Aber es passiert nichts, wenn ich draufklicke ...

Das stimmt nicht ganz. Wenn Sie den Button drücken, zeigt er dieses typische »runtergedrückte« Aussehen (was vom »Look-and-Feel« auf der jeweiligen Plattform abhängt; aber *irgendwie* lässt der Button es auf jeden Fall erkennen, dass er gedrückt ist).

Die eigentliche Frage lautet: »Wie bringe ich den Button dazu, etwas Bestimmtes zu tun, wenn der Benutzer ihn anklickt?«

Zwei Dinge brauchen wir dazu:

1. Eine **Methode**, die aufgerufen wird, wenn der Benutzer klickt (damit das passiert, was Sie als Resultat des Button-Klicks vorgesehen haben).

2. Eine Möglichkeit, um zu **erfahren**, wann die Methode aufgerufen werden soll. Mit anderen Worten: eine Möglichkeit, um zu erfahren, wann der Benutzer den Button anklickt!

Wenn der Benutzer klickt, wollen wir das erfahren.

Wir interessieren uns für das »Benutzer-führt-eine-Aktion-auf-einem-Button-aus«-Ereignis.

Es gibt keine Dummen Fragen

F: Sieht ein Button unter Windows wie ein Windows-Button aus?

A: Wenn Sie das wollen, ja. Sie können unter ein paar »Look-and-Feel«-Klassen in der Kernbibliothek wählen, die steuern, wie eine Oberfläche aussieht. In den meisten Fällen können Sie zwischen mindestens zwei verschiedenen Möglichkeiten wählen: dem Java-Standard-Look-and-Feel (auch als ***Metal*** bezeichnet) und Ihrem plattformtypischen Look-and-Feel. Die Mac OS X-Bildschirmfotos in diesem Buch zeigen entweder das OS X-***Aqua***-Look-and-Feel oder das ***Metal***-Look-and-Feel.

F: Kann ich ein Programm *immer* wie Aqua aussehen lassen? Auch wenn es unter Windows läuft?

A: Nein. Es stehen nicht für jede Plattform alle Look-and-Feels zur Verfügung. Wenn Sie auf Nummer sicher gehen wollen, können Sie das Look-and-Feel explizit auf Metal setzen, dann wissen Sie genau, was Sie bekommen – egal, wo die Anwendung läuft. Oder Sie geben gar kein Look-and-Feel an und akzeptieren das jeweils voreingestellte.

F: Ich habe gehört, Swing wäre eine lahme Ente und würde von niemandem verwendet.

A: Das galt früher einmal, heute jedoch nicht mehr. Bei leistungsschwachen Rechnern bekommen Sie es vielleicht noch zu spüren. Aber auf neueren Computern und seit der Java-Version 1.3 und höher werden Sie den Unterschied zwischen einer Swing-Oberfläche und einer nativen Oberfläche vielleicht nicht einmal bemerken. Swing wird heute sehr verbreitet eingesetzt, und zwar in den unterschiedlichsten Anwendungen.

Wie man an ein Benutzerereignis kommt

Stellen Sie sich vor, Sie möchten den Text *klick mich* auf dem Button in *ich wurde geklickt!* ändern, wenn der Benutzer den Button anklickt. Dazu können wir erst einmal eine Methode schreiben, die den Text des Buttons verändert (ein rascher Blick in die API zeigt Ihnen die entsprechende Methode von Button):

