Java 2. Techniki zaawansowane

Autorzy: Cay S. Horstmann, Gary Cornell
Tłumaczenie: Jaromir Senczyk
ISBN: 83-7197-985-1
Tytuł oryginału: Core Java 2 Volume 2 Advanced Features
Format: B5, stron: 1122

Książka ta dostarcza doświadczonym programistom rozwiązań niezbędnych do pełnego wykorzystania możliwości Javy. To praktyczny przewodnik ułatwiający rozwiązywanie nawet najbardziej złożonych problemów programistycznych. Dodatkowo zawiera zupełnie nowy rozdział poświęcony wykorzystaniu języka XML w programach pisanych w Javie oraz zaktualizowane omówienie wielu zaawansowanych możliwości tej platformy – od kolekcji po metody macierzyste, od bezpieczeństwa po bibliotekę Swing.

Autorzy identyfikują problemy najczęściej napotykane przez doświadczonych programistów Javy i dostarczają przemyślanych rozwiązań zilustrowanych przykładami kodu, które uczyniły z tej książki prawdziwy bestseller. Dzięki niej ujrzysz w nowym świetle zagadnienia interfejsu ODBC™, tworzenia aplikacji sieciowych, wykorzystania zdalnych obiektów i wiele innych.

Najważniejsze informacje dla programistów Java:

- Zaktualizowane omówienie wielowątkowości, kolekcji i aplikacji sieciowych.
- Zmienione przedstawienie problematyki zdalnych obiektów.
- Nowe, zaawansowane techniki wykorzystania architektury komponentów JavaBeans™.
- Zaawansowane techniki tworzenia interfejsu użytkownika wykorzystujące biblioteki Swing i AWT.

Książka będzie dla Ciebie kolejnym krokiem w poznaniu możliwości Javy. Jest rozszerzeniem i doskonałym uzupełnieniem publikacji „Java 2. Postawy”.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

6

Zaawansowane możliwości pakietu Swing

W tym rozdziale:

- Listy.
- Drzewa.
- Tabele.
- Komponenty formatujące tekst.
- Organizatory komponentów.

W rozdziale tym kontynuować będziemy rozpoczęte w książce *Java 2. Podstawy* omówienie pakietu Swing wykorzystywanego do tworzenia interfejsu użytkownika. Pakiet Swing posiada bardzo rozbudowane możliwości, a w książce *Java 2. Podstawy* zdołaliśmy przedstawić jedynie komponenty najczęściej używane. Większość niniejszego rozdziału poświęcimy złożonym komponentom, takim jak listy, drzewa i tabele. Komponenty umożliwiające formatowanie tekstu, na przykład HTML, posiadają jeszcze bardziej złożoną implementację. Omówimy sposób ich praktycznego wykorzystania. Rozdział zakończymy przedstawieniem organizatorów komponentów, takich jak panele z zakładkami i panele z wewnętrznymi ramkami.

Listy

Prezentując użytkownikowi zbiór elementów do wyboru, możemy skorzystać z różnych komponentów. Jeśli zbiór ten zawierać będzie wiele elementów, to ich przedstawienie za pomocą pól wyboru zajmie zdecydowanie za dużo miejsca w oknie programu. Skorzystamy wtedy zwykle z listy bądź listy rozwijalnej. Listy rozwijalne są stosunkowo prostymi komponentami i dlatego omówiliśmy je już w książce *Java 2. Podstawy*. Natomiast listy reprezentowane przez komponent `JList` posiadają dużo większe możliwości, a sposób ich użycia przypomina korzystanie z innych złożonych komponentów, takich jak drzewa czy tabele. Dlatego właśnie od list rozpoczniemy omówienie złożonych komponentów pakietu Swing.

Listy często składają się z łańcuchów znaków, ale w praktyce zawierać mogą dowolne obiekty i kontrolować przy tym sposób ich prezentacji. Wewnętrzna architektura listy, która umożliwia taki stopień ogólności, prezentuje się dość elegancko. Niestety projektanci z firmy Sun postanowili pochwalić się elegancją tworzonych rozwiązań, zamiast ukryć ją przed programistami korzystającymi z komponentu. Skutkiem tego posługiwanie się listami w najprostszych przypadkach jest trochę skomplikowane, ponieważ programista musi manipulować mechanizmami, które umożliwiają wykorzystanie list w bardziej złożonych przypadkach. Omówienie rozpoczniemy od przedstawienia najprostszego i najczęściej spotykanego zastosowania tego komponentu — listy, której elementami są łańcuchy znaków. Później przejdziemy do bardziej złożonych przykładów ilustrujących uniwersalność komponentu.

