Ereignisverarbeitung in Java

Michael Dienert

18. Juli 2012

Inhaltsverzeichnis

1 Kommandozeile oder grafische Oberfläche **2**

2 Ereignisse, Ereignisquellen und Ereignisabhörer **2**
2.1 Ereignisse 2
2.2 Ereignisquellen 3
2.3 Ereignisabhörer 3
2.4 Ereignisbehandlung 3

3 Beispiel **4**
3.1 Beschreibung des Beispielprogramms 5

4 Innere Klassen **6**

5 Adapterklassen **6**

6 Anonyme Klassen **7**

A Beispielprogramm **8**

Vorwort

Es wird immer behauptet, Java sei eine einfache Programmiersprache. Das stimmt, wenn man Java mit C/C++ vergleicht. Es gibt in Java z.B. keine Pointer, die einem in C/C++ das Leben *wirklich* schwer machen können.

Was Java schwierig macht, ist eigentlich nur der Riesenumfang der Klassenbibliothek, die so groß ist, dass man sie als nicht Profi-Entwickler wahrscheinlich nie ganz kennenlernen kann.

Dieser kleine Text hier beschäftigt sich mit der Erzeugung und Abfrage von Ereignissen (Events), mit der der Anwender eine grafische Oberfläche bedient.

Natürlich kann man sich auf den Standpunkt stellen und sagen, alles was im Folgenden kommt möchte ich im Detail nicht wissen, die für die Verarbeitung von Events nötigen Methoden erzeugt meine Entwicklungsumgebung automatisch.

Was machen Sie aber, wenn Sie beim Editieren Ihres Quelltextes versehentlich einige Zeilen dieses automatisch erzeugten Codes löschen? Wie wollen Sie das wieder reparieren, wenn Sie nicht selbst ein wenig Bescheid über Events wissen?

Und da ereignisgesteuerte Programmierung der Schlüssel jeder grafischen Benutzeroberfläche ist, muss man ein paar Grundkenntnisse zu Events einfach haben.

1 Kommandozeile oder grafische Oberfläche

Softwareanwendungen kann man grob in zwei grosse Gruppen einteilen:

1. **Kommandozeilen-Anwendungen**: hier übergibt der Anwender beim Programmaufruf mit der *Kommandozeile* eine Reihe von Parametern. Die Anwendung läuft daraufhin ab und liefert schliesslich ein Ergebnis. Eventuell sind unterwegs noch ein paar Eingaben zu machen, wichtiges Kennzeichen von Kommandozeilen-Anwendungen ist jedoch, dass der Ablauf durch die Programmstruktur *fest* vorgegeben ist. Derartige Programme lassen sich auch in Java leicht schreiben, die main-Methode stellt dazu ein String-Array zur Verfügung, das die Kommandozeilenparameter aufnimmt.

2. **Programme mit grafischer Benutzerschnittstelle**: diese Programme werden vom Anwender durch Tastatureingaben oder Mausklicks *während* der Programmlaufzeit gesteuert. Der Programmierer einer solchen Anwendung hat nun *keinerlei* Einfluss auf die Reihenfolge der Anwender-Aktionen. Das bedeutet, die Anwendung muss jederzeit auf eine solche Aktion reagieren können: das Programm wartet in einer Endlosschleife auf diese Aktionen und wenn eine eintritt, wird eine entsprechende Methode aufgerufen. Die Endlosschleife wartet also, bis irgendein *Ereignis* eintritt. Man nennt dies *Ereignisbasierte Programmierung*.

2 Ereignisse, Ereignisquellen und Ereignisabhörer

2.1 Ereignisse

Aus der Sicht des Anwenders ist klar, was ein Ereignis ist: z.B. ein Mausklick auf eine Schaltfläche (Button). Wie wird so ein Schaltflächen-Klick aber innerhalb des Programms dargestellt? Java ist eine echte objektorientierte Sprache, was liegt also näher, als ein Event innerhalb des Programms als Objekt darzustellen:

Ein Event ist ein Objekt

Um ein Objekt erzeugen zu können, muss natürlich eine entsprechende Klasse existieren. Diese Klassen für unterschiedliche Ereignisse sind bereits vordefiniert und werden dem jeweiligen Programm mit dem Paket `java.awt.event` zur Verfügung gestellt.

