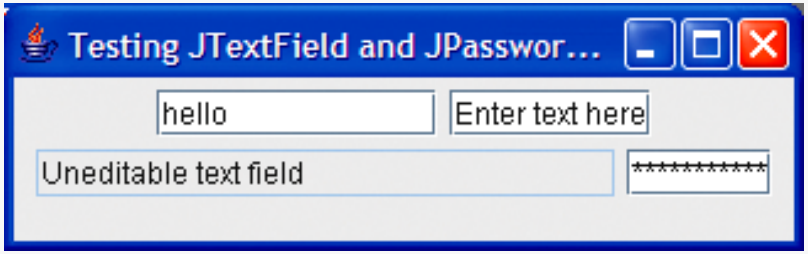


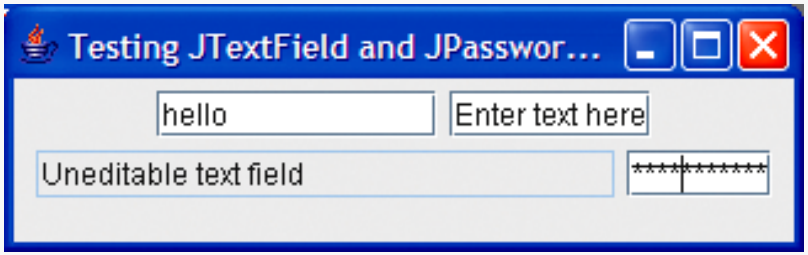
Outline



`TextFieldTest.java`

(2 от 2)





7b.7 Текстови полета и въведение към обработка на събитие ActionEvent с вложени класове

- **Вложен (*вътрешен*) клас е клас, чиято дефиниция се съдържа изцяло в (вътре) дефиницията на друг клас**
- **При обработка на събития вътрешен клас обикновено се използва да "пакетира" метод(ите) за обработка на събитието**
- **За всяко събитие има определен `interface`, който определя `методите`, с които може да се обработва това събитие.**
- **Обект от вътрешния клас, който реализира дадения интерфейс, трябва да се регистрира към съответната компонента, за да може тази компонента да реагира на събитието, по начина по който са реализирани методите на интерфейса**

Outline

TextFieldFrame .java

(1 от 3)

```
1  // Fig. 11.9: TextFieldFrame.java
2  // Demonstrating the JTextField class.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13    private JTextField textField1; // text field with set size
14    private JTextField textField2; // text field constructed with text
15    private JTextField textField3; // text field with text and size
16    private JPasswordField passwordField; // password field with text
17
18    // TextFieldFrame constructor adds JTextFields to JFrame
19    public TextFieldFrame()
20    {
21       super( "Testing JTextField and JPasswordField" );
22       setLayout( new FlowLayout() ); // set frame layout
23
24       // construct textfield with 10 columns
25       textField1 = new JTextField( 10 );
26       add( textField1 ); // add textField1 to JFrame
27
```

Външният клас **не имплементира** `ActionListener`. Събитието `ActionEvent` ще **се обработва във вътрешен клас**

Създава обект `JTextField`

Outline

TextFieldFrame .java

(2 от 3)

```
28      // construct textfield with default text
29      textField2 = new JTextField( "Enter text here" );
30      add( textField2 ); // add textField2 to JFrame
31
32      // construct textfield with default text and 21 columns
33      textField3 = new JTextField( "Uneditable text field", 21 );
34      textField3.setEditable( false ); // disable editing
35      add( textField3 ); // add textField3 to JFrame
36
37      // construct passwordfield with default text
38      passwordField = new JPasswordField( "Hidden text" );
39      add( passwordField ); // add passwordField to JFrame
40
41      // register event handlers
42      TextFieldHandler handler = new TextFieldHandler();
43      textField1.addActionListener( handler );
44      textField2.addActionListener( handler );
45      textField3.addActionListener( handler );
46      passwordField.addActionListener( handler );
47   } // end TextFieldFrame constructor
48
49   // private inner class for event handling
50   private class TextFieldHandler implements ActionListener
51   {
52      // process text field events
53      public void actionPerformed( ActionEvent event )
54      {
55         String string = ""; // declare string to display
56
```