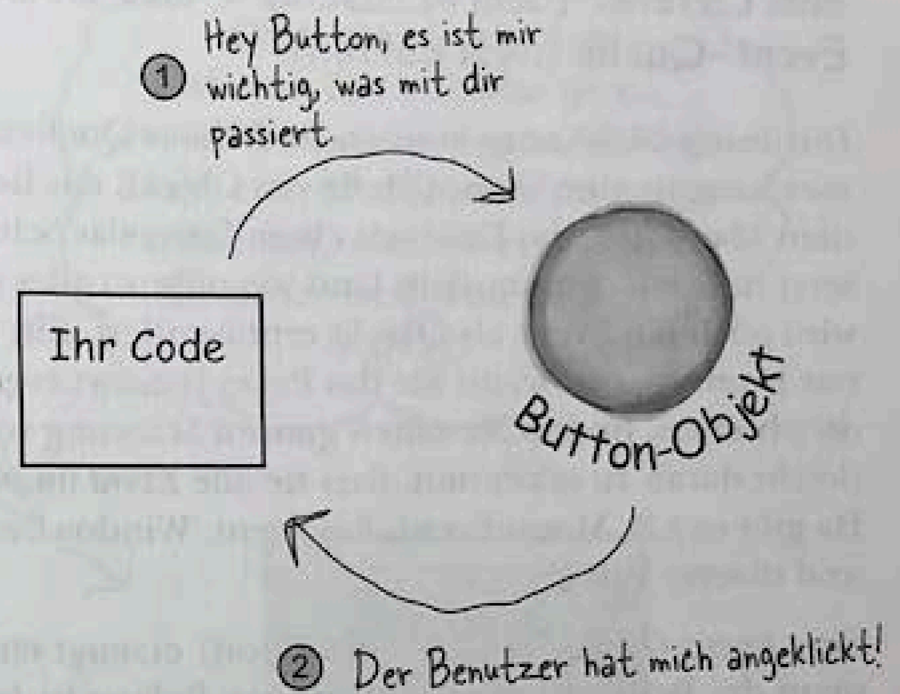
```
public void ändern() {
  button.setText("ich wurde geklickt!");
}
```

Aber was *jetzt?* Wie *erfahren* wir, wann diese Methode ausgeführt werden soll? ***Wie erfahren wir, wann der Button angeklickt wird?***

In Java bezeichnet man das Verfahren, wie man an ein Benutzerereignis – ein »Event« – gelangt und wie man es behandelt, als *Event-Handling*. Es gibt viele verschiedene Arten von Events in Java, die meisten davon hängen allerdings mit GUI-Benutzeraktionen zusammen. Wenn der Benutzer auf einen Button klickt, ist das ein Event. Ein Event, das besagt: »Der Benutzer will, dass die Aktion für diesen Button stattfindet.« Wenn es ein »Tempo verlangsamen«-Button ist, möchte der Benutzer, dass die Tempo-verlangsamen-Aktion ausgeführt wird. Wenn es ein Senden-Button ist oder ein Chat-Client, will der Benutzer, dass die Sende-meine-Nachricht-Aktion ausgeführt wird. Das einfachste Event ist also, wenn der Benutzer auf den Button klickt und damit kundtut: »Ich möchte, dass eine Aktion stattfindet.«

Bei Buttons interessieren Sie sich normalerweise nicht für irgendwelche Zwischenereignisse wie z.B. das Herunterdrücken oder das Wiederloslassen. Was Sie dem Button sagen möchten, ist Folgendes: »Es ist mir gleich, wie der Benutzer mit dem Button herumspielt, wie lange er den Mauszeiger darüber stehen lässt, wie oft er es sich anders überlegt und den Mauszeiger wieder herunterbewegt, bevor er loslässt, usw. ***Sag es mir einfach, wenn der Benutzer es wirklich ernst meint!*** Mit anderen Worten, ruf mich nur dann auf, wenn der Benutzer in einer Weise klickt, die deutlich macht: Er will, dass der verdammte Button das tut, was auf ihm geschrieben steht!«

Erst einmal muss der Button wissen, dass er uns nicht egal ist.



Zweitens braucht der Button eine Möglichkeit, uns zu benachrichtigen, wenn ein Button-geklickt-Event eintritt.

1) Wie könnten Sie einem Button-Objekt mitteilen, dass Ihnen seine Events wichtig sind? Dass Sie interessiert darauf lauschen?

2) Wie wird der Button Sie benachrichtigen? Nehmen Sie an, es gibt keine Möglichkeit, wie Sie dem Button den Namen Ihrer ganz speziellen Methode (ändern()) mitteilen können. Wodurch können wir dem Button sonst noch klarmachen, dass wir eine spezielle Methode haben, die er aufrufen kann, wenn das Event stattfindet?

