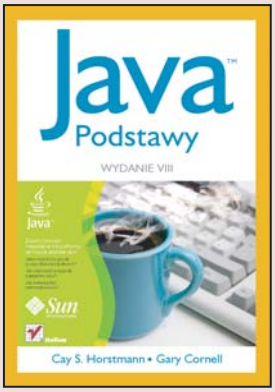
Java. Podstawy. Wydanie VIII



Autor: Cay S. Horstmann, Gary Cornell
Tłumaczenie: Łukasz Piwko
ISBN: 978-83-246-1478-3
Tytuł oryginału: Core Java(TM), Volume I–Fundamentals (8th Edition)
Format: 172x245, stron: 888

Zacznij tworzyć niezależne od platformy aplikacje jeszcze dziś!

- Jakie nowości kryją się w Java Standard Edition 6?
- Jak rozpocząć przygodę z językiem Java?
- Jak wykorzystać wielowątkowość?

Język programowania Java został stworzony i jest rozwijany przez firmę Sun Microsystems. Możliwość zastosowania go na różnych platformach została doceniona przez wielu programistów na świecie. Jednak nie jest to jedyna mocna strona Javy. Warto tu wskazać również jej silne ukierunkowanie na obiektowość, obsługę programowania rozproszonego, mechanizm automatycznego oczyszczania pamięci (ang. garbage collection). Dzięki swoim atutom, dobrej dokumentacji i licznych publikacjom Java jest dziś wiodącym rozwiązaniem na rynku języków programowania.

Książka „Java. Podstawy. Wydanie VIII" została zaktualizowana o wszystkie te elementy, które pojawiły się w wersji szóstej platformy Java Standard Edition. Tom pierwszy – „Podstawy" – zawiera wprowadzenie do języka programowania Java. Autorzy książki przedstawią tu założenia przyjęte przez firmę Sun przy tworzeniu tej platformy. Dowiesz się, jakie prawa rządzą programowaniem obiektowym oraz jak wykorzystać interfejsy i obsługę wyjątków. Dodatkowo będziesz mieć możliwość zapoznania się z elementami projektowania i tworzenia interfejsu użytkownika. W ostatnim rozdziale autorzy omówią wielowątkowość oraz sposób zastosowania tego typu rozwiązań w codziennej pracy programisty języka Java.

- Podstawy języka Java
  – Programowanie obiektowe – Interfejsy – Sposób użycia klas proxy oraz klas wewnętrznych – Projektowanie interfejsu użytkownika z wykorzystaniem biblioteki Swing – Obsługa wyjątków – Wykrywanie i rozwiązywanie problemów w kodzie – Wielowątkowość

Wykorzystaj siłę obiektów.
Programowanie obiektowe w języku Java ma przyszłość!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

8

Obsługa zdarzeń

W tym rozdziale:

- Podstawy obsługi zdarzeń
- Akcje
- Zdarzenia generowane przez mysz
- Hierarchia zdarzeń `AWT`

Obsługa zdarzeń ma fundamentalne znaczenie w programach z graficznym interfejsem użytkownika. Każdy, kto chce tworzyć interfejsy graficzne w Javie, musi opanować obsługę zdarzeń. Niniejszy rozdział opisuje model obsługi zdarzeń biblioteki `AWT`. Do opisywanych zagadnień należą przechwytywanie zdarzeń w komponentach interfejsu użytkownika i urządzeniach wejściowych, a także **akcje** (ang. *actions*), czyli bardziej strukturalna metoda przetwarzania zdarzeń.

Podstawy obsługi zdarzeń

Każdy system operacyjny posiadający graficzny interfejs użytkownika stale monitoruje zachodzące w nim zdarzenia, jak naciskanie klawiszy na klawiaturze czy kliknięcia przyciskiem myszy. Informacje o tych zdarzeniach są przesyłane do uruchomionych programów. Następnie każdy program podejmuje samodzielną decyzję, w jaki sposób, jeśli w ogóle, zareagować na te zdarzenia. W takich językach jak Visual Basic relacje pomiędzy zdarzeniami a kodem są oczywiste. Programista pisze kod obsługi każdego interesującego go zdarzenia i umieszcza go w tzw. **procedurze obsługi zdarzeń** (ang. *event procedure*). Na przykład z przyciskiem o nazwie `HelpButton` w języku Visual Basic może być skojarzona procedura obsługi zdarzeń o nazwie `HelpButton_Click`. Kod niniejszej procedury jest wykonywany w odpowiedzi na każde kliknięcie niniejszego przycisku. Każdy komponent GUI w języku Visual Basic reaguje na ustalony zestaw zdarzeń — nie można zmienić zdarzeń, na które reaguje dany komponent.

Natomiast programiści czystego języka C zajmujący się zdarzeniami muszą pisać procedury nieprzerwanie monitorujące kolejkę zdarzeń w celu sprawdzenia, jakie powiadomienia przesyła system operacyjny (z reguły do tego celu stosuje się pętlę zawierającą bardzo rozbudowaną instrukcję `switch`!). Technika ta jest oczywiście bardzo mało elegancka i sprawia wiele problemów podczas pisania kodu. Jej zaletą jest natomiast to, że nie ma żadnych ograniczeń dotyczących zdarzeń, na które można reagować, w przeciwieństwie do innych języków, np. Visual Basica, które wkładają bardzo dużo wysiłku w ukrywanie kolejki zdarzeń przed programistą.

W środowisku programistycznym Javy przyjęto podejście pośrednie pomiędzy językami Visual Basic a C, jeśli chodzi o oferowane możliwości, a co za tym idzie — także złożoność. Poruszając się w zakresie zdarzeń, które obsługuje biblioteka `AWT`, programista ma pełną kontrolę nad sposobem przesyłania zdarzeń ze **źródeł zdarzeń** (ang. *event sources*), np. przycisków lub pasków przewijania, do **słuchaczy zdarzeń** (ang. *event listener*). Na słuchacza zdarzeń można desygnować każdy **obiekt** — w praktyce wybiera się ten obiekt, który z łatwością może wykonać odpowiednie działania w odpowiedzi na zdarzenie. Ten delegacyjny model zdarzeń daje znacznie większe możliwości niż język Visual Basic, w którym słuchacz jest ustalony z góry.

Źródła zdarzeń dysponują metodami, w których można rejestrować słuchaczy zdarzeń. Kiedy ma miejsce określone zdarzenie, źródło wysyła powiadomienie o nim do wszystkich obiektów nasłuchujących, które zostały dla niego zarejestrowane.

Jak można się spodziewać, informacje o zdarzeniu w języku obiektowym, takim jak Java, są pakowane w **obiekcie zdarzeń** (ang. *event object*). W Javie wszystkie obiekty zdarzeń należą do klasy `java.util.EventObject`. Oczywiście istnieją też podklasy reprezentujące każdy typ zdarzenia, takie jak `ActionEvent` czy `WindowEvent`.

Różne źródła zdarzeń mogą dostarczać różnego rodzaju zdarzeń. Na przykład przycisk może wysyłać obiekty `ActionEvent`, podczas gdy okno wysyła obiekty `WindowEvent`.

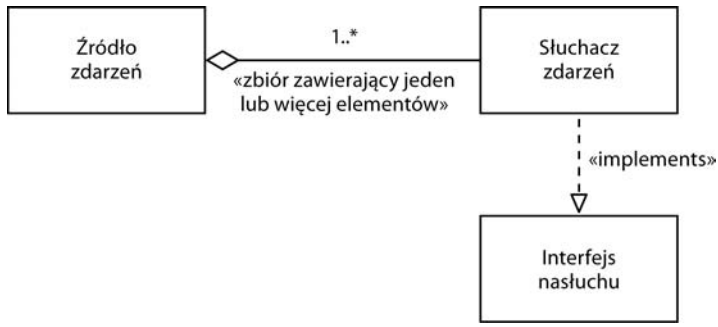
Podsumujmy, co już wiemy na temat obsługi zdarzeń w bibliotece `AWT`:

- Obiekt nasłuchujący jest egzemplarzem klasy implementującej specjalny **interfejs nasłuchu** (ang. *listener interface*).
- Źródło zdarzeń to obiekt, który może rejestrować obiekty nasłuchujące i wysyłać do nich obiekty zdarzeń.
- Źródło zdarzeń wysyła obiekty zdarzeń do wszystkich zarejestrowanych słuchaczy w chwili wystąpienia zdarzenia.
- Informacje zawarte w obiekcie zdarzeń są wykorzystywane przez obiekty nasłuchujące przy podejmowaniu decyzji dotyczącej reakcji na zdarzenie.

Rysunek 8.1 przedstawia relacje pomiędzy klasami obsługi zdarzeń a interfejsami

Poniżej znajduje się przykładowa definicja słuchacza:

```
ActionListener listener = . . .;
JButton button = new JButton("Ok");
button.addActionListener(listener);
```



Rysunek 8.1. *Relacje pomiędzy źródłami zdarzeń a słuchaczami*

Od tej pory obiekt `listener` będzie powiadamiany o każdym zdarzeniu akcji w przycisku. Jak się można domyślić, zdarzenie akcji w przypadku przycisku to jego kliknięcie.

Klasa implementująca interfejs `ActionListener` musi definiować metodę o nazwie `actionPerformed`, która jako parametr przyjmuje obiekt typu `ActionEvent`:

```
class MyListener implements ActionListener
{
   . . .
   public void actionPerformed(ActionEvent event)
   {
      // Instrukcje wykonywane w odpowiedzi na kliknięcie przycisku.
      . . .
   }
}
```

Kiedy użytkownik kliknie przycisk, obiekt typu `JButton` tworzy obiekt typu `ActionEvent` i wywołuje metodę `listener.actionPerformed(event)`, przekazując do niej niniejszy obiekt zdarzenia. Źródło zdarzeń, takie jak przycisk, może mieć kilku słuchaczy. W takim przypadku kliknięcie przycisku przez użytkownika powoduje wywołanie metod `actionPerformed` wszystkich słuchaczy.