Създава обект `JTextField`

Създава обект `JTextField` без да позволява

Създава обект `JPasswordField`

Създава обект за обработка на ActionEvent event handler

Регистрира обект за обработка на ActionEvent към всяка компонента

Създава **вътрешен клас реализиращ метод за обработка** на събитието определен от interface `ActionListener`

Дефинира метод `actionPerformed` за обработка на събитието

```
57          // user pressed Enter in JTextField textField1
58          if ( event.getSource() == textField1 )
59             string = String.format( "textField1: %s",
60                event.getActionCommand() );
61
62          // user pressed Enter in JTextField textField2
63          else if ( event.getSource() == textField2 )
64             string = String.format( "textField2: %s",
65                event.getActionCommand() );
66
67          // user pressed Enter in JTextField textField3
68          else if ( event.getSource() == textField3 )
69             string = String.format( "textField3: %s",
70                event.getActionCommand() );
71
72          // user pressed Enter in JTextField passwordF
73          else if ( event.getSource() == passwordField )
74             string = String.format( "passwordField: %s",
75                new String( passwordField.getPassword() ) );
76
77          // display JTextField content
78          JOptionPane.showMessageDialog( null, string );
79       } // end method actionPerformed
80    } // end private inner class TextFieldHandler
81 } // end class TextFieldFrame
```

Проверява дали събитието е породено от textField1

Прочита текст въведен в тестовото поле

Проверява дали събитието е породено от textField2

Прочита текст въведен в тестовото

Проверява дали събитието е породено от textField3

Прочита текст въведен в тестовото поле

Проверява дали събитието е породено от passwordField

Прочита паролата въведена в passwordField

TextFieldFrame.java

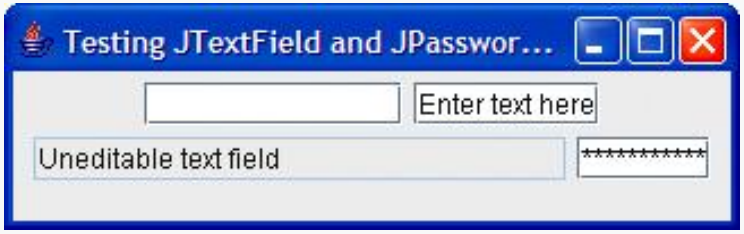
(3 от 3)

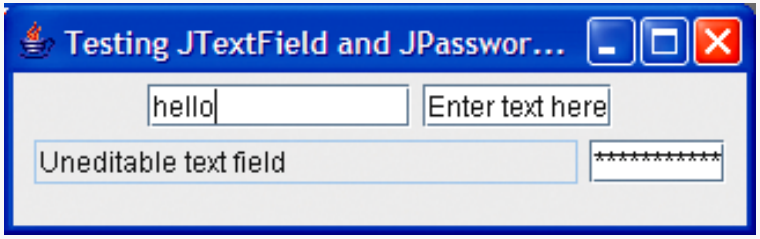
Outline

TextFieldTest.java

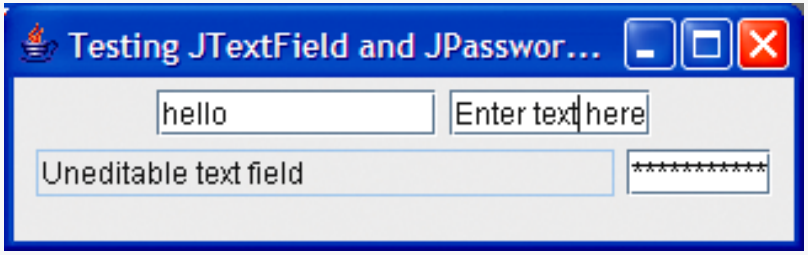
(1 от 2)

```
1  // Fig. 11.10: TextFieldTest.java
2  // Testing TextFieldFrame.
3  import javax.swing.JFrame;
4
5  public class TextFieldTest
6  {
7     public static void main( String args[] )
8     {
9        TextFieldFrame textFieldFrame = new TextFieldFrame();
10       textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textFieldFrame.setSize( 325, 100 ); // set frame size
12       textFieldFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class TextFieldTest
```