Komponent JList

Zastosowanie komponentu `JList` przypomina użycie zbioru komponentów, takich jak przyciski lub pola wyboru. Różnica polega na tym, że elementy listy umieszczone są we wspólnej ramce, a wyboru dokonuje się, wskazując dany element, a nie związane z nim pole bądź przycisk. Użytkownik może też wybrać wiele elementów listy, jeśli pozwolimy na to.

Rysunek 6.1 pokazuje najprostszy przykład listy. Użytkownik może z niej wybrać atrybuty opisujące lisa, takie jak „quick", „brown", „hungry" i „wild" oraz, z braku innych pomysłów, „static", „private" i „final".

Rysunek 6.1.
Komponent klasy JList

Tworzenie listy rozpoczynamy od skonstruowania tablicy łańcuchów znaków, którą następnie przekazujemy konstruktorowi klasy `JList`:

```
String[] words = { "quick","brown","hungry","wild", . . .};
JList wordList = new JList(words);
```

Możemy wykorzystać w tym celu także anonimową tablicę:

```
JList wordList = new JList( new String[]
   { "quick","brown","hungry","wild", . . .});
```

Komponenty `JList` nie przewijają automatycznie swojej zawartości. W tym celu musimy umieścić listę w panelu przewijalnym:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

Panel ten, a nie listę, umieszczamy następnie na docelowym panelu okna.

Rozdzielenie prezentacji listy od mechanizmu przewijania jest z pewnością rozwiązaniem eleganckim, ale mało praktycznym. Właściwie prawie wszystkie listy wymagają przewijania. Zmuszanie programistów, by za każdym razem, gdy tworzą najprostszą listę, podziwiali działanie tego mechanizmu, jest okrutne.

Domyślnie lista mieści osiem elementów widocznych jednocześnie. Możemy to zmienić, korzystając z metody `setVisibleRowCount`:

```
wordList.setVisibleRowCount(10);  // pokazuje jednocześnie 10 elementów
```

Domyślnie użytkownik może wybierać wiele elementów listy. Wymaga to od niego zaawansowanego posługiwania się myszą: wybierając kolejne elementy, musi jednocześnie wciskać klawisz *Ctrl*. Aby wytypować pewien ciągły zakres elementów, użytkownik powinien zaznaczyć pierwszy z nich a następnie, wciskając klawisz *Shift*, wybrać ostatni z elementów.

Możliwość wyboru elementów przez użytkownika możemy ograniczyć, korzystając z metody `setSelectionMode`:

```
wordList.setSelectionMode
   (ListSelectionModel.SINGLE_SELECTION);
   // możliwość wyboru pojedynczego elementu
wordList.setSelectionMode
   (ListSelectionModel.SINGLE_INTERVAL_SELECTION);
   // możliwość wyboru pojedynczego elementu lub pojedynczego zakresu elementów
```

Z lektury książki *Java 2. Podstawy* przypominamy sobie z pewnością, że podstawowe elementy interfejsu użytkownika generują zdarzenia akcji w momencie ich aktywacji przez użytkownika. Listy wykorzystują jednak inny mechanizm powiadomień. Zamiast nasłuchiwać zdarzeń akcji, w przypadku list nasłuchiwać będziemy zdarzeń wyboru na liście. W tym celu musimy dodać do komponentu listy obiekt nasłuchujący wyboru i zaimplementować następującą metodę obiektu nasłuchującego:

```
public void valueChanged(ListSelectionEvent evt)
```

Podczas dokonywania wyboru na liście generuje się sporo zdarzeń. Załóżmy na przykład, że użytkownik przechodzi do kolejnego elementu listy. Gdy naciska klawisz myszy, generowane jest zdarzenie zmiany wyboru na liście. Zdarzenie to jest przejściowe i wywołanie metody

```
event.isAdjusting()
```

zwraca wartość `true`, gdy wybór nie jest ostateczny. Gdy użytkownik puszcza klawisz myszy, generowane jest kolejne zdarzenie, dla którego wywołanie metody `isAdjusting` zwróci tym razem wartość `false`. Jeśli nie jesteśmy zainteresowani przejściowymi zdarzeniami na liście, to wystarczy poczekać jedynie na zdarzenie, dla którego metoda `isAdjusting` zwróci właśnie wartość `false`. W przeciwnym razie musimy obsługiwać wszystkie zdarzenia.