Wenn Sie sich für die Ereignisse des Buttons interessieren, implementieren Sie ein Interface, das ihm mitteilt: »Ich *lausche* auf deine Events.«

Ein **Listener-Interface** ist die Brücke zwischen dem **Listener** (dem »Lauscher«, also Ihnen) und der **Event-Quelle** (dem Button).

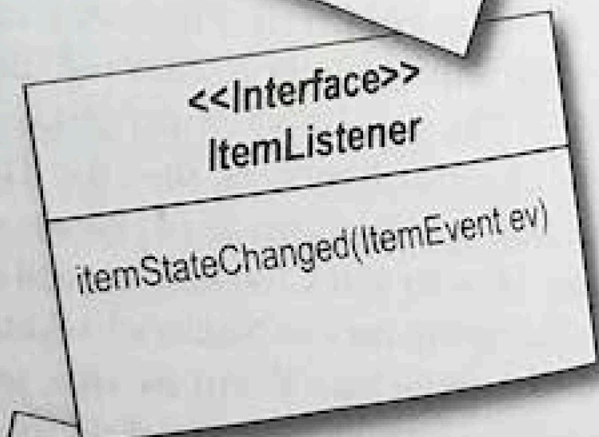
Die Swing-GUI-Komponenten sind Event-Quellen. Im Java-Zusammenhang ist eine Event-Quelle ein Objekt, das Benutzeraktionen (den Mausklick, das Drücken einer Taste, das Schließen eines Fensters) in Events umwandelt. Und wie nahezu alles andere in Java wird auch ein Event als Objekt repräsentiert. Ein Objekt irgendeiner Event-Klasse. Wenn Sie das Paket java.awt.event in der API durchsehen, finden Sie einen ganzen Schwung von Event-Klassen (leicht daran zu erkennen, dass sie alle *Event* im Namen tragen). Da gibt es z.B. MouseEvent, KeyEvent, WindowEvent, ActionEvent und diverse andere.

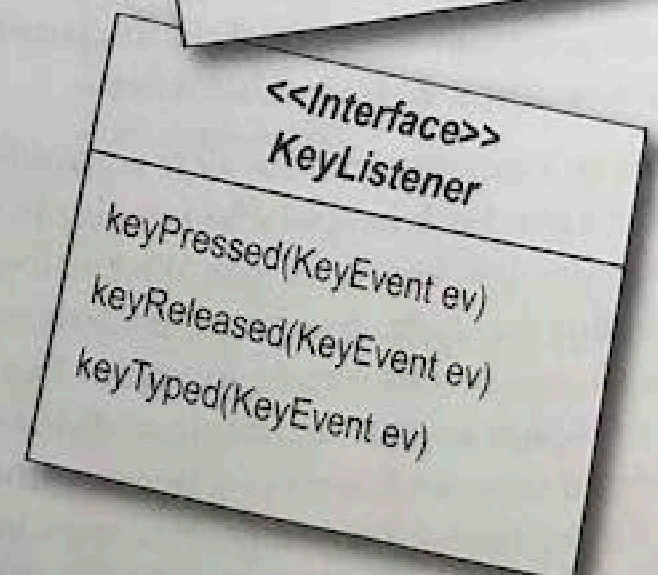
Eine Event-*Quelle* (wie z.B. ein Button) erzeugt ein ***Event-Objekt***, wenn der Benutzer etwas tut, was von Belang ist (wie z.B. den Button *anklicken*). Der größte Teil Ihres selbst geschriebenen Codes (und der gesamte Code in diesem Buch) wird keine Events *erzeugen*, sondern sie *empfangen*. Anders ausgedrückt: Den größten Teil Ihrer Zeit werden Sie als Event-*Listener* verbringen, nicht als Event-*Quelle*.

Zu jedem Event-Typ gehört ein passendes Listener-Interface. Wenn Sie MouseEvents haben wollen, implementieren Sie das MouseListener-Interface. Sie wollen WindowEvents? Implementieren Sie WindowListener. Sie verstehen das Prinzip? Und erinnern Sie sich an die Regeln zu den Interfaces – um ein Interface zu implementieren, *deklarieren* Sie, dass Sie es implementieren (class Hund implements Haustier), was bedeutet, dass Sie selbst eine *Implementierungsmethode* für jede der Methoden aus dem Interface schreiben müssen.