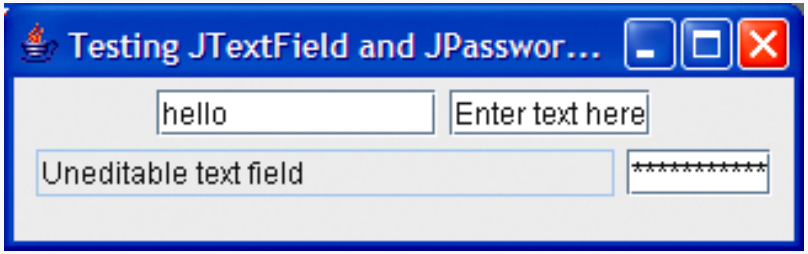


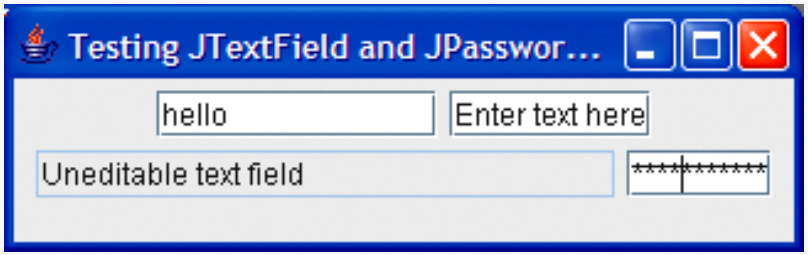
Outline



`TextFieldTest.java`

(2 от 2)





Стъпки, необходими за обработване на събития от графичния интерфейс

Следните стъпки са необходими:

1. **Създаване на клас за "*пакетиране*" на метод за обработка на събитието- той имплементира стандартен (библиотечен) интерфейс съответстващ по име на събитието**

   (събитие `ActionEvent` ←→ интерфейс `ActionListener`)

2. **Реализиране на методите на интерфейса в този клас съобразно събитието, което ще се обработва**

   (например `public void actionPerformed()`)

3. **Регистриране на обект (например, `actionHandler)` от класа с реализацията метода(ите) за обработка на събитието към графичната компонента (например, `componentRefVar)`, която ще реагира на (*слуша за*) събитието (например, `ActionEvent)`**

(например
`componentRefVar.addActionListener(actionHandler)` **)**

Използване на вложен клас за реализиране на обработката на събитието накратко

- **Класове на горно ниво**
  - **Не са вложени в други класове**
- **Вложени класове**
  - **Декларират се вътре в друг клас**
  - **не-`статично` вложените класове се наричат вложени "вътрешни" (`inner`) класове**
  - **Най- често се използват при обработка на събития**
  - **Всеки вложен клас има директен достъп до членовете на обхващащия го клас, дори и когато тези членове са `private`.**

Използвана на вложен клас за обработка на събитие

- *JTextField* и *JPasswordField* компоненти
  - **Натискане на бутон на мишката във всяко от тези компоненти води до пораждане на събитието *ActionEvent***
    - **Обработката на това събитие се извършва, чрез класове имплементиращи *ActionListener* interface**
    - **Този интерфейс изисква реализирането на метод *actionPerformed()*, който служи за обработка на този тип събитие**

Регистриране на обработчика на събитието

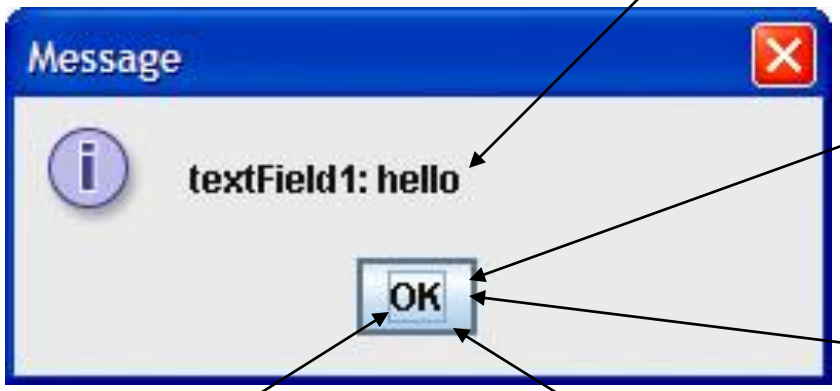
- **Регистрирането се извършва като се**
  - **Извиква метод `addActionListener` за регистриране на обект от типа на интерфейса `ActionListener` (преобразуване нагоре до типа на интерфейса!)**
  - **`ActionListener` обектът започва да "*слуша*" за събития от типа на *`ActionEvent`***
  - ***Пропускането да се регистрира обработчика на събитието, не позволява на приложението да реагира на това събитие***