Zwykle po zawiadomieniu o zdarzeniu będziemy chcieli się dowiedzieć, które elementy zostały wybrane. Metoda `getSelectedValues` zwraca *tablicę obiektów* zawierającą wybrane elementy.

Każdy z jej elementów musimy rzutować na łańcuch znaków.

```
Object[] values = list.getSelectedValues();
for(int i=0; i < values.length; i++)
   przetwarzanie elementów postaci (String)values[i];
```

> **Nie możemy rzutować tablicy** `Object[]` **zwróconej przez metodę** `getSelectedValues` **na tablicę** `String[]`. **Zwracana przez metodę tablica została utworzona nie jako tablica łańcuchów znaków, ale jako tablica obiektów, z których każdy jest akurat łańcuchem znaków. Jeśli chcemy przetwarzać zwróconą tablicę jako tablicę łańcuchów znaków, to powinniśmy skorzystać z poniższego kodu:**
>
> ```
> int length = values.length;
> String[] words = new String[length];
> System.arrayCopy(values, 0, words, 0, length);
> ```

Jeśli lista nie dopuszcza wyboru wielu elementów, to możemy skorzystać z metody `getSelectedValue`. Zwraca ona pierwszy element wybrany na liście.

```
String selection = (String)source.getSelectedValue();
```

> **Listy nie obsługują dwukrotnych kliknięć myszą. Projektanci pakietu Swing założyli, że na liście dokonuje się jedynie wyboru, a następnie zaznacza się komponent przycisku, aby wykonać jakąś akcję. Niektóre interfejsy użytkownika posiadają możliwość dwukrotnego kliknięcia elementu listy w celu dokonania jego wyboru i wykonania na nim pewnej akcji. Uważamy, że nie jest to zbyt dobry styl tworzenia interfejsu użytkownika, ponieważ wymaga, by użytkownik samodzielnie odkrył możliwość takiego jego działania. Jeśli jednak z pewnych powodów chcemy skorzystać z możliwości implementacji takiego zachowania listy, to musimy dodać do niej obiekt nasłuchujący mysz i obsługiwać zdarzenia myszy w następujący sposób:**
>
> ```
> public void mouseClicked(MouseEvent evt)
> {
>    if (evt.getClickCount()== 2)
>    {
>       JList source = (JList)evt.getSource();
>       Object[] selection = source.getSelectedValues();
>       doAction(selection);
>    }
> }
> ```

Listing 6.1 zawiera tekst źródłowy programu demonstrującego wykorzystanie listy wypełnionej łańcuchami znaków. Zwróćmy uwagę na sposób, w jaki metoda `valueChanged` tworzy łańcuch komunikatu z wybranych elementów listy.

Listing 6.1. *ListTest.java*

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

/**
   Program demonstrujący wykorzystanie listy zawierającej łańcuchy znaków.
*/
public class ListTest
```

```
{
   public static void main(String[] args)
   {
      JFrame frame = new ListFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca listę słów i etykietę pokazującą zdanie
   złożone z wybranych słów. Przytrzymując klawisz Ctrl, wybrać można wiele słów,
   a klawisz Shift pozwala na wybór całego zakresu słów.
*/
class ListFrame extends JFrame
{
   public ListFrame()
   {
      setTitle("ListTest");
      setSize(WIDTH, HEIGHT);

      String[] words =
      {
         "quick","brown","hungry","wild","silent",
         "huge","private","abstract","static","final"
      };

      wordList = new JList(words);
      JScrollPane scrollPane = new JScrollPane(wordList);

      JPanel p = new JPanel();
      p.add(scrollPane);
      wordList.addListSelectionListener(new
         ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent event)
            {
               Object[] values = wordList.getSelectedValues();

               StringBuffer text = new StringBuffer(prefix);
               for (int i = 0; i < values.length; i++)
               {
                  String word = (String)values[i];
                  text.append(word);
                  text.append(" ");
               }
               text.append(suffix);

               label.setText(text.toString());
            }
         });

      Container contentPane = getContentPane();
      contentPane.add(p, BorderLayout.SOUTH);
      label = new JLabel(prefix + suffix);
      contentPane.add(label, BorderLayout.CENTER);
   }
```