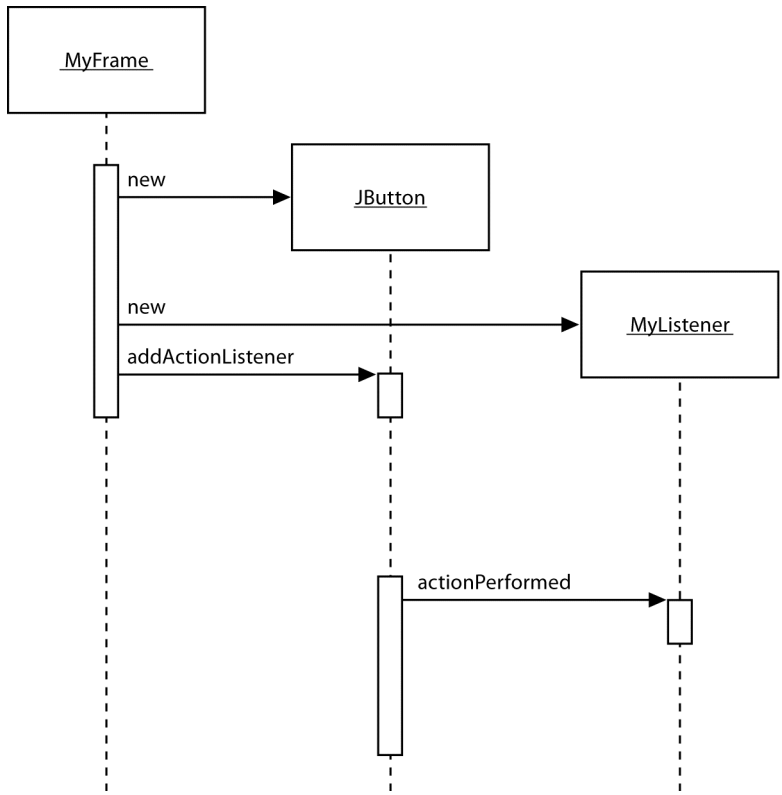
Rysunek 8.2 przedstawia relacje pomiędzy źródłem zdarzeń, słuchaczem zdarzeń a obiektem zdarzeń.

Przykład - obsługa kliknięcia przycisku

Aby nabrać biegłości w posługiwaniu się modelem delegacji zdarzeń, przeanalizujemy szczegółowo prosty program reagujący na kliknięcie przycisku. Utworzymy panel zawierający trzy przyciski, których zdarzeń będą nasłuchiwać trzy obiekty nasłuchujące.

W tym przypadku za każdym razem, gdy użytkownik kliknie jeden z przycisków na panelu, skojarzony z tym przyciskiem obiekt odbierze obiekt typu `ActionEvent` oznaczający kliknięcie przycisku. W odpowiedzi obiekt nasłuchujący zmieni kolor tła panelu.

Przed przejściem do programu, który nasłuchuje kliknięć przycisków, musimy najpierw zapoznać się z techniką tworzenia i dodawania przycisków do panelu (więcej informacji na temat elementów GUI znajduje się w rozdziale 9.).



Rysunek 8.2. *Powiadamianie o zdarzeniach*

Tworzenie przycisku polega na podaniu jego konstruktorowi łańcucha określającego etykietę przycisku, ikony lub jednego i drugiego. Poniżej znajdują się przykłady tworzenia dwóch przycisków:

```
JButton yellowButton = new JButton("Żółty");
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

Przyciski do panelu dodaje się za pomocą metody `add`:

```
JButton yellowButton = new JButton("Żółty");
JButton blueButton = new JButton("Niebieski");
JButton redButton = new JButton("Czerwony");

buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);
```

Wynik powyższych działań przedstawia rysunek 8.3.

Następnie konieczne jest dodanie procedur nasłuchujących tych przycisków. Do tego potrzebne są klasy implementujące interfejs `ActionListener`, który, jak już wspominaliśmy, zawiera tylko jedną metodę: `actionPerformed`. Sygnatura niniejszej metody jest następująca:

```
public void actionPerformed(ActionEvent event)
```

Rysunek 8.3. *Panel z przyciskami*

Interfejs `ActionListener` nie ogranicza się tylko do kliknięć przycisków. Znajduje on zastosowanie w wielu innych sytuacjach, takich jak:

- wybór elementu z pola listy za pomocą dwukrotnego kliknięcia,
- wybór elementu menu,
- kliknięcie klawisza *Enter* w polu tekstowym,
- upływ określonej ilości czasu dla komponentu `Timer`.

Więcej szczegółów na ten temat znajduje się w niniejszym i kolejnym rozdziale.

Sposób użycia interfejsu `ActionListener` jest taki sam we wszystkich sytuacjach: metoda `actionPerformed` (jedyna w interfejsie `ActionListener`) przyjmuje obiekt typu `ActionEvent` jako parametr. Ten obiekt zdarzenia dostarcza informacji o zdarzeniu, które miało miejsce.

Reakcją na kliknięcie przycisku ma być zmiana koloru tła panelu. Żądany kolor będziemy przechowywać w klasie nasłuchującej:

```
class ColorAction implements ActionListener
{
   public ColorAction(Color c)
   {
      backgroundColor = c;
   }

   public void actionPerformed(ActionEvent event)
   {
      // Ustawienie koloru tła panelu.
      . . .
   }
   private Color backgroundColor;
}
```

Następnie dla każdego koloru tworzymy osobny obiekt i każdy z nich rejestrujemy jako słuchacza przycisku.

```
ColorAction yellowAction = new ColorAction(Color.YELLOW);
ColorAction blueAction = new ColorAction(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

Jeśli użytkownik kliknie na przykład przycisk z napisem Żółty, zostanie wywołana metoda actionPerformed obiektu yellowAction. Pole backgroundColor niniejszego obiektu ma wartość color.YELLOW.

Został jeszcze tylko jeden problem do rozwiązania. Obiekt typu ColorAction nie ma dostępu do zmiennej buttonPanel. Można to rozwiązać na jeden z dwóch sposobów. Można zapisać panel w obiekcie ColorAction i skonstruować go w konstruktorze ColorAction. Wygodniej jednak byłoby, gdyby ColorAction była klasą wewnętrzną klasy ButtonFrame. Dzięki temu jej metody miałyby automatycznie dostęp do zewnętrznego panelu (więcej informacji na temat klas wewnętrznych znajduje się w rozdziale 6.).

Zastosujemy drugą z opisanych metod. Poniżej przedstawiamy klasę ColorAction wewnątrz klasy ButtonFrame:

```
class ButtonPanel extends JFrame
{
   . . .
   private class ColorAction implements ActionListener
   {
      . . .
      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }

      private Color backgroundColor;
   }
   private JPanel buttonPanel;
}
```

Przypatrzmy się uważniej metodzie actionPerformed. Klasa ColorAction nie posiada pola buttonPanel. Ma go natomiast zewnętrzna klasa ButtonFrame.

Jest to bardzo często spotykana sytuacja. Obiekty nasłuchu zdarzeń często muszą wykonywać działania, które mają wpływ na inne obiekty. Klasę nasłuchującą często można umieścić w strategicznym miejscu wewnątrz klasy, której obiekt ma mieć zmieniony stan.

Listing 8.1 przedstawia kompletny program. Kliknięcie jednego z przycisków powoduje zmianę koloru tła panelu przez odpowiedniego słuchacza akcji.

Listing 8.1. ButtonTest.java

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

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class ButtonTest
{
```

```
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ButtonFrame frame = new ButtonFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * Ramka z panelem zawierającym przyciski.
 */
class ButtonFrame extends JFrame
{
   public ButtonFrame()
   {
      setTitle("ButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Tworzenie przycisków.
      JButton yellowButton = new JButton("Żółty");
      JButton blueButton = new JButton("Niebieski");
      JButton redButton = new JButton("Czerwony");

      buttonPanel = new JPanel();

      // Dodanie przycisków do panelu.
      buttonPanel.add(yellowButton);
      buttonPanel.add(blueButton);
      buttonPanel.add(redButton);

      // Dodanie panelu do ramki.
      add(buttonPanel);

      // Utworzenie akcji przycisków.
      ColorAction yellowAction = new ColorAction(Color.YELLOW);
      ColorAction blueAction = new ColorAction(Color.BLUE);
      ColorAction redAction = new ColorAction(Color.RED);

      // Powiązanie akcji z przyciskami.
      yellowButton.addActionListener(yellowAction);
      blueButton.addActionListener(blueAction);
      redButton.addActionListener(redAction);
   }

   /**
    * Słuchacz akcji ustawiający kolor tła panelu.
    */
   private class ColorAction implements ActionListener
   {
```

```
        public ColorAction(Color c)
        {
           backgroundColor = c;
        }

        public void actionPerformed(ActionEvent event)
        {
           buttonPanel.setBackground(backgroundColor);
        }

        private Color backgroundColor;
     }

     private JPanel buttonPanel;

     public static final int DEFAULT_WIDTH = 300;
     public static final int DEFAULT_HEIGHT = 200;
  }
```

API javax.swing.JButton **1.2**

- JButton(String label)
- JButton(Icon icon)
- JButton(String label, Icon icon)

  Tworzy przycisk. Łańcuch etykiety może zawierać sam tekst lub (od Java SE 1.3) kod HTML, np. "<html>Ok</html>".

API java.awt.Container **1.0**

- Component add(Component c)

  Dodaje komponent c do kontenera.

API javax.swing.ImageIcon **1.2**

- ImageIcon(String filename)

  Tworzy ikonę, której obraz jest zapisany w pliku.

Nabywanie biegłości w posługiwaniu się klasami wewnętrznymi

Niektórzy programiści nie przepadają za klasami wewnętrznymi, ponieważ uważają, że klasy i obiekty o dużych rozmiarach spowalniają działanie programu. Przyjrzyjmy się temu twierdzeniu. Nie potrzebujemy nowej klasy dla każdego elementu interfejsu użytkownika. W naszym programie wszystkie trzy przyciski współdzielą jedną klasę nasłuchującą. Oczywiście każdy z nich posiada osobny obiekt nasłuchujący. Ale obiekty te nie są duże. Każdy z nich zawiera wartość określającą kolor i referencję do panelu. A tradycyjne rozwiązanie, z zastosowaniem instrukcji if-else, również odwołuje się do tych samych obiektów kolorów przechowywanych przez słuchaczy akcji, tylko że jako zmienne lokalne, a nie pola obiektów.

Poniżej przedstawiamy dobry przykład tego, jak anonimowe klasy wewnętrzne mogą uprościć kod programu. W programie na listingu 8.1 z każdym przyciskiem związane są takie same działania:

1. Utworzenie przycisku z etykietą.
2. Dodanie przycisku do panelu.
3. Utworzenie słuchacza akcji z odpowiednim kolorem.
4. Dodanie słuchacza akcji.

Napiszemy metodę pomocniczą, która będzie upraszczała niniejsze czynności:

```
public void makeButton(String name, Color backgroundColor)
{
JButton button = new JButton(name);
buttonPanel.add(button);
ColorAction action = new ColorAction(backgroundColor);
button.addActionListener(action);
}
```

Teraz wystarczą tylko następujące wywołania:

```
makeButton("żółty", Color.YELLOW);
makeButton("niebieski", Color.BLUE);
makeButton("czerwony", Color.RED);
```

Możliwe są dalsze uproszczenia. Zauważmy, że klasa `ColorAction` jest potrzebna tylko **jeden raz** — w metodzie `makeButton`. A zatem można ją przerobić na klasę anonimową:

```
public void makeButton(String name, final Color backgroundColor)
{
   JButton button = new JButton(name);
   buttonPanel.add(button);
   button.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            buttonPanel.setBackground(backgroundColor);
         }
      });
}
```

Kod słuchacza akcji stał się znacznie prostszy. Metoda `actionPerformed` odwołuje się do zmiennej parametrycznej `backgroundColor` (podobnie jak w przypadku wszystkich zmiennych lokalnych wykorzystywanych w klasie wewnętrznej, parametr ten musi być finalny).

Nie jest potrzebny żaden jawny konstruktor. Jak widzieliśmy w rozdziale 6., mechanizm klas wewnętrznych automatycznie generuje konstruktor zapisujący wszystkie finalne zmienne lokalne, które są używane w jednej z metod klasy wewnętrznej.

Anonimowe klasy wewnętrzne potrafią zmylić niejednego programistę. Można jednak przyzwyczaić się do ich rozszyfrowywania, wyrabiając sobie umiejętność pomijania wzrokiem kodu procedury:

```
button.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }
   });
```

Akcja przycisku ustawia kolor tła. Dopóki procedura obsługi zdarzeń składa się z tylko kilku instrukcji, wydaje się, że z odczytem nie powinno być problemów, zwłaszcza jeśli w sferze naszych zainteresowań nie leżą mechanizmy klas wewnętrznych.

Słuchaczem przycisku może być obiekt **każdej** klasy, która implementuje interfejs `ActionListener`. My wolimy używać obiektów nowej klasy, która została utworzona specjalnie z myślą o wykonywaniu akcji przycisku. Jednak niektórzy programiści nie czują się pewnie w stosowaniu klas wewnętrznych i wybierają inne podejście. Implementują interfejs `ActionListener` w kontenerze źródeł zdarzeń. Następnie kontener ten ustawia **sam siebie** jako słuchacza w następujący sposób:

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

W tej sytuacji żaden z trzech przycisków nie ma osobnego słuchacza. Dysponują one wspólnym obiektem, którym jest ramka przycisku. W związku z tym metoda `actionPerformed` musi sprawdzić, który przycisk został kliknięty.

```
class ButtonFrame extends JFrame implements ActionListener
   {
      . . .
      public void actionPerformed(ActionEvent event)
      {
         Object source = event.getSource();
         if (source == yellowButton) . . .
         else if (source == blueButton) . . .
         else if (source == redButton ) . . .
         else . . .
      }
   }
```

Jak widać, metoda ta jest nieco zagmatwana, przez co nie zalecamy jej stosowania.

API `java.util.EventObject` **1.1**

- `Object setSource()`

  Zwraca referencję do obiektu, w którym wystąpiło zdarzenie.

API java.awt.event.ActionEvent **1.1**

- `String getActionCommand()`

  Zwraca łańcuch polecenia skojarzonego z danym zdarzeniem akcji. Jeśli zdarzenie pochodzi od przycisku, łańcuch polecenia jest taki sam jak etykieta przycisku, chyba że został zmieniony za pomocą metody `setActionCommand`.

API java.beans.EventHandler **1.4**

- `static Object create(Class listenerInterface, Object target, String action)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)`

  Tworzy obiekt klasy pośredniczącej implementującej dany interfejs. Albo podana metoda, albo wszystkie metody interfejsu wykonują dane akcje na rzecz obiektu docelowego.

  Akcją może być metoda lub własność obiektu docelowego. Jeśli jest to własność, wykonywana jest jej metoda ustawiająca. Na przykład akcja `text` jest zamieniana na wywołanie metody `setText`.

  Własność zdarzenia składa się z jednej lub większej liczby nazw własności oddzielonych kropkami. Pierwsza własność jest wczytywana z parametru metody nasłuchującej. Druga własność pochodzi od powstałego obiektu itd. Wynik końcowy staje się parametrem akcji. Na przykład własność `source.text` jest zamieniana na wywołania metod `getSource` i `getText`.

Tworzenie słuchaczy zawierających jedno wywołanie metody

W Java SE 1.4 wprowadzono mechanizm umożliwiający określanie prostych słuchaczy zdarzeń bez tworzenia klas wewnętrznych. Wyobraźmy sobie na przykład, że mamy przycisk z etykietą *Load*, którego procedura obsługi zdarzeń zawiera tylko jedną metodę:

```
frame.loadData();
```

Oczywiście można użyć anonimowej klasy wewnętrznej:

```
loadButton.addActionListener(new ActionListener()
{
   public void actionPerformed(ActionEvent event)
   {
      frame.loadData();
   }
});
```

Ale klasa `EventHandler` może utworzyć takiego słuchacza automatycznie, za pomocą następującego wywołania:

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

Oczywiście nadal konieczne jest zainstalowanie procedury obsługi:

```
loadButton.addActionListener(
   EventHandler.create(ActionListener.class, frame, "loadData"));
```

Jeśli słuchacz wywołuje metodę z jednym parametrem, który można uzyskać z parametru zdarzenia, można użyć innego rodzaju metody `create`. Na przykład wywołanie:

```
EventHandler.create(ActionListener.class, frame, "loadData", "source.text")
```

jest równoznaczne z:

```
new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         frame.loadData(((JTextField) event.getSource()).getText());
      }
   }
```

Nazwy własności `source` i `text` zamieniają się w wywołania metod `getSource` i `getText`.

Przykład - zmiana stylu

Domyślnym stylem programów pisanych przy użyciu Swinga jest Metal. Istnieją dwa sposoby na zmianę stylu. Pierwszy z nich polega na utworzeniu pliku *swing.properties* w katalogu *jre/lib* w miejscu instalacji Javy. W pliku tym należy ustawić własność *swing.defaultlaf* na nazwę klasy stylu, który chcemy zastosować. Na przykład:

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Zauważmy, że styl Metal jest zlokalizowany w pakiecie `javax.swing`. Pozostałe style znajdują się w pakiecie `com.sun.java` i nie muszą być obecne w każdej implementacji Javy. Obecnie ze względu na prawa autorskie pakiety stylów systemów Windows i Macintosh są dostępne wyłącznie z wersjami środowiska uruchomieniowego Javy przeznaczonymi dla tych systemów.

Ponieważ w plikach własności linie zaczynające się od znaku # są ignorowane, można w takim pliku umieścić kilka stylów i wybierać je wedle upodobania, odpowiednio zmieniając położenie znaku #:

```
#swing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Aby zmienić styl w ten sposób, konieczne jest ponowne uruchomienie programu. Programy Swing wczytują plik *swing.properties* tylko jeden raz — przy uruchamianiu.

Drugi sposób polega na dynamicznej zmianie stylu. Należy wywołać statyczną metodę `UIManager.setLookAndFeel` oraz przekazać do niej nazwę klasy wybranego stylu. Następnie wywołujemy statyczną metodę `SwingUtilities.updateComponentTreeUI` w celu odświeżenia całego zbioru komponentów. Metodzie tej wystarczy przekazać tylko jeden komponent, a pozostałe znajdzie ona samodzielnie. Metoda `UIManager.setLookAndFeel` może spowodować

kilka wyjątków, jeśli nie znajdzie żądanego stylu lub jeśli wystąpi błąd podczas ładowania stylu. Jak zwykle nie zgłębiamy kodu obsługującego wyjątki, ponieważ szczegółowo zajmiemy się tym w rozdziale 11.

Poniższy przykładowy fragment programu przedstawia sposób przełączenia na styl Motif:

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
   UIManager.setLookAndFeel(plaf);
   SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e) { e.printStackTrace(); }
```

Aby odnaleźć wszystkie zainstalowane style, należy użyć wywołania:

```
UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
```

W takiej sytuacji nazwę każdego stylu i jego klasy można uzyskać następująco:

```
String name = infos[i].getName();
String className = infos[i].getClassName();
```

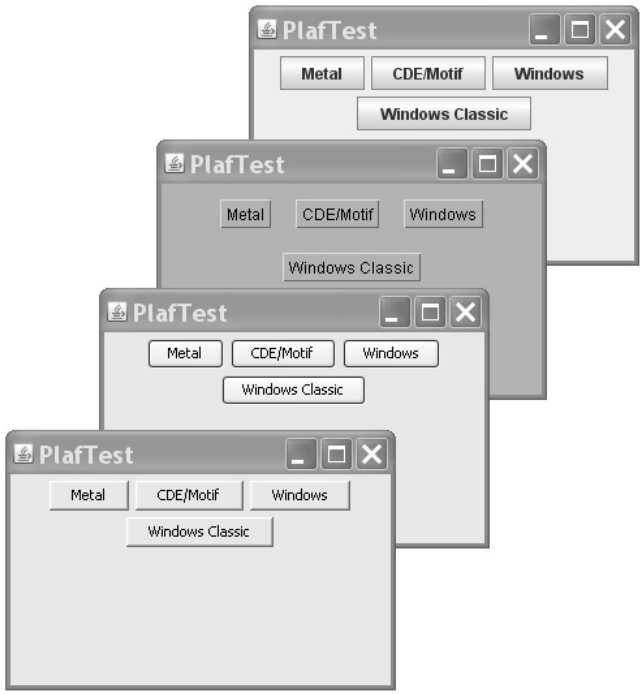
Listing 8.2 przedstawia pełny kod programu demonstrującego przełączanie stylów (zobacz rysunek 8.4). Program ten jest podobny do programu z listingu 8.1. Idąc za radą z poprzedniej sekcji, akcję przycisku, polegającą na zmianie stylu, określiliśmy za pomocą metody pomocniczej `makeButton` i anonimowej klasy wewnętrznej.

Listing 8.2. PlafTest.java

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

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class PlafTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               PlafFrame frame = new PlafFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}
```

```
/**
 * Ramka z panelem zawierającym przyciski zmieniające styl.
 */
class PlafFrame extends JFrame
{
   public PlafFrame()
   {
      setTitle("PlafTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonPanel = new JPanel();

      UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
      for (UIManager.LookAndFeelInfo info : infos)
         makeButton(info.getName(), info.getClassName());

      add(buttonPanel);
   }

   /**
    * Tworzy przycisk zmieniający styl.
    * @param name nazwa przycisku
    * @param plafName nazwa klasy stylu
    */
   void makeButton(String name, final String plafName)
   {
      // Dodanie przycisku do panelu.

      JButton button = new JButton(name);
      buttonPanel.add(button);

      // Ustawienie akcji przycisku.

      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // Akcja przycisku — przełączenie na nowy styl.
               try
               {
                  UIManager.setLookAndFeel(plafName);
                  SwingUtilities.updateComponentTreeUI(PlafFrame.this);
               }
               catch (Exception e)
               {
                  e.printStackTrace();
               }
            }
         });
   }

   private JPanel buttonPanel;

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}
```