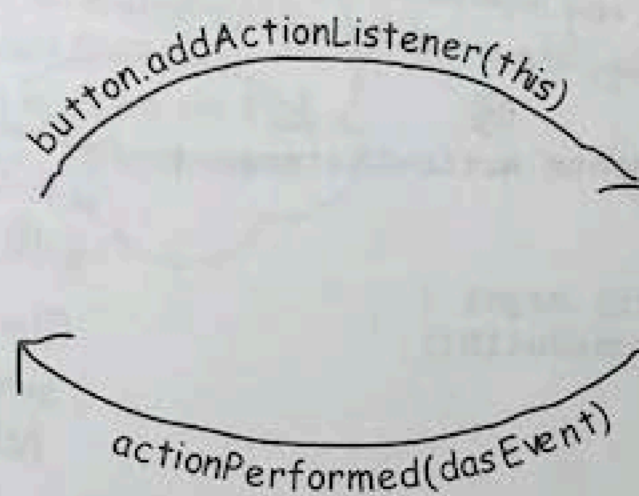
Manche Interfaces enthalten mehr als eine Methode, weil das Event selbst in unterschiedlichen Geschmacksrichtungen zu haben ist. Wenn Sie beispielsweise MouseListener implementieren, können Sie mousePressed-, mouseReleased-, mouseMoved-Events usw. erhalten. Zu jedem dieser Mausereignisse gehört eine eigene Methode im Interface, auch wenn sie alle ein MouseEvent entgegennehmen. Wenn Sie MouseListener implementieren, wird die Methode mousePressed() aufgerufen, wenn der Benutzer – richtig geraten! – die Maustaste drückt. Und wenn der Benutzer loslässt, wird die Methode mouseReleased() aufgerufen. Bei Mausereignissen gibt es also nur ein einziges Event-*Objekt* – MouseEvent –, aber mehrere verschiedene Event-*Methoden*, die die unterschiedlichen *Arten* von Mausereignissen repräsentieren.

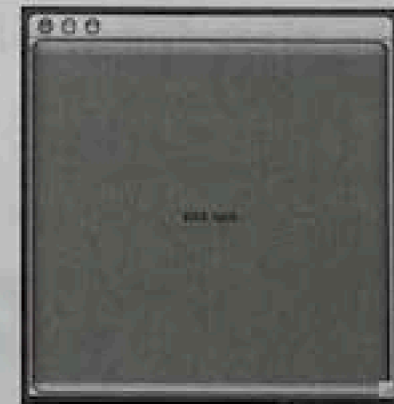
Wenn Sie ein Listener-Interface implementieren, geben Sie damit dem Button eine Möglichkeit, Sie zu benachrichtigen. In dem Interface wird die Methode für die Benachrichtigung deklariert.





Wie sich der Listener und die Quelle verständigen:





Der Listener

Wenn Ihre Klasse etwas über die ActionEvents eines Buttons erfahren will, implementieren Sie das Interface ActionListener. Der Button muss wissen, dass Sie interessiert sind, daher registrieren Sie sich bei ihm, indem Sie seine Methode addActionListener(this) aufrufen und ihr eine ActionListener-Referenz übergeben (in diesem Fall sind *Sie* der ActionListener, also übergeben Sie *this*). Der Button braucht eine Möglichkeit, Sie zu benachrichtigen, wenn das Event eintritt, daher ruft er die Methode aus dem Listener-Interface auf. Als ActionListener *müssen* Sie die einzige Methode aus diesem Interface – actionPerformed() – implementieren. Das garantiert der Compiler.

Die Event-Quelle

Ein Button ist eine Quelle von Action-Events und muss deshalb wissen, welche Objekte als Listener an ihm interessiert sind. Der Button hat eine addActionListener()-Methode, die es den betreffenden Objekten (Listenern) ermöglicht, ihm *mitzuteilen*, dass sie interessiert sind.