Ein Beispiel: die Klasse java.awt.event.MouseEvent. Objekte dieser Klasse stellen Eigenschaften und Methoden zur Verfügung, die Details über den Zustand der Maus liefern. Z.B. die Methoden `getX()` und `getY()` liefern die aktuellen Mauskoordinaten.

Wenn nun die Maus bewegt, gezogen (=bewegt mit gedrückter Taste) oder eine Maustaste gedrückt oder losgelassen wird, wird ein `MouseEvent`-Objekt erzeugt. Wer das Objekt erzeugt erfahren Sie im nächsten Kapitel.

2.2 Ereignisquellen

Eine Ereignisquelle ist, wie könnte es auch anders sein, ein Objekt. Soweit nichts Neues. Aber dieses Objekt erzeugt *automatisch* Event-Objekte, wenn ein entsprechendes Ereignis auftritt.

Ein Beispiel für Maus-Ereignisquellen sind Objekte aller GUI-Klassen von Java (also Unterklassen der Klasse `java.awt.Component`). Objekte dieser Klassen erzeugen also Objekte der Klasse `MouseEvent`, wenn irgendeine Benutzeraktion mit der Maus auftritt.

Mit der Methode `getSource()`, die jedes Event-Objekt besitzt, kann man herausfinden, welche Grafikkomponente das Ereignisobjekt erzeugt hat, wer also die Quelle des Event-Objekts ist.

2.3 Ereignisabhörer

Die Ereignisabhörer (EventListener) sind selbstverständlich auch Objekte. Wir sehen: alles ist ein Objekt (in Java). Doch das Besondere an diesen Objekten ist, dass sie *informiert werden möchten*, wenn ein spezielles Ereignis auftritt. Das funktioniert so:

> Jede Ereignis*quelle* hat eine Liste, auf der alle `EventListener`-Objekte stehen, die informiert werden sollen, wenn ein Ereignis auftritt.

Um diese Liste zu verwalten hat jede Ereignisquelle Methoden um `EventListeners` in die Liste einzutragen oder zu entfernen. (add - remove - Methoden). Das Eintragen eines Listeners in diese Liste nennt man *registrieren*. Die add- remove-Methoden erwarten als Parameter ein `EventListener`-Objekt.

Noch eine wichtige Besonderheit: die Ereignisabhörer sind - wie alles andere in Java auch - Objekte. Allerdings gibt es keine fertigen `EventLister` - Klassen, sondern nur *Interfaces*. Das heisst, alle `EventListener` wie z.B. `MouseListener` oder `MouseMotionListener` sind "Klassen", die nur abstrakte, also leere Methoden (nur den Methodenkopf, ohne Rumpf) enthalten. Diese leeren Methoden muss man als Programmierer erst innerhalb einer eigenen Klasse mit echten Methoden überschreiben - man sagt implementieren - bevor man `EventListener`-Objekte bilden kann.

Warum aber liefert Sun mit Java `EventListener`-Interfaces und nicht gleich fertige Klassen mit und zwingt damit die Programmierer *alle* Methoden eines Interfaces zu implementieren?

Die Antwort ist einfach: es ist genau diese individuelle Implementierung, die den tatsächlichen Java-Code enthält, der als Reaktion auf ein Event ausgeführt werden soll!

2.4 Ereignisbehandlung

Wie wird nun ein registrierter Ereignisabhörer über ein bestimmtes Ereignis informiert?

Die Ereignisquelle übergibt einfach das Ereignisobjekt an die *entsprechende* Methode eines registrierten Abhörers. Welche Methode das Ereignisobjekt übergeben bekommt, hängt von der Art des Ereignisses ab.

Das `MouseListener`-Interface stellt zum Beispiel die 5 Methoden:

1. `mouseClicked(MouseEvent e)`
2. `mouseEntered(MouseEvent e)`
3. `mouseExited(MouseEvent e)`
4. `mousePressed(MouseEvent e)`
5. `mouseReleased(MouseEvent e)`

zur Verfügung. Zwei weitere Methoden,

6. `mouseDragged(MouseEvent e)`
7. `mouseMoved(MouseEvent e)`

werden von `MouseMotionListener`-Interface definiert.
Schaut man sich nun die Dokumentation der `MouseEvent`-Klasse an fällt auf, dass `MouseEvent`-Objekte die folgenden 7 Konstanten besitzen:

1. `public static final int MOUSE_CLICKED`
2. `public static final int MOUSE_DRAGGED`
3. `public static final int MOUSE_ENTERED`
4. `public static final int MOUSE_EXITED`
5. `public static final int MOUSE_MOVED`
6. `public static final int MOUSE_PRESSED`
7. `public static final int MOUSE_RELEASED`

Zu jeder dieser Konstanten gibt es also genau eine entsprechende Methode im `MouseListener` bzw. `MouseMotionListener` Interface.