Rysunek 8.4.
Zmienianie stylu



Jedna rzecz w niniejszym programie jest godna szczególnej uwagi. Metoda `actionPerformed` wewnętrznej klasy nasłuchującej akcji musi przekazać referencję `this` klasy zewnętrznej `PlafFrame` do metody `updateComponentTreeUI`. Przypomnijmy sobie z rozdziału 6., że przed wskaźnikiem `this` obiektu klasy zewnętrznej musi znajdować się przedrostek w postaci nazwy klasy zewnętrznej:

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this);
```

API `javax.swing.UIManager` **1.2**

- `static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()`

  Tworzy tablicę obiektów reprezentujących zainstalowane style.

- `static setLookAndFeel(String className)`

  Ustawia aktualny styl, wykorzystując do tego podaną nazwę klasy (np. `javax.swing.plaf.metal.MetalLookAndFeel`).

API `javax.swing.UIManager.LookAndFeelInfo` **1.2**

- `String getName()`

  Zwraca nazwę stylu.

- `String getClassName()`

  Zwraca nazwę klasy implementującej dany styl.

Klasy adaptacyjne

Nie wszystkie zdarzenia są tak łatwe w obsłudze jak kliknięcie przycisku. W profesjonalnym programie należy stale sprawdzać, czy użytkownik nie zamyka głównej ramki, aby zapobiec ewentualnej utracie jego danych. Gdy użytkownik zamyka ramkę, powinno wyświetlać się okno dialogowe monitujące o potwierdzenie niniejszego zamiaru.

Kiedy użytkownik zamyka okno, obiekt klasy `JFrame` jest źródłem zdarzenia `WindowEvent`. Aby przechwycić to zdarzenie, konieczny jest odpowiedni obiekt nasłuchujący, który należy dodać do listy słuchaczy okna ramki.

```
WindowListener listener = . . .;
frame.addWindowListener(listener);
```

Obiekt nasłuchujący okna musi należeć do klasy implementującej interfejs `WindowListener`. Interfejs ten zawiera siedem metod. Ramka wywołuje jedną z nich w odpowiedzi na jedno z siedmiu zdarzeń, które mogą mieć miejsce w przypadku okna. Nazwy tych metod mówią same za siebie. Należy tylko wyjaśnić, że `iconfied` w systemie Windows oznacza to samo co `minimized`. Poniżej widać cały interfejs `WindowListener`:

```
public interface WindowListener
{
void windowOpened(WindowEvent e);
void windowClosing(WindowEvent e);
void windowClosed(WindowEvent e);
void windowIconified(WindowEvent e);
void windowDeiconified(WindowEvent e);
void windowActivated(WindowEvent e);
void windowDeactivated(WindowEvent e);
}
```

Aby sprawdzić, czy okno zostało zmaksymalizowane, należy zainstalować obiekt `WindowStateListener`. Zobacz wyciąg z API na stronie 373.

W Javie klasa, która implementuje dany interfejs, musi definiować wszystkie jego metody. W tym przypadku oznacza to implementację **siedmiu** metod. Przypomnijmy jednak, że interesuje nas tylko jedna z nich, o nazwie `windowClosing`.

Oczywiście nic nie stoi na przeszkodzie, aby zaimplementować niniejszy interfejs, wstawić wywołanie `System.exit(0)` do metody `windowClosing` i napisać sześć nicnierobiących funkcji dla pozostałych metod:

```
class Terminator implements WindowListener
{
public void windowClosing(WindowEvent e)
{
if (użytkownik potwierdza)
System.exit(0);
}
public void windowOpened(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
```

```
public void windowDeiconified(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
}
```

Pisanie sześciu metod, które nic nie robią, jest tym rodzajem pracy, której nikt nie lubi. Zadanie to ułatwiają **klasy adaptacyjne** (ang. *adapter class*) dostępne z każdym interfejsem nasłuchującym w bibliotece AWT, który ma więcej niż jedną metodę. Klasy te implementują wszystkie metody interfejsów, którym odpowiadają, ale metody te nic nie robią. Na przykład klasa WindowAdapter zawiera definicje siedmiu nicnierobiących metod. Oznacza to, że klasa adaptacyjna automatycznie spełnia wymagania techniczne stawiane przez Javę, a dotyczące implementacji odpowiadającego jej interfejsu nasłuchującego. Klasę adaptacyjną można rozszerzyć, definiując w podklasie metody odpowiadające niektórym, ale nie wszystkim typom zdarzeń interfejsu (interfejsy, które mają tylko jedną metodę, np. ActionListener, nie potrzebują metod adaptacyjnych).

Rozszerzymy klasę WindowAdapter. Odziedziczymy po niej sześć nicnierobiących metod, a metodę windowClossing przesłonimy:

```
class Terminator extends WindowAdapter
{
   public void windowClosing(WindowEvent e)
   {
      if (użytkownik potwierdza)
         System.exit(0);
   }
}
```

Teraz możemy zarejestrować obiekt typu Terminator jako słuchacza zdarzeń:

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

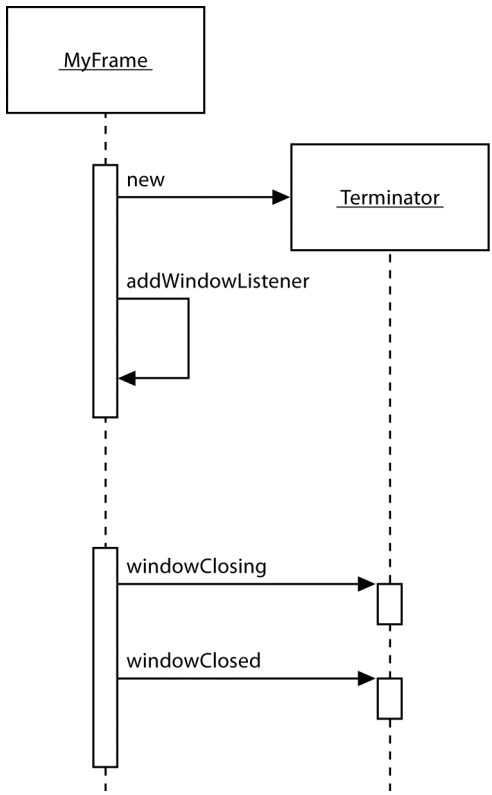
Każde zdarzenie okna wygenerowane przez ramkę jest przekazywane do obiektu listener za pomocą wywołania jednej z jego siedmiu metod (zobacz rysunek 8.5). Sześć z nich nie robi nic, a metoda windowClosing wywołuje metodę System.exit(0), zamykając tym samym aplikację.

Jeśli w nazwie metody rozszerzanej klasy adaptacyjnej znajdzie się błąd, kompilator go nie wykryje. Jeśli na przykład w klasie rozszerzającej WindowAdapter zostanie zdefiniowana metoda windowIsClosing, nowa klasa będzie zawierała osiem metod, a metoda windowClosing nic nie będzie robiła.

Utworzenie klasy rozszerzającej klasę adaptacyjną WindowAdapter jest krokiem naprzód, ale można posunąć się jeszcze dalej. Nie ma potrzeby nadawać obiektowi listener nazwy. Wystarczy napisać:

```
frame.addWindowListener(new Terminator());
```

Ale czemu poprzestawać na tym? Klasa nasłuchująca może być anonimową klasą wewnętrzną ramki.



Rysunek 8.5. *Obiekt nasłuchujący zdarzeń dotyczących okna*

```
frame.addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent e)
      {
         if (użytkownik potwierdza)
            System.exit(0);
      }
   });
```

Powyższy fragment programu ma następujące działanie:

- Definiuje klasę bez nazwy, rozszerzającą klasę `WindowAdapter`.
- Do utworzonej anonimowej klasy dodaje metodę `windowClosing` (podobnie jak wcześniej, metoda ta zamyka program).
- Dziedziczy sześć pozostałych nicnierobiących metod po klasie `WindowAdapter`.
- Tworzy obiekt tej nowej klasy — obiekt również nie ma nazwy.
- Przekazuje niniejszy obiekt do metody `addWindowListener`.

Powtarzamy jeszcze raz, że do składni wewnętrznych klas anonimowych trzeba się przyzwyczaić. Dzięki nim można pisać tak zwięzły kod, jak to tylko możliwe.

API `java.awt.event.WindowListener` **1.1**

- `void windowOpened(WindowEvent e)`

  Jest wywoływana po otwarciu okna.

- `void windowClosing(WindowEvent e)`

  Jest wywoływana, kiedy użytkownik wyda polecenie menedżera okien, aby zamknąć okno. Okno zostanie zamknięte tylko wtedy, gdy zostanie wywołana jego metoda `hide` lub `dispose`.

- `void windowClosed(WindowEvent e)`

  Jest wywoływana po zamknięciu okna.

- `void windowIconified(WindowEvent e)`

  Jest wywoływana po zminimalizowaniu okna.

- `void windowDeiconified(WindowEvent e)`

  Jest wywoływana po przywróceniu okna.

- `void windowActivated(WindowEvent e)`

  Jest wywoływana po uaktywnieniu okna. Aktywna może być tylko ramka lub okno dialogowe. Z reguły menedżer okien zaznacza w jakiś sposób aktywne okno — np. podświetlając pasek tytułu.

- `void WindowDeactivated(WindowEvent e)`

  Jest wywoływana po dezaktywowaniu okna.

API `java.awt.event.WindowStateListener` **1.4**

- `void windowStateChanged(WindowEvent event)`

  Jest wywoływana po zmaksymalizowaniu, zminimalizowaniu lub przywróceniu do normalnego rozmiaru okna.

API `java.awt.event.WindowEvent` **1.1**

- `int getNewState() 1.4`
- `int getOldState() 1.4`

  Zwraca nowy i stary stan okna w zdarzeniu zmiany stanu okna. Zwracana liczba całkowita jest jedną z następujących wartości:

```
Frame.NORMAL
Frame.ICONIFIED
Frame.MAXIMIZED_HORIZ
Frame.MAXIMIZED_VERT
Frame.MAXIMIZED_BOTH
```

Akcje

Często jedną opcję można wybrać na kilka różnych sposobów. Użytkownik może wybrać odpowiednią funkcję w menu, nacisnąć określony klawisz lub przycisk na pasku narzędzi. Zaprogramowanie takiej funkcjonalności w modelu zdarzeń `AWT` jest proste — należy wszystkie zdarzenia związać z tym samym obiektem nasłuchującym. Wyobraźmy sobie, że `blueAction` jest obiektem nasłuchującym akcji, którego metoda `actionPerformed` zmienia kolor tła na niebieski. Jeden obiekt można związać jako słuchacza z kilkoma źródłami zdarzeń:

- przyciskiem paska narzędzi z etykietą *Niebieski*;
- elementem menu z etykietą *Niebieski*;
- skrótem klawiszowym *Ctrl+N*.

Dzięki temu zmiana koloru będzie wykonywana zawsze w taki sam sposób, bez znaczenia, czy wywoła ją kliknięcie przycisku, wybór elementu menu, czy naciśnięcie klawisza.

W pakiecie Swing dostępna jest niezwykle przydatna struktura opakowująca polecenia i wiążąca je z różnymi źródłami zdarzeń — interfejs `Action`. **Akcja** to obiekt, który opakowuje:

- opis polecenia (łańcuch tekstowy i opcjonalna ikona),
- parametry niezbędne do wykonania polecenia (w naszym przypadku wymagany kolor).

Interfejs `Action` zawiera następujące metody:

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

Pierwsza z tych metod jest już nam znana z interfejsu `ActionListener`. Należy dodać, że interfejs `Action` rozszerza interfejs `ActionListener`. W związku z tym wszędzie, gdzie powinien znaleźć się obiekt `ActionListener`, można użyć obiektu `Action`.

Dwie kolejne metody włączają i wyłączają akcję oraz sprawdzają, czy akcja jest aktualnie włączona. Kiedy akcja jest związana z menu lub paskiem narzędzi i jest wyłączona, odpowiadająca jej opcja ma kolor szary.

Metody `putValue` i `getValue` zapisują i pobierają pary nazwa – wartość z obiektu akcji. Nazwy akcji i ikony są zapisywane w obiektach akcji za pomocą dwóch predefiniowanych łańcuchów: `Action.NAME` i `Action.SMALL_ICON`:

```
action.putValue(Action.NAME, "Niebieski");
action.putValue(Action.SMALL_ICON, new ImageIcon("blue-ball.gif"));
```

Tabela 8.1 przedstawia zestawienie wszystkich predefiniowanych nazw tablicowych akcji.

Tabela 8.1. *Predefiniowane stałe interfejsu Action*

Nazwa	Wartość
NAME	Nazwa akcji — wyświetlana na przyciskach i elementach menu.
SMALL_ICON	Mała ikona — może być wyświetlana na przyciskach, pasku narzędzi lub elementach menu.
SHORT_DESCRIPTION	Krótki opis ikony — wyświetlany w etykiecie narzędzia.
LONG_DESCRIPTION	Długi opis ikony — do użytku w pomocy internetowej. Żaden komponent Swinga nie używa tej wartości.
MNEMONIC_KEY	Skrót akcji — wyświetlany na elementach menu (zobacz rozdział 9.).
ACCELERATOR_KEY	Skrót klawiaturowy. Żaden komponent Swinga nie używa tej wartości.
ACTION_COMMAND_KEY	Używana w przestarzałej już metodzie `registerKeyboardAction`.
DEFAULT	Własność pasująca do wszystkiego. Żaden komponent Swinga nie używa tej wartości.

Jeśli obiekt akcji jest dodawany do menu lub paska narzędzi, jego nazwa i ikona są automatycznie pobierane i wyświetlane w menu lub na pasku narzędzi. Wartość własności `SHORT_DESCRIPTION` zamienia się w dymek opisujący narzędzie.

Pozostałe dwie metody interfejsu `Action` umożliwiają powiadamianie innych obiektów, zwłaszcza menu i pasków narzędzi, które są źródłem akcji, o zmianach własności obiektu akcji. Jeśli na przykład menu jest dodawane jako obiekt nasłuchujący zmian własności obiektu akcji i obiekt ten zostanie następnie wyłączony, menu zostanie wywołane, a nazwa akcji będzie szara. Obiekty nasłuchu zmian własności są ogólną konstrukcją stanowiącą część modelu komponentów JavaBean. Więcej informacji na temat Beanów i ich własności znajduje się w drugim tomie.

Nie należy zapominać, że `Action` to interfejs, a nie klasa. Każda klasa implementująca go musi definiować wszystkie siedem metod, które opisaliśmy. Na szczęście jakiś dobry człowiek napisał klasę o nazwie `AbstractAction`, która implementuje wszystkie niniejsze metody z wyjątkiem `actionPerformed`. Klasa ta zajmuje się zapisywaniem par nazwa – wartość i zarządzaniem obiektami nasłuchującymi zmian własności. Wystarczy rozszerzyć klasę `AbstractAction` i zdefiniować metodę `actionPerformed`.

Utworzymy obiekt wykonujący polecenia zmiany koloru. Zapiszemy nazwę polecenia, ikonę i żądany kolor. Kolor zapiszemy w tablicy par nazwa – wartość dostarczanej przez klasę `AbstractAction`. Poniżej znajduje się kod źródłowy klasy `ColorAction`. Konstruktor ustawia pary nazwa – wartość, a metoda `actionPerformed` wykonuje akcję zmiany koloru.

```
public class ColorAction extends AbstractAction
{
   public ColorAction(String name, Icon icon, Color c)
   {
      putValue(Action.NAME, name);
      putValue(Action.SMALL_ICON, icon);
      putValue("color", c);
      putValue(Action.SHORT_DESCRIPTION, "Ustaw kolor panelu na " + name.toLowerCase());
   }
```

```
    public void actionPerformed(ActionEvent event)
    {
       Color c = (Color) getValue("color");
       buttonPanel.setBackground(c);
    }
}
```

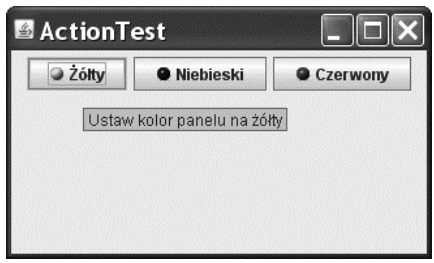
Nasz przykładowy program tworzy trzy obiekty niniejszej klasy, np.:

```
Action blueAction = new ColorAction("Niebieski", new ImageIcon("blue-ball.gif"),
➥Color.BLUE);
```

Teraz konieczne jest związanie akcji z przyciskiem. Jest to łatwe, ponieważ możemy użyć konstruktora `JButton`, który przyjmuje obiekt typu `Action`.

```
JButton blueButton = new JButton(blueAction);
```

Konstruktor odczytuje nazwę i ikonę z akcji, ustawia krótki opis jako etykietę oraz ustawia akcję jako słuchacza. Ikony i etykietę przedstawia rysunek 8.6.



Rysunek 8.6. *Przyciski zawierają ikony z obiektów akcji*

W kolejnym rozdziale wykażemy, że tak samo łatwe jest dodawanie tej samej akcji do menu.

Na koniec przypiszemy obiekty akcji do klawiszy, dzięki czemu akcje te będą wykonywane, kiedy użytkownik wpisze polecenia z klawiatury. Kojarzenie akcji z klawiszami należy zacząć od wygenerowania obiektu klasy `KeyStroke`. Klasa ta opakowuje opis klawisza. Do utworzenia obiektu typu `KeyStroke` nie używa się konstruktora, ale statycznej metody `getKeyStroke` klasy `KeyStroke`.

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl N");
```

Do zrozumienia następnego etapu potrzebna jest znajomość pojęcia **aktywności komponentu** (ang. *keyboard focus*). Interfejs użytkownika może składać się z wielu przycisków, menu, pasków przewijania i innych komponentów. Kiedy zostanie naciśnięty klawisz, zdarzenie to zostaje wysłane do aktywnego komponentu. Komponent ten jest z reguły (choć nie zawsze) w jakiś sposób wizualnie wyróżniony. Na przykład w stylu Javy tekst na aktywnym przycisku ma cienką obwódkę. Fokus (aktywność komponentu) można przenosić na różne komponenty za pomocą klawisza *Tab*. Naciśnięcie klawisza spacji powoduje kliknięcie aktywnego przycisku. Inne klawisze wywołują inne działania. Na przykład klawisze strzałek mogą sterować paskiem przewijania.

Jednak my nie chcemy wysyłać zdarzenia naciśnięcia klawisza do aktywnego komponentu. W przeciwnym razie każdy przycisk musiałby znać procedurę obsługi kombinacji klawiszy *Ctrl+Y*, *Ctrl+B* i *Ctrl+R*.

Jest to bardzo powszechny problem. Jednak projektanci biblioteki Swing znaleźli dla niego proste rozwiązanie. Każdy `JComponent` posiada trzy **mapy wejścia** (ang. *input maps*), z których każda odwzorowuje obiekty `KeyStroke` na związane z nimi akcje. Mapy te odpowiadają trzem różnym sytuacjom (zobacz tabela 8.2).

Tabela 8.2. *Mapy klawiaturowe*

Znacznik	Wywołuje działanie, gdy
`WHEN_FOCUSED`	komponent jest aktywny;
`WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`	komponent zawiera komponent aktywny;
`WHEN_IN_FOCUSED_WINDOW`	komponent znajduje się w tym samym oknie co komponent aktywny.