Wird die Methode addActionListener() des Buttons ausgeführt (weil ein potenzieller Listener sie aufgerufen hat), nimmt der Button den Parameter (eine Referenz auf das Listener-Objekt) und speichert ihn in einer Liste. Wenn der Benutzer den Button anklickt, wird das Event vom Button abgesetzt (»abgefeuert«), indem dieser auf jedem Listener in seiner Liste die Methode actionPerformed() aufruft.

Wie man an das ActionEvent eines Buttons kommt

1. Implementieren Sie das Interface ActionListener.
2. Registrieren Sie sich beim Button (teilen Sie ihm mit, dass Sie auf Events lauschen wollen).
3. Definieren Sie die Event-Handling-Methode (implementieren Sie die Methode actionPerformed() aus dem Interface ActionListener).

```
import javax.swing.*;
import java.awt.event.*;

public class EinfacheGui1B implements ActionListener {
  JButton button;

  public static void main (String[] args) {
    EinfacheGui1B gui = new EinfacheGui1B();
    gui.los();
  }

  public void los() {
    JFrame frame = new JFrame();
    button = new JButton("klick mich");

    button.addActionListener(this);

    frame.getContentPane().add(button);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300,300);
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    button.setText("Ich wurde geklickt!");
  }
}
```

Eine neue Importanweisung für das Paket, zu dem ActionListener und ActionEvent gehören.

(1) Implementieren Sie das Interface. Das bedeutet: »Eine Instanz von EinfacheGui1B IST-EIN ActionListener.«

(Der Button schickt Events nur an registrierte Implementierer von ActionListener.)

(2) Registrieren Sie sich beim Button. Damit sagen Sie zum Button: »Nimm mich in deine Listener-Liste auf.« Das von Ihnen übergebene Argument MUSS ein Objekt einer Klasse sein, die ActionListener implementiert!!

(3) Implementieren Sie die Methode actionPerformed() aus dem Interface ActionListener. Dies ist die eigentliche Event-Handling-Methode!

Der Button ruft diese Methode auf und lässt Sie damit wissen, dass ein Event eingetreten ist. Als Argument sendet er Ihnen ein ActionEvent-Objekt, aber das benötigen wir nicht. Es genügt uns zu wissen, dass das Event eingetreten ist.

Listener, Quellen und Events

Im größten Teil Ihrer Java-Karriere in diesem Universum werden nicht *Sie* die *Quelle* von Events sein.

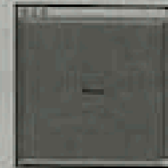
(Wie sehr Sie sich selbst auch als das Zentrum Ihres sozialen Universums sehen mögen.)

Gewöhnen Sie sich daran. ***Ihr Job ist es, als Listener gut hinzuhören.***

(Was Ihre sozialen Beziehungen in der Tat verbessern *kann*, wenn Sie sich ernsthaft Mühe geben.)

Meine Aufgabe als Listener ist es, das Interface zu **implementieren**, mich beim Button zu **registrieren** und mich um die Behandlung des Events zu **kümmern**.

Listener ERHÄLT das Event



Meine Aufgabe als Event-Quelle ist es, Registrierungen (von Listenern) **entgegenzunehmen**, Events vom Benutzer zu **bekommen** und die Event-Handling-Methode des Listeners **aufzurufen** (wenn der Benutzer mich anklickt).

Quelle SENDET das Event

He, was ist mit mir? Ich spiele doch auch mit! Als Event-Objekt bin ich das **Argument** der Event-Benachrichtigungsmethode (aus dem Interface), und mein Job ist es, dem Listener **Daten** über das Event zu **liefern**.

Event-Objekt ENTHÄLT DATEN über das Event

Es gibt keine Dummen Fragen

F: Warum kann ich keine Quelle von Events sein?