Wird nun die Maus bewegt oder gezogen, erzeugt die Ereignisquelle (z.B. eine Schaltfläche, Button), ein `MouseEvent`-Objekt. Dieses wird von der Schaltfläche an die entsprechende Methode des registrierten `MouseMotionListener`-Objekts übergeben. Bedient man die (OSX) bzw. eine der Maustasten (X11), bekommt die entsprechende Methode des `MouseListener`-Objekts das `MouseEvent`-Objekt übergeben.

Da Sie, als Entwickler diese Methoden selbst geschrieben haben (schreiben mussten!), können Sie auf diese Weise eine beliebige Funktionalität mit einem MouseEvent verbinden.

3 Beispiel

Das im Anhang A enthaltene Beispiel demonstriert noch einmal das oben Beschriebene. Der Programmcode ist erhältlich bei
`http://examples.oreilly.com/9780596006204/` oder bei
`http://www.davidflanagan.com/javaexamples2`.
Leider ist David Flanagans Seite im Moment nich in Betrieb. Der Download von O'Reilly funktioniert. Das zugehörige Buch von David Flanagan, Java Examples in a Nutshell, 2. Auflage, gibt´s ebenfalls bei O´Reilly.

3.1 Beschreibung des Beispielprogramms

Hier ein paar Erläuterungen zum Quellcode. Da wäre zunächst die Klassendeklaration:

```
public class ScribblePanel extends JPanel
    implements MouseListener,
               MouseMotionListener
   {...}
```

Was sagen uns diese beiden Zeilen?

Der Klassenname ist `ScribblePanel` und `ScribblePanel` erbt von der Klasse `JPanel`.

`JPanel` kann man sich einfach als eine rechteckige Zeichenfläche vorstellen, auf der man zeichnen und die grafische Komponenten enthalten kann.

Ausserdem implementiert unsere `ScribblePanel`-Klasse die beiden Schnittstellen `MouseListener` und `MouseMotionListener`. Abbildung 1 zeigt diese Vererbungsstruktur.

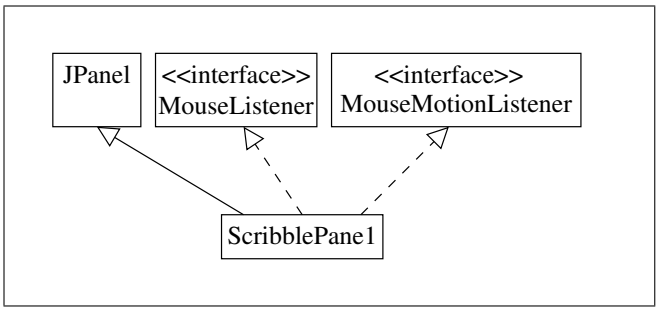

Abbildung 1: Vererbung und Implementieren von Interfaces

Implementiert man eine Schnittstelle, muss man *alle* ihre Methoden überschreiben. Geht man durch den weiteren Programmcode, wird man tatsächlich alle 7 bereits oben besprochenen Methoden dieser beiden Interfaces vorfinden. Dass einige davon einen leeren Methodenkörper haben heisst einfach nur, dass wir die zugehörigen MouseEvents nicht abfangen, man muss diese Methoden dennoch implementieren!

In diesem Beispielprogramm wird lediglich das Drücken der Maustaste (`mousePressed`) und Ziehen mit gedrückter Taste (`mouseDragged`) mit einer Methode hinterlegt.

Weiter im Text.

Das Beispiel ist eine Klasse, die keine main-Methode enthält. Um es zu testen, muss man eine weitere Klasse schreiben, die ein Fenster auf dem Bildschirm öffnet, ein Objekt unserer `ScribblePanel`-Klasse bildet und dieses Objekt in das Fenster einfügt und sichtbar macht.