Използване на метода `actionPerformed`

- **Взима за аргумент обект от породеното събитие (*`ActionEvent`*)**
  - **този обект се предава на `actionPerformed` от източника на събитието (Event source)**
  - **източникът на събитието е компонентата, от която е породила събитието (създала е обектът от клас *`ActionEvent`*)**
  - **Референция към източника на събитието се съхранява в обекта на събитието и може да се получи с метода `getSource`**
  - **Когато източникът е текстово поле `JTextField`, текстът в него се получава с `getActionCommand` или `getText`**
  - **Текст от `JPasswordField` се получава с using `getPassword`**

Обработване на събитие

5. Компонентата подава обекта на събитието (ActionEvent) като аргумент на метода (actionPerformed()) от обект за обработка на събитието (handler) и изпълнява този метод



4. По кода на сигнала компонентата открива регистрирания към нея обект за обработка на събитието (handler)

3. По кода на сигнала се компонентата създава обект на събитие (ActionEvent)

7b. При натискане на бутона мишката изпраща сигнал на Операционната система

2. ОС изпраща съобщение с кода на сигнала до компонентата, която "*слуша*" за такъв сигнал

7b.8 Общи типове събития и съответните им интерфейс-и

- **Типове събития**
  - **Производни на клас** `AWTEvent`
  - **Повечето са декларирани в package** `java.awt.event`
  - **Други, спесифични за Swing компоненти са декларирани в** `javax.swing.event`

7b.8 Общи типове събития и съответните им интерфейс-и

- **Модел на делегиране при обработка на събитията**
  - **Източникът на събитието е тази компонента с която потребителят взаимодейства**
  - **Обектът на събитието се създава и съдържа информация за събитието, което е породено**
  - **Слушател (listener) на събитието получава съобщение за станалото събитие т.е. изпълнява съответен метод за обработка на събитието**