A: Doch, *können* Sie. Wir haben nur gesagt, dass Sie *fast immer* der Empfänger des Events sein werden und nicht sein Ursprung (zumindest *am Beginn* Ihrer glanzvollen Java-Karriere). Die meisten Events, die Sie interessieren könnten, werden von Klassen der Java-API »abgefeuert«, und Sie müssen nichts weiter tun, als Listener für diese Events zu sein. Aber vielleicht entwerfen Sie ja auch ein Programm, bei dem Sie ein selbst definiertes Event brauchen – z.B. ein AktienMarktEvent, das ausgelöst wird, wenn Ihre Aktienmarktbeobachtungsanwendung etwas entdeckt, das sie als wichtig einstuft. In diesem Fall würden Sie das AktienBeobachter-Objekt zu einer Event-Quelle machen und das Gleiche tun, was auch ein Button (oder irgendeine andere Quelle) tut: ein Listener-Interface für Ihr selbst definiertes Event schreiben und eine Registrierungsmethode zur Verfügung stellen (addAktienListener()), und wenn diese von jemandem aufgerufen wird, wird der Aufrufer (ein Listener) in die Listener-Liste eingetragen. Wenn dann ein Aktienereignis eintritt, instantiieren Sie ein AktienEvent-Objekt (eine weitere selbst geschriebene Klasse) und senden es an die Listener auf Ihrer Liste, indem Sie deren Methode aktienVerändert(AktienEvent ev) aufrufen. Und vergessen Sie nicht, dass es zu jedem Event-Typ ein *passendes Listener-Interface* geben muss (d.h., Sie schreiben ein AktienListener-Interface mit einer Methode aktienVerändert()).

F: Die Bedeutung des Event-Objekts, das den Methoden zur Event-Benachrichtigung übergeben wird, ist mir noch nicht klar. Wenn jemand meine mousePressed()-Methode aufruft, brauche ich doch sonst keine weiteren Informationen, oder?

A: Die meiste Zeit und bei den meisten Entwürfen brauchen Sie das Event-Objekt nicht. Es ist nichts weiter als ein kleiner Datenträger, mit dem sich zusätzliche Informationen über das Ereignis übermitteln lassen. Aber manchmal möchten Sie vielleicht bestimmte Einzelheiten über das Event wissen. Wenn beispielsweise Ihre mousePressed()-Methode aufgerufen wird, wissen Sie, dass Ihre Maustaste heruntergedrückt wurde. Aber wenn Sie nun wissen möchten, wo genau diese gedrückt wurde? Anders ausgedrückt, wenn Sie wissen wollen, bei welchen x- und y-Bildschirmkoordinaten die Maustaste gedrückt wurde?

Oder: Manchmal möchten Sie den *gleichen* Listener bei *mehreren* Objekten registrieren. Ein Bildschirmtaschenrechner beispielsweise hat zehn Zahlentasten, und da diese alle das Gleiche tun, möchten Sie nicht für jede einzelne Taste einen separaten Listener machen. Stattdessen könnten Sie einen einzigen Listener bei allen zehn Tasten registrieren, und wenn ein Event bei Ihnen eingeht (weil Ihre Event-Benachrichtigungsmethode aufgerufen wird), können Sie eine Methode auf dem Event-Objekt aufrufen, um herauszufinden, *wer* denn nun die Event-Quelle war bzw. *welche Taste dieses Event geschickt hat.*

Spitzen Sie Ihren Bleistift

Jedes dieser Widgets (Benutzeroberflächenobjekte) ist die Quelle für ein oder mehrere Events. Ordnen Sie die Widgets den von Ihnen aufgerufenen Methoden zu. Manche der Widgets können als Quelle für mehr als ein Event fungieren, und manche der Events können von mehr als einem der Widgets erzeugt werden.

Widgets	Event-Methoden
Checkbox	windowClosing()
Textfeld	actionPerformed()
scrollbare Liste	itemStateChanged()
Button	mousePressed()
Dialogfeld	keyTyped()
Radio-Button	mouseExited()
Menüpunkt (engl. menu item)	focusGained()

Woher WEISS man, ob ein Objekt eine Event-Quelle ist?

Sehen Sie in der API nach.

OK. Und wonach suche ich?

Nach einer Methode, die mit »add« beginnt, mit »Listener« aufhört und ein Listener-Interface-Argument übergeben bekommt. Wenn Sie so etwas sehen:

addKeyListener(KeyListener k)

wissen Sie, dass die Klasse mit dieser Methode eine Quelle für KeyEvents ist. Es gibt eine Namenskonvention.