Was hat es nun mit dem ScribblePanel-Objekt auf sich? Denken wir wieder an Abb. 1. Also gelten folgende Aussagen:

- das `ScribblePanel`-Objekt ist ein `JPanel`-Objekt (Vererbung)
- es ist aber auch ein `MouseListener`-Objekt

- und es ist gleichzeitig ein `MouseMotionListener`-Objekt

Wenn das gilt, gelten aber auch folgende Aussagen:

- das `ScribblePanel`-Objekt ist eine Maus-Ereignisquelle, da `JPanel` eine GUI-Komponente ist und alle GUI-Komponenten `MouseEvents` erzeugen können.
- gleichzeitig ist das `ScribblePanel`-Objekt aber auch ein Listener für Mause-Ereignisse, eben ein `MouseListener`-Objekt

Das erklärt nun die beiden ersten Befehle des Konstruktors:

```
this.addMouseListener(this);
this.addMouseMotionListener(this);
```

Da ist also die bereits erwähnte `add`-Methode um einen Listener zu registrieren. Parameter der `add`-Methode muss ein `EventListener`-Objekt sein:

Das Wörtchen `this` steht für ein beliebiges, von dieser Klasse instanziertes Objekt, eben ein `ScribblePanel`-Objekt.

Als Ereignisquelle trägt dieses Objekt zwei Listener-Objekte in seine Liste der registrierten Abhörer ein. Und diese beiden Listener-Objekte sind wiederum das Objekt selbst!

Oder einfacher: das Objekt registriert sich selbst als Ereignisabhörer, bei einem Maus-Ereignis benachrichtigt sich das Objekt folglich selbst.

Der Rest des Programmcodes sei an dieser Stelle nicht besprochen, da er durch die von David Flanagan eingefügten Kommentare für sich selbst spricht.

4 Innere Klassen

Anstelle dieser Pseudo-Mehrfachvererbung, bei der ein Objekt merkwürdigerweise vom Typ `JPanel` und gleichzeitig vom Typ `MouseListener` ist, hätten wir doch ganz einfach zwei neue Klassen erzeugen können, die jeweils nur das `MouseListener`- und das `MouseMotionListener`-Interface implementieren und hätten von diesen Klassen jeweils ein Objekt gebildet und dieses an die add*Xxx*Listener-Methoden übergeben.

Der Nachteil dieser Vorgehensweise wäre aber, dass dann die Methoden, die zu den jeweiligen Mausereignissen gehören, in einer anderen Datei stehen. Das macht die Lesbarkeit des Programms nicht eben besser.

Und da hat sich Sun die sogenannten *Inneren Klassen* einfallen lassen: eine innere Klasse ist eine Klasse, die *innerhalb* einer anderen Klasse deklariert wird. Sie hat Zugriff auf alle Eigenschaften und Methoden der äusseren Klasse.

Statt die beiden oben beschriebenen Listener-Klassen auszulagern, könnten wir diese als innere Klassen schreiben.

5 Adapterklassen

Mit den inneren Klassen ist noch nicht viel gewonnen. Die inneren Klassen implementieren eine Listener-Schnittstelle, folglich müssen wir *alle* abstrakten Methoden der

Schnittstelle schreiben (implementieren).

Einigen Programmierern war wohl das Tippen von so vielen leeren Methoden lästig und da hat Sun sich nochmals was Neues einfallen lassen: die Adapterklassen.

Eine Adapterklasse ist eine Klasse, die eine Schnittstelle implementiert.

Hier als Beispiel die `MouseAdapter`-Klasse:

```
public abstract class MouseAdapter
   implements MouseListener{

   public void mouseClicked(MouseEvent e){};
   public void mouseEntered(MouseEvent e){};
   public void mouseExited(MouseEvent e){};
   public void mousePressed(MouseEvent e){};
   public void mouseReleased(MouseEvent e){};
}
```

vergleicht man dies mit der ursprünglichen Deklaration des `MouseListener`-Interfaces:

```
public abstract interface MouseListener
   extends EventListener{

   public abstract
      void mouseClicked(MouseEvent e);
   public abstract
      void mouseEntered(MouseEvent e);
   public abstract
      void mouseExited(MouseEvent e);
   public abstract
      void mousePressed(MouseEvent e);
   public abstract
      void mouseReleased(MouseEvent e);
}
```

könnte man beinahe den kleinen aber entscheidenden Unterschied übersehen: bei der MouseAdapter-Klasse sind die Methoden nun nicht mehr abstrakt! Und sie müssen deshalb auch *nicht* alle überschrieben werden. Wie funktioniert das?

Man erzeugt einfach eine neue Klasse, die von der Adapterklasse erbt und *überschreibt* nur die Methoden, für die man sich interessiert.