Niniejsze mapy są sprawdzane w następującej kolejności w wyniku naciśnięcia klawisza:

1. Sprawdzenie mapy `WHEN_FOCUSED` aktywnego komponentu. Jeśli dany skrót klawiaturowy istnieje, następuje wykonanie powiązanego z nim działania. Jeśli działanie zostaje wykonane, następuje zatrzymanie sprawdzania warunków.
2. Następuje sprawdzenie map `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` aktywnego komponentu, a następnie jego komponentów nadrzędnych. Gdy zostanie znaleziona mapa z danym skrótem klawiaturowym, następuje wykonanie działania. Jeśli działanie zostaje wykonane, następuje zatrzymanie sprawdzania warunków.
3. Odszukanie wszystkich **widocznych** i **włączonych** komponentów w aktywnym oknie, w których mapie `WHEN_IN_FOCUSED_WINDOW` znajduje się dany skrót klawiaturowy. Umożliwienie tym komponentom (w kolejności zgodnej z rejestracją zdarzeń naciśnięcia klawisza) wykonania odpowiednich działań. Po wykonaniu pierwszego działania następuje zatrzymanie przetwarzania. Ta część procesu może być źródłem problemów, jeśli dany skrót klawiaturowy pojawia się w więcej niż jednej mapie `WHEN_IN_FOCUSED_WINDOW`.

Mapę wejścia komponentu tworzy się za pomocą metody `getInputMap`. Na przykład:

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

Warunek `WHEN_FOCUSED` powoduje, że ta mapa będzie sprawdzana, gdy komponent jest aktywny. Nam potrzebna jest inna mapa. Aktywny jest jeden z przycisków, nie panel. Do wstawienia skrótów klawiszy zmieniających kolor nadaje się jedna z pozostałych dwóch map. W naszym przykładowym programie użyjemy mapy `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`.

Klasa `InputMap` nie odwzorowuje bezpośrednio obiektów `KeyStroke` w postaci obiektów `Action`. W zamian odwzorowuje w postaci dowolnych obiektów, a druga mapa, zaimplementowana w klasie `ActionMap`, mapuje obiekty na akcje. Dzięki temu łatwiej jest współdzielić te same akcje przez skróty klawiaturowe pochodzące z różnych map wejścia.

A zatem każdy komponent posiada trzy mapy wejścia i jedną mapę akcji (ang. *action map*). Aby je powiązać, trzeba wymyślić nazwy dla akcji. Klawisz można powiązać z akcją w następujący sposób:

```
imap.put(KeyStroke.getKeyStroke("ctrl Z"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

W przypadku akcji niewykonującej żadnych działań zwyczajowo stosuje się łańcuch `none`. W ten sposób można łatwo dezaktywować klawisz:

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```

Dokumentacja JDK zaleca stosowanie jako klucza akcji jej nazwy. Naszym zdaniem nie jest to dobre rozwiązanie. Nazwa akcji jest wyświetlana na przyciskach i elementach menu, w związku z czym może się zmieniać w zależności od kaprysu projektanta interfejsu oraz może być przetłumaczona na wiele języków. Takie niestałe łańcuchy nie są dobrym wyborem w przypadku klawiszy wyszukiwania. Zalecamy wymyślenie nazw akcji niezależnych od wyświetlanych nazw.

Poniżej znajduje się zestawienie działań, które trzeba wykonać, aby wywołać to samo działanie w odpowiedzi na zdarzenie naciśnięcia przycisku, wyboru elementu z menu lub naciśnięcia klawisza:

1. Utwórz podklasę klasy `AbstractAction`. Można użyć tej samej klasy dla wielu spokrewnionych akcji.
2. Utwórz obiekt powyższej klasy akcji.
3. Utwórz przycisk lub element menu z obiektu powyższej klasy akcji. Konstruktor odczyta etykietę i ikonę z tego obiektu.
4. W przypadku akcji uruchamianych przez naciśnięcie klawisza konieczne jest wykonanie dodatkowych czynności. Najpierw należy zlokalizować komponent najwyższego poziomu w oknie, np. panel zawierający wszystkie pozostałe elementy.
5. Pobierz mapę `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` komponentu najwyższego poziomu. Utwórz obiekt klasy `KeyStroke` reprezentujący odpowiedni skrót klawiaturowy. Utwórz obiekt będący kluczem działania, np. łańcuch opisujący akcję. Wstaw niniejszą parę danych (klawisz, klucz działania) do mapy wejścia.
6. Pobierz mapę akcji komponentu najwyższego poziomu. Dodaj parę klucz akcji – obiekt akcji do tej mapy.

Listing 8.3 przedstawia kompletny kod programu mapującego przyciski i klawisze na obiekty akcji. Można go wypróbować — kliknięcie jednego z przycisków lub naciśnięcie kombinacji klawiszy *Ctrl+Z*, *Ctrl+N* lub *Ctrl+C* spowoduje zmianę koloru panelu.

Listing 8.3. ActionTest.java

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

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
```

```
public class ActionTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ActionFrame frame = new ActionFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * Ramka z panelem, który demonstruje akcje zmiany koloru.
 */
class ActionFrame extends JFrame
{
   public ActionFrame()
   {
      setTitle("ActionTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonPanel = new JPanel();

      // Definicje akcji.
      Action yellowAction = new ColorAction("Żółty", new ImageIcon("yellow-ball.gif"),
            Color.YELLOW);
      Action blueAction = new ColorAction("Niebieski", new ImageIcon("blue-ball.gif"),
      ↪Color.BLUE);
      Action redAction = new ColorAction("Czerwony", new ImageIcon("red-ball.gif"),
      ↪Color.RED);

      // Dodanie przycisków dla akcji.
      buttonPanel.add(new JButton(yellowAction));
      buttonPanel.add(new JButton(blueAction));
      buttonPanel.add(new JButton(redAction));

      // Dodanie panelu do ramki.
      add(buttonPanel);

      // Powiązanie klawiszy Z, N i C z nazwami.
      InputMap imap = buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_
      ↪COMPONENT);
      imap.put(KeyStroke.getKeyStroke("ctrl Z"), "panel.yellow");
      imap.put(KeyStroke.getKeyStroke("ctrl N"), "panel.blue");
      imap.put(KeyStroke.getKeyStroke("ctrl C"), "panel.red");

      // Powiązanie nazw z akcjami.
      ActionMap amap = buttonPanel.getActionMap();
      amap.put("panel.yellow", yellowAction);
      amap.put("panel.blue", blueAction);
      amap.put("panel.red", redAction);
   }
```

```
public class ColorAction extends AbstractAction
{
   /**
    * Tworzy akcję zmiany koloru.
    * @param name nazwa, która pojawi się na przycisku
    * @param icon ikona, która pojawi się na przycisku
    * @param c kolor tła
    */
   public ColorAction(String name, Icon icon, Color c)
   {
      putValue(Action.NAME, name);
      putValue(Action.SMALL_ICON, icon);
      putValue(Action.SHORT_DESCRIPTION, "Ustaw kolor panelu na " + name.
      ↪toLowerCase());
      putValue("color", c);
   }

   public void actionPerformed(ActionEvent event)
   {
      Color c = (Color) getValue("color");
      buttonPanel.setBackground(c);
   }
}

private JPanel buttonPanel;

public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;
}
```

API `javax.swing.Action` **1.2**

- `boolean isEnabled()`
- `void setEnabled(boolean b)`

  Pobiera lub ustawia własność `enabled` akcji.

- `void putValue(String key, Object value)`

  Wstawia parę nazwa – wartość do obiektu akcji.

  Parametry: `key` Nazwa własności, która ma zostać zapisana z obiektem akcji. Może to być dowolny łańcuch, ale jest kilka nazw o z góry zdefiniowanym znaczeniu — zobacz tabelę 8.1 na stronie 375.

  `value` Obiekt powiązany z nazwą.

- `Object getValue(String key)`

  Zwraca wartość z zapisanej pary nazwa – wartość.

API `javax.swing.KeyStroke` **1.2**

- `static KeyStroke getKeyStroke(String description)`

  Tworzy skrót klawiaturowy z czytelnego dla człowieka opisu (ciągu łańcuchów rozdzielonych spacjami). Opis zaczyna się od zera lub większej liczby modyfikatorów `shift control ctrl meta alt altGraph`, a kończy się łańcuchem `typed` i łańcuchem składającym się z jednego znaku (na przykład `typed a`) lub opcjonalnym specyfikatorem zdarzenia (`pressed` — domyślny lub `released`) i kodem klawisza. Kod klawisza, jeśli ma przedrostek `VK_`, powinien odpowiadać stałej `KeyEvent`, na przykład `INSERT` odpowiada `KeyEvent.VK_INSERT`.

API `javax.swing.JComponent` **1.2**

- `ActionMap getActionMap()` **1.3**

  Zwraca mapę wiążącą klucze mapy akcji (które mogą być dowolnymi obiektami) z obiektami klasy `Action`.

- `InputMap getInputMap(int flag)` **1.3**

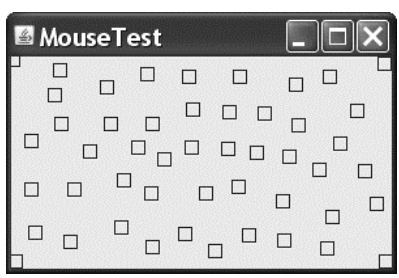
  Pobiera mapę wejścia, która odwzorowuje klawisze w postaci kluczy mapy akcji.

  Parametry: `flag` Warunek określający, kiedy element aktywny ma wywołać akcję — jedna z wartości z tabeli 8.2 na stronie 377.

Zdarzenia generowane przez mysz

Takie zdarzenia jak kliknięcie przycisku lub elementu w menu za pomocą myszy nie wymagają pisania procedur obsługi. Niniejsze zdarzenia są obsługiwane automatycznie przez różne elementy interfejsu użytkownika. Aby jednak umożliwić rysowanie za pomocą myszy, konieczne jest przechwycenie zdarzeń ruchu, kliknięcia i przeciągania myszy.

W niniejszym podrozdziale prezentujemy prosty edytor grafiki pozwalający umieszczać, przesuwać i usuwać kwadraty z obszaru roboczego (zobacz rysunek 8.7).



Rysunek 8.7. *Program obsługujący zdarzenia myszy*

Kiedy użytkownik naciśnie przycisk myszy, wywoływane są trzy metody nasłuchujące: `mousePressed` po naciśnięciu przycisku, `mouseReleased` po zwolnieniu przycisku myszy i `mouseClicked`. Jeśli w sferze zainteresowań leżą wyłącznie pełne kliknięcia, pierwsze dwie z wymienionych metod można pominąć. Wywołując metody `getX` i `getY` na rzecz obiektu klasy `MouseEvent`, można sprawdzić współrzędne `x` i `y` wskaźnika myszy w chwili kliknięcia. Do rozróżnienia pojedynczych, podwójnych i potrójnych (!) kliknięć służy metoda `getClickCount`.