```
    private static final int WIDTH = 400;
    private static final int HEIGHT = 300;
    private JList wordList;
    private JLabel label;
    private String prefix = "The ";
    private String suffix = "fox jumps over the lazy dog.";
}
```

javax.swing.JList

- `JList(Object[] items)` tworzy listę wyświetlającą podane elementy.
- `void setVisibleRowCount(int c)` określa liczbę elementów widocznych jednocześnie na liście (bez przewijania).
- `void setSelectionMode(int mode)` określa możliwość wyboru pojedynczego lub wielu elementów.

  Parametry: `mode` jedna z wartości `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION`, `MULTIPLE_INTERVAL_SELECTION`.
- `void addListSelectionListener(ListSelectionListener listener)` dodaje do listy obiekt nasłuchujący zdarzeń wyboru na liście.
- `Object[] getSelectedValues()` zwraca elementy wybrane na liście lub pustą tablicę, jeśli żaden element nie został wybrany.
- `Object getSelectedValue()` zwraca pierwszy element wybrany na liście lub wartość `null`.

javax.swing.event.ListSelectionListener

- `void valueChanged(ListSelectionEvent e)` metoda wywoływana za każdym razem, gdy wybór na liście uległ zmianie.

Modele list

W poprzednim podrozdziale pokazaliśmy najczęściej spotykany sposób wykorzystania list polegający na:

- utworzeniu niezmiennego zbioru elementów listy (łańcuchów znaków),
- umożliwieniu przewijania listy,
- obsłudze zdarzeń wyboru elementów listy.

W dalszej części przedstawimy bardziej skomplikowane sposoby wykorzystania list, czyli:

- listy o bardzo dużej liczbie elementów,
- listy o zmiennej zawartości,
- listy zawierające elementy inne niż łańcuchy znaków.

W naszym pierwszym przykładzie utworzyliśmy listę zawierającą określony zbiór łańcuchów znaków. Często jednak zachodzi potrzeba dodawania nowych elementów listy bądź usuwania elementów już umieszczonych na liście. Zaskakujący może wydać się fakt, że klasa `JList` nie zawiera metod umożliwiających takie operacje na liście. Aby zrozumieć tego przyczynę, musimy zapoznać się bliżej z wewnętrzną architekturą komponentu listy. Podobnie jak w przypadku komponentów tekstowych także i lista jest przykładem zastosowania wzorca model-widok-nadzorca w celu oddzielenia wizualizacji listy (czyli kolumny elementów wyświetlonych w pewien sposób) od danych (kolekcji obiektów).

Klasa `JList` odpowiedzialna jest jedynie za wizualizację danych i niewiele wie na temat ich reprezentacji. Potrafi jedynie pobrać dane, korzystając z obiektu implementującego interfejs `ListModel`:

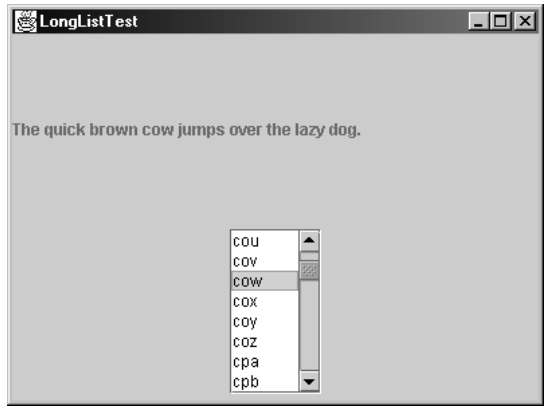
```
public interface ListModel
{
   public int getSize();
   public Object getElementAt(int i);
   public void addListDataListener(ListDataListener l);
   public void removeListDataListener(ListDataListener l);
}
```

Wykorzystując ten interfejs, komponent klasy `JList` może określić liczbę elementów i pobrać każdy z nich. Może także dodać się jako *obiekt nasłuchujący danych*. Dzięki temu będzie powiadamiany o każdej zmianie w kolekcji elementów i będzie mógł aktualizować reprezentację list na ekranie.

Jaki jest cel takiego rozwiązania? Dlaczego komponent `JList` nie przechowuje po prostu swoich elementów za pomocą wektora?

Zwróćmy uwagę, że interfejs nie określa sposobu przechowywania obiektów. W szczególności nie wymaga on nawet, by obiekty były przechowywane! Metoda `getElementAt` może wyznaczać wartość elementu od nowa, za każdym razem gdy jest wywoływana. Może okazać się to przydatne, jeśli lista prezentować ma bardzo liczną kolekcję danych, których nie chcemy przechowywać.

A oto najprostszy przykład: lista umożliwiać będzie użytkownikowi wybór spośród wszystkich możliwych kombinacji trzech liter (patrz rysunek 6.2).



Rysunek 6.2. *Wybór z listy zawierającej dużą liczbę elementów*

Istnieje 26 * 26 * 26 = 17 576 takich kombinacji. Zamiast przechowywać je wszystkie, program będzie tworzył je podczas przewijania listy przez użytkownika.

Implementacja programu okazuje się bardzo prosta. Zadanie dodania i usuwania odpowiednich obiektów nasłuchujących wykona za nas klasa `AbstractListModel`, którą rozszerzymy. Naszym zadaniem będzie jedynie dostarczenie implementacji metod `getSize` i `getElementAt`:

```
class WordListModel extends AbstractListModel
{
   public WordListModel(int n) { length = n; }
   public int getSize() { return (int)Math.pow(26, length); }
   public Object getElementAt(int n)
   {
      // wyznacza n-ty łańcuch
      . . .
   }
   . . .
}
```

Wyznaczenie *n*-tego łańcucha jest trochę skomplikowane — szczegóły znajdziemy w tekście programu umieszczonym w listingu 6.2.

Po utworzeniu modelu listy łatwo wykreować taką listę, która pozwoli użytkownikowi na przeglądanie wartości dostarczanych przez model:

```
wordList = new JList(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

Zaletą programu jest to, że prezentowane na liście łańcuchy nie są nigdzie przechowywane. Generowane są jedynie elementy listy widoczne w danym momencie dla użytkownika.

Musimy jeszcze dostarczyć do listy informację, że każdy z jej elementów posiada stałą szerokość i wysokość.

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```

W przeciwnym razie lista będzie wyznaczać te wartości dla każdego elementu, co będzie zbyt czasochłonne.

W praktyce listy zawierające tak dużą liczbę elementów są rzadko przydatne, ponieważ przeglądanie ich jest kłopotliwe dla użytkownika. Dlatego też uważamy, że projektanci list pakietu Swing przesadzili nieco z ich uniwersalnością. Liczba elementów, które użytkownik może wygodnie przeglądać na ekranie, jest tak mała, że mogłyby one być przechowywane po prostu wewnątrz komponentu listy. Oszczędziłoby to programistom tworzenia modeli list. Z drugiej jednak strony zastosowane rozwiązanie sprawia, że sposób wykorzystania komponentu `JList` jest spójny ze sposobami używania komponentów `JTree` i `JTable`, w przypadku których taka uniwersalność okazuje się przydatna.

Listing 6.2. *LongListTest.java*

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


/**
   Program demonstrujący wykorzystanie listy, która dynamicznie wyznacza swoje
elementy.
*/
public class LongListTest
{
   public static void main(String[] args)
   {
      JFrame frame = new LongListFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca długą listę słów i etykietę pokazującą zdanie
   złożone z wybranych słów.
*/
class LongListFrame extends JFrame
{
   public LongListFrame()
   {
      setTitle("LongListTest");
      setSize(WIDTH, HEIGHT);

      wordList = new JList(new WordListModel(3));
      wordList.setSelectionMode
         (ListSelectionModel.SINGLE_SELECTION);

      wordList.setFixedCellWidth(50);
      wordList.setFixedCellHeight(15);

      JScrollPane scrollPane = new JScrollPane(wordList);

      JPanel p = new JPanel();
      p.add(scrollPane);
      wordList.addListSelectionListener(new
         ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent evt)
            {
               StringBuffer word
                  = (StringBuffer)wordList.getSelectedValue();
               setSubject(word.toString());
            }

         });

      Container contentPane = getContentPane();
      contentPane.add(p, BorderLayout.SOUTH);
```

```
      label = new JLabel(prefix + suffix);
      contentPane.add(label, BorderLayout.CENTER);
      setSubject("fox");
   }

   /**
      Określa podmiot zdania pokazywanego za pomocą etykiety.
      @param word nowy podmiot zdania
   */
   public void setSubject(String word)
   {
      StringBuffer text = new StringBuffer(prefix);
      text.append(word);
      text.append(suffix);
      label.setText(text.toString());
   }

   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;
   private JList wordList;
   private JLabel label;
   private String prefix = "The quick brown ";
   private String suffix = " jumps over the lazy dog.";
}