All diesen Aufwand mit den Adapterklassen und den ListenerInterfaces musste man nur treiben, um ein `EventListener`-Objekt (in unseren Beispielen ein `MouseListener`- / `MouseMotionListener`-Objekt) zu erzeugen. Und dieses Objekt wird zu allem Überfluss sofort an die add-Methoden der GUI-Komponente übergeben. Niemand interessiert sich für den Namen dieses Objekts und auch nicht für seine Klasse. Und weil das so ist, hat Sun sich nochmal was einfallen lassen:

6 Anonyme Klassen

Für anonyme Klassen gilt:

- sie werden innerhalb einer Klasse deklariert, sind also eine Variante der inneren Klassen

- eine anonyme Klasse ist Unterklasse einer anderen Klasse oder implementiert ein Interface
- anonyme Klassen werden innerhalb einer einzigen Anweisung deklariert und gleichzeitig wird ein Objekt von ihnen gebildet
- nach dem Schlüsselwort `new` steht der Standardkonstruktor der Oberklasse oder der Name der Schnittstelle

Hier ein Code-Beispiel, wie unser `MouseMotionListener`-Objekt mit Hilfe einer anonymen Klasse und einer Adapterklasse gebildet werden kann:

```
addMouseListener(
  new MouseAdapter(){
    public void mousePressed(MouseEvent e){
      moveto(e.getX(), e.getY());
    }
  }
);
```

Dieses Beispiel besteht tatsächlich nur aus der Anweisung: `addMouseListener(MouseListener);`. Und der Parameter dieser Methode, das `MouseListener`-Objekt wird als namenloses Objekt einer namenlosen Klasse gebildet, die von `MouseAdapter` erbt. In dieser anonymen Unterklasse wird nur eine Methode der Adapterklasse überschrieben. Das ist in diesem Fall erlaubt, da die Methoden der Adapterklasse nicht mehr abstrakt sind.

In den meisten Javaprogrammen werden Sie derartige Code-Konstruktionen finden. Wegen der besseren Lesbarkeit ziehe ich persönlich jedoch den zuerstbeschriebenen Ansatz vor. Auch wenn es etwas mehr Tipparbeit bedeutet.

A Beispielprogramm

Abbildung 2: Klassendefinition und Objekterzeugung in Java

```
/*
 * Copyright (c) 2000 David Flanagan.  All rights reserved.
 * This code is from the book Java Examples in a Nutshell, 2nd Edition.
 * It is provided AS-IS, WITHOUT ANY WARRANTY either expressed or implied.
 * You may study, use, and modify it for any non-commercial purpose.
 * You may distribute it non-commercially as long as you retain this notice.
 * For a commercial use license, or to purchase the book (recommended),
 * visit http://www.davidflanagan.com/javaexamples2.
 */
package com.davidflanagan.examples.gui;
import javax.swing.*;       // For JPanel component
import java.awt.*;          // For Graphics object
import java.awt.event.*;    // For Event and Listener objects

/**
 * A simple JPanel subclass that uses event listeners to allow the user
 * to scribble with the mouse.  Note that scribbles are not saved or redrawn.
 **/
public class ScribblePane1 extends JPanel
    implements MouseListener, MouseMotionListener {
    protected int last_x, last_y;  // Previous mouse coordinates

    public ScribblePane1() {
        // This component registers itself as an event listener for
        // mouse events and mouse motion events.
        this.addMouseListener(this);
        this.addMouseMotionListener(this);

        // Give the component a preferred size
        setPreferredSize(new Dimension(450,200));
    }

    // A method from the MouseListener interface.  Invoked when the
    // user presses a mouse button.
    public void mousePressed(MouseEvent e) {
        last_x = e.getX();  // remember the coordinates of the click
        last_y = e.getY();
    }

    // A method from the MouseMotionListener interface.  Invoked when the
    // user drags the mouse with a button pressed.
    public void mouseDragged(MouseEvent e) {
        int x = e.getX();    // Get the current mouse position
        int y = e.getY();
        // Draw a line from the saved coordinates to the current position
        this.getGraphics().drawLine(last_x, last_y, x, y);
        last_x = x;          // Remember the current position
        last_y = y;
    }

    // The other, unused methods of the MouseListener interface.
    public void mouseReleased(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

    // The other, unused, method of the MouseMotionListener interface.
    public void mouseMoved(MouseEvent e) {}
}
```