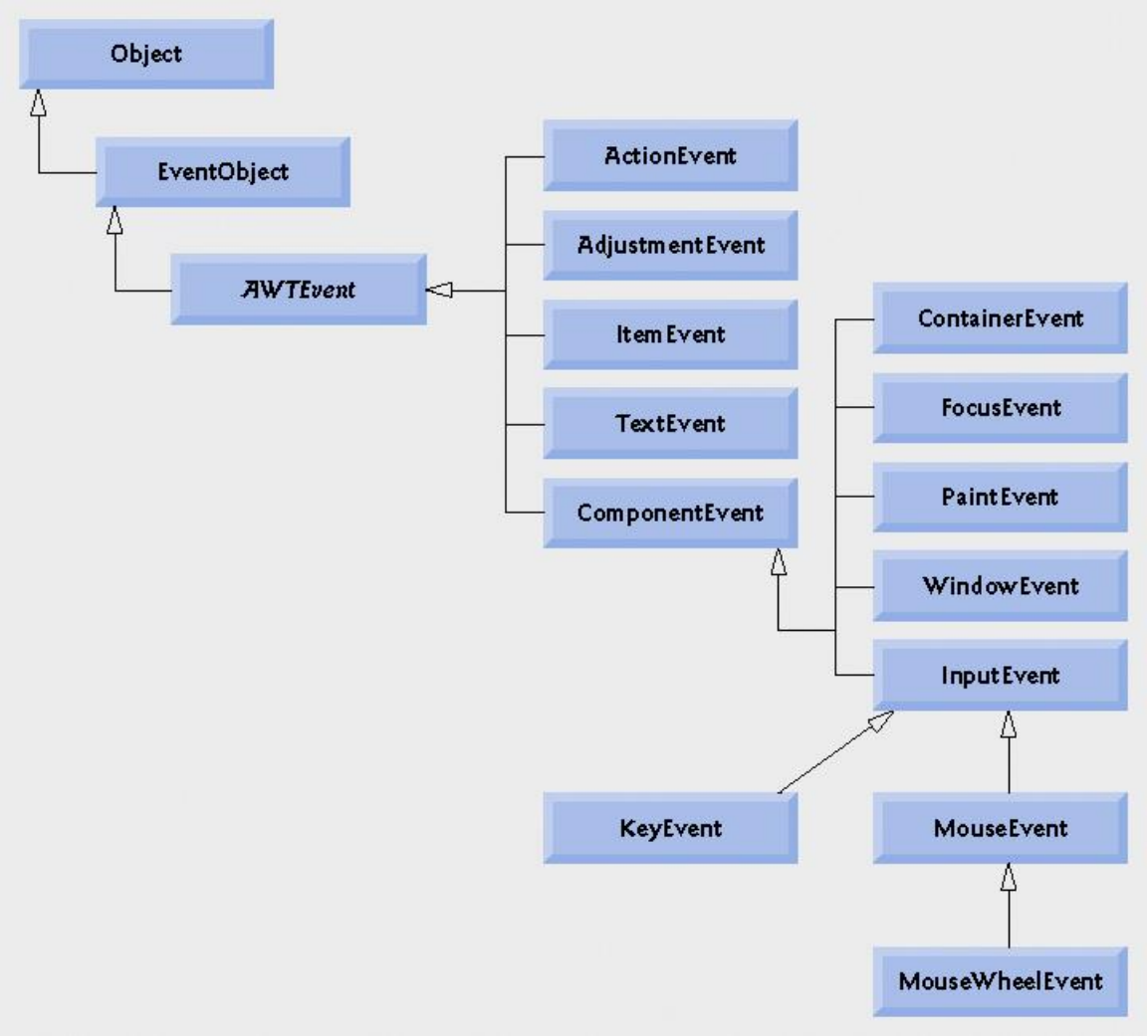


Fig. 17b.11 | Някой от основните събития в package `java.awt.event`.