Niektórzy projektanci interfejsów tworzą kombinacje klawiszy połączone z kliknięciami myszką, np. *Ctrl*+*Shift*+kliknięcie. Naszym zdaniem jest to postępowanie niegodne naśladowania. Osoby, które nie zgadzają się z naszą opinią, może przekonać fakt, że sprawdzanie przycisków myszy i klawiszy specjalnych jest niezwykle zagmatwanym zadaniem — niebawem się o tym przekonamy.

Aby sprawdzić, które modyfikatory zostały ustawione, należy użyć maski bitowej. W oryginalnym API dwie maski przycisków są równoważne z maskami klawiszy specjalnych, mianowicie:

```
BUTTON2_MASK == ALT_MASK
BUTTON3_MASK == META_MASK
```

Zrobiono tak, aby użytkownicy posiadający myszkę z jednym przyciskiem mogli naśladować pozostałe przyciski za pomocą klawiszy specjalnych (ang. *modifier keys*). Od Java SE 1.4 zaproponowano jednak inną metodę. Od tej pory istnieją następujące maski:

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

Metoda `getModifiersEx` zwraca dokładne informacje o przyciskach myszy i klawiszach specjalnych użytych w zdarzeniu myszy.

Pamiętajmy, że maska `BUTTON3_DOWN_MASK` w systemie Windows sprawdza prawy (nie główny) przycisk myszy. Na przykład poniższy fragment programu sprawdza, czy prawy przycisk myszy jest wciśnięty:

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)
   . . . // procedury obsługi zdarzenia kliknięcia prawym przyciskiem myszy
```

W przykładowym programie definiujemy zarówno metodę `mousePressed`, jak i `mouseClicked`. Jeśli użytkownik kliknie piksel nieznajdujący się w obrębie żadnego z narysowanych kwadratów, zostanie dodany nowy kwadrat. Działanie to zostało zaimplementowane w metodzie `mousePressed`, a więc kwadrat pojawia się natychmiast po kliknięciu, przed zwolnieniem przycisku myszy. Dwukrotne kliknięcie przyciskiem myszy w obrębie narysowanego kwadratu powoduje jego usunięcie. Implementacja tej funkcji została umieszczona w metodzie `mouseClicked`, ponieważ konieczne jest sprawdzenie liczby kliknięć:

```
public void mousePressed(MouseEvent event)
{
   current = find(event.getPoint());
   if (current == null)    // nie w obrębie żadnego kwadratu
      add(event.getPoint());
}

public void mouseClicked(MouseEvent event)
{
   current = find(event.getPoint());
   if (current != null && event.getClickCount() >= 2)
      remove(current);
}
```

Kiedy kursor myszy przesuwa się nad oknem, odbiera ono stały strumień zdarzeń ruchu myszy. Zauważmy, że są osobne interfejsy `MouseListener` i `MouseMotionListener`. Wyróżniono je z chęci zwiększenia efektywności. Kiedy użytkownik przesuwa mysz, powstaje cała masa zdarzeń dotyczących tej czynności. Obiekt nasłuchujący, który oczekuje na kliknięcia, nie jest zajmowany przez nieinteresujące go zdarzenia ruchu.

Nasz testowy program przechwytuje zdarzenia ruchu i w odpowiedzi na nie zmienia wygląd kursora (na krzyżyk). Odpowiedzialna jest za to metoda `getPredefinedCursor` z klasy `Cursor`. Tabela 8.3 przedstawia stałe podawane jako argument wspomnianej funkcji oraz reprezentowane przez nie kursory w systemie Windows.

Tabela 8.3. *Przykładowe kursory*

Ikona	Stała	Ikona	Stała
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

Poniżej znajduje się kod źródłowy metody `mouseMoved` z klasy `MouseMotionListener` zdefiniowanej w naszym przykładowym programie:

```
public void mouseMoved(MouseEvent event)
{
   if (find(event.getPoint()) == null)
      setCursor(Cursor.getDefaultCursor());
   else
      setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
}
```

Można zdefiniować własne typy kursorów. Służy do tego metoda `createCustomCursor` z klasy `Toolkit`:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor = tk.createCustomCursor(img, new Point(10, 10), "dynamite
➥stick");
```

Pierwszy argument niniejszej metody określa plik graficzny przedstawiający kursor. Drugi wyznacza przesunięcie punktu aktywnego kursora. Trzeci jest łańcuchem opisującym kursor. Łańcuch ten może służyć zwiększeniu dostępności. Na przykład program czytający z ekranu używany przez osobę niedowidzącą może przeczytać opis takiego kursora.

Jeśli w czasie przesuwania myszy użytkownik kliknie jej przycisk, generowane są wywołania metody `mouseDragged` zamiast `mouseMoved`. Nasz przykładowy program zezwala na przeciąganie kwadratów pod kursorem. Efekt ten uzyskaliśmy, aktualizując położenie przeciąganego kwadratu, tak aby jego środek znajdował się w tym samym miejscu co punkt centralny myszki. Następnie ponownie rysujemy obszar roboczy, aby ukazać nowe położenie kursora myszy.

```
public void mouseDragged(MouseEvent event)
{
   if (current != null)
   {
      int x = event.getX();
      int y = event.getY();

      current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
      repaint();
}
```

Metoda `mouseMoved` jest wywoływana tylko wtedy, gdy kursor znajduje się w obrębie komponentu. Natomiast metoda `mouseDragged` jest wywoływana nawet wtedy, gdy kursor opuści komponent.

Istnieją jeszcze dwie inne metody obsługujące zdarzenia myszy: `mouseEntered` i `mouseExited`. Są one wywoływane, gdy kursor myszy wchodzi do komponentu lub go opuszcza.

Na zakończenie wyjaśnimy sposób nasłuchiwania zdarzeń generowanych przez mysz. Kliknięcia przyciskiem myszy są raportowane przez metodę `mouseClicked` należącą do interfejsu `MouseListener`. Ponieważ wiele aplikacji korzysta wyłącznie z kliknięć myszką i występują one bardzo często, zdarzenia ruchu myszy i przeciągania zostały zdefiniowane w osobnym interfejsie o nazwie `MouseMotionListener`.

W naszym programie interesują nas oba rodzaje zdarzeń generowanych przez mysz. Zdefiniowaliśmy dwie klasy wewnętrzne o nazwach `MouseHandler` i `MouseMotionHandler`. Pierwsza z nich jest podklasą klasy `MouseAdapter`, ponieważ definiuje tylko dwie z pięciu metod interfejsu `MouseListener`. Klasa `MouseMotionHandler` implementuje interfejs `MouseMotionListener`, co znaczy, że zawiera definicje obu jego metod. Listing 8.4 przedstawia kod źródłowy omawianego programu.

Listing 8.4. MouseTest.java

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

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class MouseTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               MouseFrame frame = new MouseFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * Ramka zawierająca panel testujący działania myszy.
 */
class MouseFrame extends JFrame
{
   public MouseFrame()
   {
      setTitle("MouseTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Dodanie komponentu do ramki.

      MouseComponent component = new MouseComponent();
      add(component);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
 * Komponent z działaniami myszy, do którego można dodawać (lub z którego można usuwać) kwadraty.
 */
class MouseComponent extends JComponent
{
   public MouseComponent()
   {
      squares = new ArrayList<Rectangle2D>();
      current = null;
```

```
      addMouseListener(new MouseHandler());
      addMouseMotionListener(new MouseMotionHandler());
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;

      // Rysowanie wszystkich kwadratów.
      for (Rectangle2D r : squares)
         g2.draw(r);
   }

   /**
    * Znajduje pierwszy kwadrat zawierający punkt.
    * @param p punkt
    * @return pierwszy kwadrat zawierający punkt p
    */
   public Rectangle2D find(Point2D p)
   {
      for (Rectangle2D r : squares)
      {
         if (r.contains(p)) return r;
      }
      return null;
   }

   /**
    * Dodaje kwadrat do zbioru.
    * @param p środek kwadratu
    */
   public void add(Point2D p)
   {
      double x = p.getX();
      double y = p.getY();

      current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH,
            SIDELENGTH);
      squares.add(current);
      repaint();
   }

   /**
    * Usuwa kwadrat ze zbioru.
    * @param s kwadrat, który ma być usunięty
    */
   public void remove(Rectangle2D s)
   {
      if (s == null) return;
      if (s == current) current = null;
      squares.remove(s);
      repaint();
   }

   private static final int SIDELENGTH = 10;
   private ArrayList<Rectangle2D> squares;
   private Rectangle2D current;
```

```
   // Kwadrat zawierający kursor myszy.

   private class MouseHandler extends MouseAdapter
   {
      public void mousePressed(MouseEvent event)
      {
         // Dodanie nowego kwadratu, jeśli kursor nie jest wewnątrz innego kwadratu.
         current = find(event.getPoint());
         if (current == null) add(event.getPoint());
      }

      public void mouseClicked(MouseEvent event)
      {
         // Usunięcie kwadratu w wyniku jego dwukrotnego kliknięcia.
         current = find(event.getPoint());
         if (current != null && event.getClickCount() >= 2) remove(current);
      }
   }

   private class MouseMotionHandler implements MouseMotionListener
   {
      public void mouseMoved(MouseEvent event)
      {
         // Ustawienie kursora na krzyżyk, jeśli znajduje się wewnątrz
         // kwadratu.

         if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());
         else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
      }

      public void mouseDragged(MouseEvent event)
      {
         if (current != null)
         {
            int x = event.getX();
            int y = event.getY();

            // Przeciągnięcie aktualnego kwadratu w celu wycentrowania go w punkcie (x, y).
            current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH,
            ↪SIDELENGTH);
            repaint();
         }
      }
   }
}
```

API `java.awt.event.MouseEvent` **1.1**

- `int getX()`
- `int getY()`
- `Point getPoint()`

  Zwraca współrzędne `x` (pozioma) i `y` (pionowa) lub punkt, w którym miało miejsce zdarzenie, mierząc od lewego górnego rogu komponentu będącego źródłem zdarzenia.

- `int getClickCount()`

  Zwraca liczbę kolejnych kliknięć przyciskiem myszy związanych z danym zdarzeniem (odstęp czasu oddzielający zdarzenia określane jako kolejne zależy od systemu).

API `java.awt.event.InputEvent` **1.1**

- `int getModifiersEx()` **1.4**

  Zwraca rozszerzone modyfikatory zdarzenia. Do sprawdzania zwróconych wartości służą następujące maski:

  ```
  BUTTON1_DOWN_MASK
  BUTTON2_DOWN_MASK
  BUTTON3_DOWN_MASK
  SHIFT_DOWN_MASK
  CTRL_DOWN_MASK
  ALT_DOWN_MASK
  ALT_GRAPH_DOWN_MASK
  META_DOWN_MASK
  ```

- `static String getModifiersExText(int modifiers)` **1.4**

  Zwraca łańcuch typu `Shift+Button1` opisujący rozszerzone modyfikatory w danym zbiorze znaczników.

API `java.awt.Toolkit` **1.0**

- `public Cursor createCustomCursor(Image image, Point hotSpot, String name)` **1.2**

  Tworzy nowy obiekt niestandardowego kursora.

Parametry:		
	`image`	Obraz reprezentujący kursor
	`hotSpot`	Punkt centralny kursora (na przykład końcówka strzałki lub środek krzyżyka)
	`name`	Opis kursora wspomagający dostępność w specjalnych środowiskach

API `java.awt.Component` **1.0**

- `public void setCursor(Cursor cursor)` **1.1**

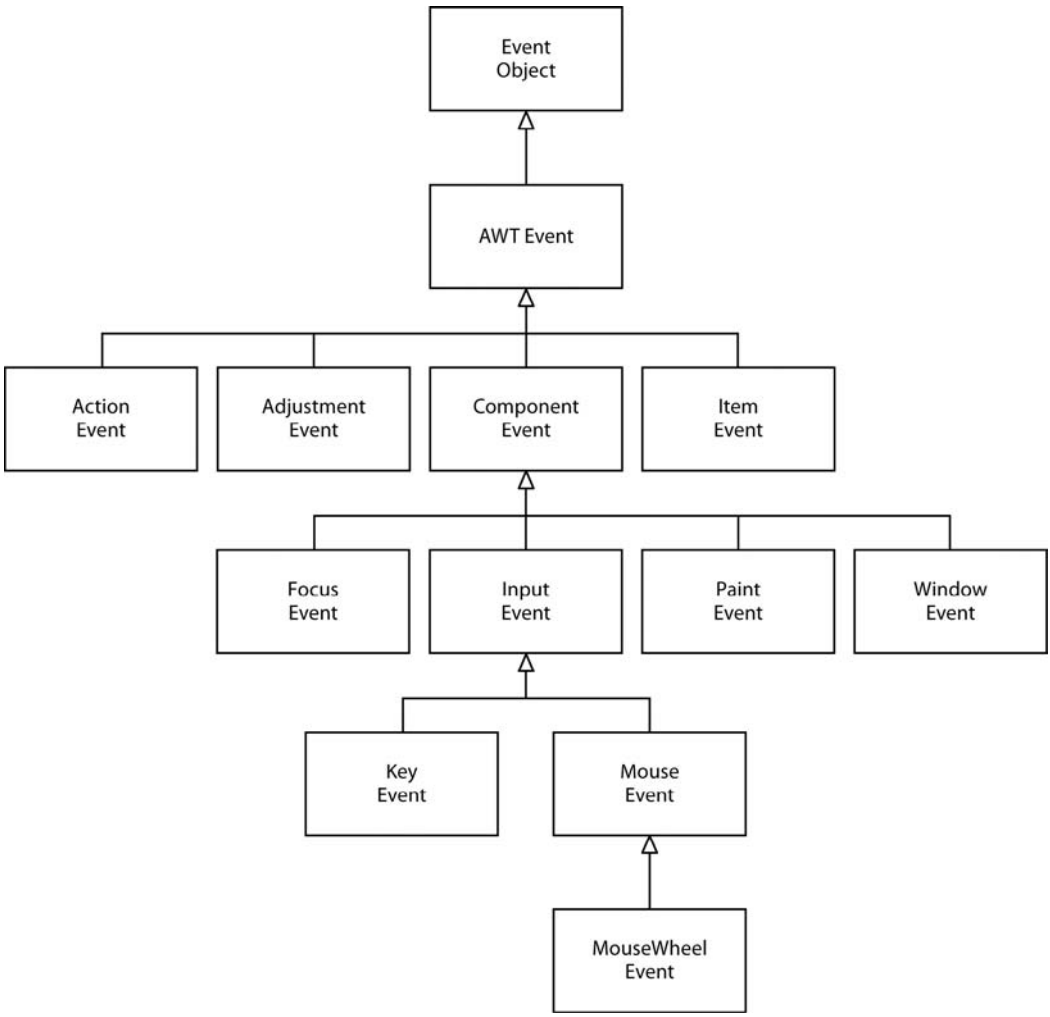
  Ustawia obraz kursora na określony kursor.

Hierarchia zdarzeń w bibliotece AWT

Mając już pewne rozeznanie w temacie obsługi zdarzeń, na zakończenie niniejszego rozdziału zrobimy krótki przegląd architektury obsługi zdarzeń biblioteki `AWT`.

Jak wspominaliśmy wcześniej, zdarzenia w Javie są obsługiwane w metodologii obiektowej, a wszystkie zdarzenia pochodzą od klasy `EventObject` z pakietu `java.util` (nazwą wspólnej nadklasy nie jest `Event`, ponieważ taką nazwę nosi klasa zdarzeń w starym modelu zdarzeń — mimo że model ten jest obecnie odradzany, jego klasy nadal wchodzą w skład biblioteki Javy).

Klasa `EventObject` posiada podklasę `AWTEvent` będącą nadklasą wszystkich klas zdarzeniowych `AWT`. Rysunek 8.8 przedstawia diagram dziedziczenia zdarzeń `AWT`.



Rysunek 8.8. *Diagram dziedziczenia klas zdarzeniowych AWT*

Niektóre komponenty Swing generują obiekty zdarzeniowe jeszcze innych typów zdarzeń. Rozszerzają one bezpośrednio klasę `EventObject`, a nie `AWTEvent`.

Obiekty zdarzeniowe zawierają informacje o zdarzeniach przesyłanych przez źródło zdarzeń do swoich słuchaczy. W razie potrzeby można przeanalizować obiekty zdarzeniowe, które zostały przekazane do obiektów nasłuchujących, co zrobiliśmy w przykładzie z przyciskiem za pomocą metod `getSource` i `getActionCommands`.

Niektóre klasy zdarzeniowe `AWT` są dla programisty Javy bezużyteczne. Na przykład biblioteka `AWT` wstawia do kolejki zdarzeń obiekty `PaintEvent`, ale obiekty te nie są dostarczane do słuchaczy. Programiści Javy nie nasłuchują zdarzeń rysowania. Przesłaniają oni metodę `paintComponent`, aby móc kontrolować ponowne rysowanie. Ponadto `AWT` generuje pewne zdarzenia, które są potrzebne tylko programistom systemowym. Nie opisujemy tych specjalnych typów zdarzeń.

Zdarzenia semantyczne i niskiego poziomu

Biblioteka `AWT` rozróżnia zdarzenia **niskiego poziomu** i zdarzenia **semantyczne**. Zdarzenie semantyczne jest dziełem użytkownika (jest to np. kliknięcie przycisku). Dlatego zdarzenie `ActionEvent` jest zdarzeniem semantycznym. Zdarzenia niskiego poziomu to takie zdarzenia, które umożliwiają zaistnienie zdarzeń semantycznych. W przypadku kliknięcia przycisku jest to jego naciśnięcie, szereg ruchów myszą i zwolnienie (ale tylko jeśli zwolnienie nastąpi w obrębie przycisku). Może to być naciśnięcie klawisza mające miejsce po wybraniu przycisku przez użytkownika za pomocą klawisza *Tab* i naciśnięcia go za pomocą spacji. Podobnie semantycznym zdarzeniem jest przesunięcie paska przewijania, a ruch myszą jest zdarzeniem niskiego poziomu.

Poniżej znajduje się lista najczęściej używanych klas zdarzeń semantycznych pakietu `java.awt.event`:

- `ActionEvent` — kliknięcie przycisku, wybór elementu z menu, wybór elementu listy, naciśnięcie klawisza *Enter* w polu tekstowym.
- `AdjustmentEvent` — przesunięcie paska przewijania.
- `ItemEvent` — wybór jednego z pól do wyboru lub elementów listy.

Do najczęściej używanych klas zdarzeń niskiego poziomu zaliczają się:

- `KeyEvent` — naciśnięcie lub zwolnienie klawisza.
- `MouseEvent` — naciśnięcie lub zwolnienie przycisku myszy, poruszenie lub przeciągnięcie myszą.
- `MouseWheelEvent` — pokręcenie kółkiem myszy.
- `FocusEvent` — uaktywnienie lub dezaktywacja elementu.
- `WindowEvent`— zmiana stanu okna.

Niniejszych zdarzeń nasłuchują następujące interfejsy:

```
ActionListener
AdjustmentListener
FocusListener
ItemListener
```

```
KeyListener
MouseListener
MouseMotionListener
MouseWheelListener
WindowListener
WindowFocusListener
WindowStateListener
```

Niektóre interfejsy nasłuchujące AWT, te zawierające więcej niż jedną metodę, posiadają odpowiadające im klasy adaptacyjne, które implementują wszystkie ich metody (pozostałe interfejsy mają tylko jedną metodę, a więc utworzenie dla nich klas adaptacyjnych nie dałoby żadnych korzyści). Poniższe klasy adaptacyjne są często używane:

```
FocusAdapter
KeyAdapter
MouseAdapter
MouseMotionAdapter
WindowAdapter
```

Tabela 8.4 przedstawia najważniejsze interfejsy nasłuchowe, zdarzenia i źródła zdarzeń biblioteki AWT.

Tabela 8.4. *Obsługa zdarzeń*

Interfejs	Metody	Parametry/ metody dostępu	Zdarzenia generowane przez
ActionListener	actionPerformed	ActionEvent getActionCommand getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent getAdjustable getAdjustmentType getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent getItem getItemSelectable getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent getKeyChar getKeyCode getKeyModifiersText getKeyText isActionText	Component

Tabela 8.4. *Obsługa zdarzeń (ciąg dalszy)*

Interfejs	Metody	Parametry / metody dostępu	Zdarzenia generowane przez
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent getClickCount getX getY getPoint translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	MouseWheelMoved	MouseWheelEvent getWheelRotation getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent getOldState getNewState	Window

Pakiet `javax.swing.event` zawiera dodatkowe zdarzenia specyficzne dla komponentów Swinga. Niektóre z nich opisujemy w następnym rozdziale.

Na tym zakończymy opis technik obsługi zdarzeń `AWT`. W następnym rozdziale nauczymy się wykorzystywać najpopularniejsze komponenty Swinga oraz szczegółowo przeanalizujemy generowane przez nie zdarzenia.