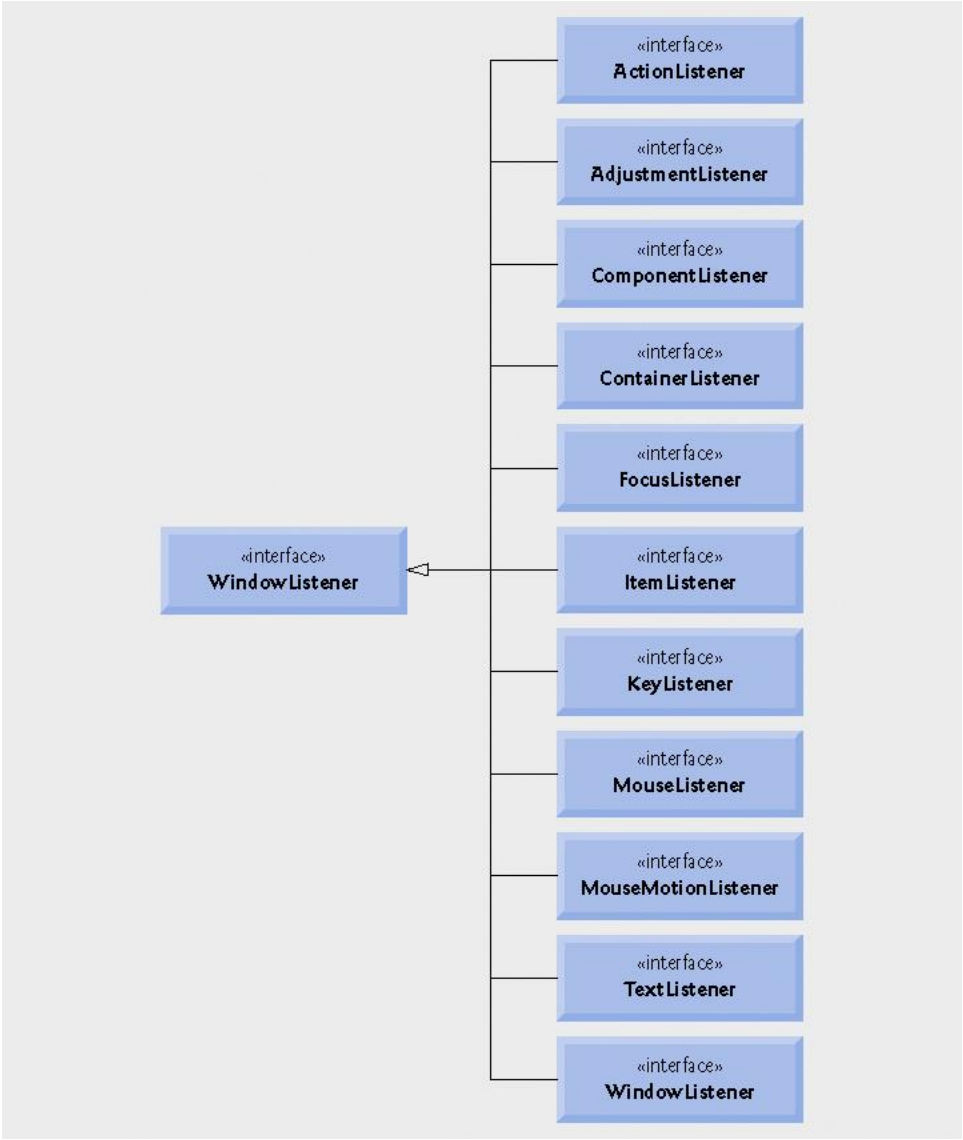


Fig. 17b.12 | Някой от основните събития в package `java.awt.event`.

7b.9 Схематично представяне на модела за обработка на събития в Java

- **Оставащи въпроси за изясняване**
  - **Как се регистрира обектът от класа за обработка на събитието?**
  - **Как графичната компонента знае да изпълни `actionPerformed` а не някой друг метод?**

Регистриране на събитията

- **Всеки обект от клас производен на `JComponent` има данна `listenerList` от `class EventListenerList`, която има списък с референции към всички слушатели (обработчици на събития) регистрирани с тази компонента**
- **Този списък е в съответствие с уникалния код на сигнала подаван от ОС при възникване на събитието**
  - **Например, при изпълнение на**

```
textField7b.addActionListener( handler );
```

- **се добавя нова референция към данната `listenerList` на обекта `textField1`**
- **обектът `textField1` ще приема сигнали от ОС (чрез JVM) при възникване на `ActionEvent`**

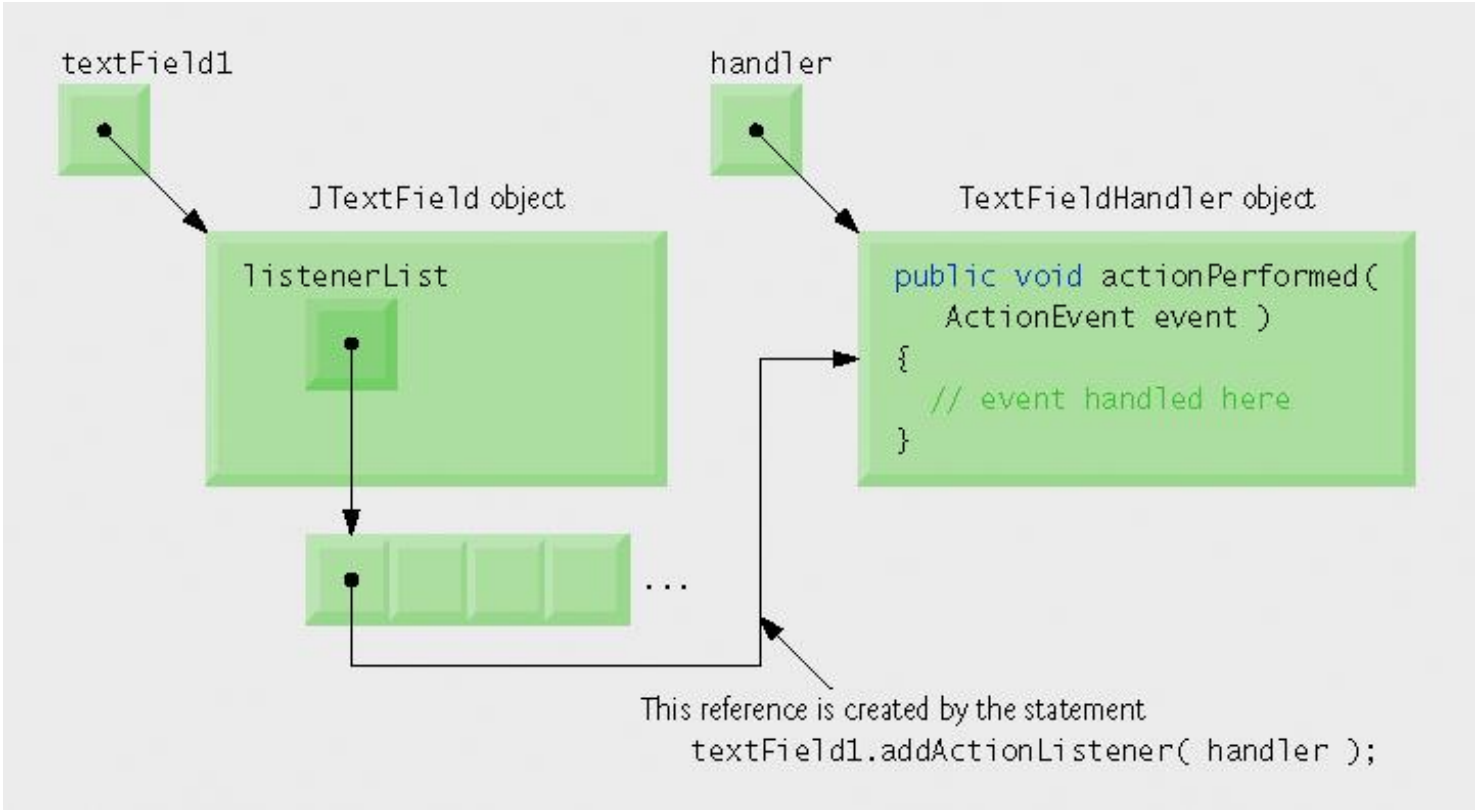


Fig. 17b.13 | Регистриране на обект за обработка на събитие към `JTextField` `textField7b`.

Изпълнение на метода за обработка

- **Събитията се прехвърлят за обработка на обекти от съответния им тип**
  - **Всяко събитие прихванато от ОС има уникален индентификатор за разпознаване на типа му**
  - **Графичната компонента получава този идентификатор от ОС и го използва да открие регистриран към нея обект за обработка на това събитие (проверява се дали на съответното място в списъка *`listenerList има референция към обект`*)**
  - **Ако се открие регистриран обект за обработка на събитието, то се изпълнява метод в този обект, съобразно имплементирания в него интерфейс**
- `MouseEvent` **се обработват от** `MouseListener` **и** `MouseMotionsListener` **обекти**
- `KeyEvent` **се обработват от** `KeyListener` **обект и пр.**

Задачи

Задача 1

Напишете ***class A*** , който **има** само **конструктор** за общо ползване (няма конструктор по подразбиране). **Напишете** ***class B***, който има метод ***getA()***, връщащ референция до ***class A*** – нека връщаният обектът се създава като **използвате анонимен клас**, наследяващ от ***class A.*** **Напишете** приложение на ***Java*** за тестване на метод ***getA().***

Задача 2

В задача 1 добавете ***public void tryMe(String txt)*** метод към ***class A*** и го предефинирайте (***override***), в анонимния клас дефиниран в метода ***getA()*** на ***class B.*** **Напишете** приложение на ***Java*** за тестване на метод ***tryMe(String txt)***

Задача 3

Напишете class A , който има private данна и **private** метод. **Напишете** вътрешен клас с метод, който **променя данната** на външния клас и **изпълнява метода** на външния клас. **Напишете** втори метод във външния клас, който **създава** обект от вътрешния клас, **извиква** метода на вътрешния клас и **проверява** как се е променила данната на външния клас. **Напишете** приложение на **Java** за тестване и обяснете резултата.

Задачи

Задача 4

Напишете ***UML диаграма за системата за управление на събития*** **class** **`GreenhouseControls`**. Напишете в този контролер вътрешни класове за дефиниране на събитие **`turnFansOn`** и **`turnFansOff`** (включване и изключване на вентилатора)