/**
   Model listy dynamicznie generujący kombinacje n liter.
*/
class WordListModel extends AbstractListModel
{
   /**
      Tworzy model.
      @param n długość kombinacji (słowa)
   */
   public WordListModel(int n) { length = n; }

   public int getSize()
   {
      return (int)Math.pow(LAST - FIRST + 1, length);
   }

   public Object getElementAt(int n)
   {
      StringBuffer r = new StringBuffer();;
      for (int i = 0; i < length; i++)
      {
         char c = (char)(FIRST + n % (LAST - FIRST + 1));
         r.insert(0, c);
         n = n / (LAST - FIRST + 1);
      }
      return r;
   }

   private int length;
   public static final char FIRST = 'a';
   public static final char LAST = 'z';
}
```

javax.swing.JList

- `JList(ListModel dataModel)` tworzy listę, która wyświetla elementy dostarczane przez określony model.
- `void setFixedCellWidth(int width)` jeśli parametr `width` ma wartość większą od zera, to określa on szerokość każdej komórki listy. Wartością domyślną jest –1, co wymusza wyznaczenie rozmiarów każdej komórki z osobna.
- `void setFixedCellHeight(int height)` jeśli parametr `height` ma wartość większą od zera, to określa on wysokość każdej komórki listy. Wartością domyślną jest –1, co wymusza wyznaczenie rozmiarów każdej komórki z osobna.

javax.swing.ListModel

- `int getSize()` zwraca liczbę elementów w danym modelu.
- `Object getElementAt(int index)` zwraca element modelu.

Wstawianie i usuwanie

Kolekcji elementów listy nie możemy modyfikować bezpośrednio. Operacje te możemy wykonywać jedynie za pośrednictwem *modelu*. Załóżmy, że chcemy umieścić na liście kolejne elementy. Najpierw pobierzemy referencję odpowiedniego modelu:

```
ListModel model = list.getModel();
```

Jednak interfejs `ListModel` nie zawiera metod umożliwiających wstawianie i usuwanie elementów, ponieważ nie wymaga on przechowywania elementów listy.

Spróbujmy więc inaczej. Jeden z konstruktorów klasy `JList` korzysta z wektora obiektów:

```
Vector values = new Vector();
values.addElement("quick");
values.addElement("brown");
. . .
JList list = new JList(values);
```

Elementy wektora możemy następnie usuwać lub dodawać, ale oczywiście lista nie zarejestruje tych operacji i wobec tego nie będzie odzwierciedlać zmian w zbiorze elementów.

> **Nie istnieje konstruktor klasy `JList`, który posiadałby parametr klasy `ArrayList`. Jednak już konstrukcja komponentu listy z wykorzystaniem wektora jest rzadko przydatna i wobec tego brak ten nie stanowi istotnego ograniczenia.**

Rozwiązanie polega na utworzeniu modelu klasy `DefaultListModel`, wypełnieniu go początkowymi elementami i związaniu z listą.

```
DefaultListModel model = new DefaultListModel();
model.addElement"quick");
model.addElement"brown");
. . .
JList list = new JList(model);
```

Możemy teraz dodawać i usuwać elementy modelu, który będzie powiadamiać listę o tych zmianach.

```
model.removeElement"brown");
model.addElement"slow");
```

Jak łatwo zauważyć, klasa `DefaultModelList` stosuje nazwy metod odmienne niż klasy kolekcji.

Zastosowany model listy wykorzystuje wektor do przechowania danych. Model ten dziedziczy mechanizm powiadamiania listy z klasy `AbstractListModel`, podobnie jak nasza klasa w poprzednim podrozdziale.

Dostępne są konstruktory klasy `JList` tworzące listę w oparciu o tablicę lub wektor obiektów. Wydawać się może, że wykorzystują one model `DefaultListModel` w celu przechowania elementów listy. Nie jest to prawdą. Konstruktory te korzystają z uproszczonego modelu, który umożliwia jedynie dostęp do elementów, ale nie posiada mechanizmu powiadamiania o zmianach. Poniżej prezentujemy kod konstruktora klasy `JList` tworzącego listę na podstawie wektora:

```
public JList(final Vector listData)
{
   this (new AbstractListModel()
   {
      public int getSize() { return listData.size(); }
      public Object getEelementAt(int i)
      { return listData.elementAt(i); }
   });
}
```

Pokazuje on, że jeśli zmienimy zawartość wektora po utworzeniu listy, to pokazywać ona będzie mylącą mieszankę starych i nowych elementów, aż do momentu gdy cała lista zostanie odrysowana. (Słowo kluczowe `final` w deklaracji konstruktora nie zabrania wprowadzania zmian zawartości wektora w innych fragmentach kodu. Oznacza ono jedynie, że konstruktor nie modyfikuje referencji `listData`. Słowo kluczowe `final` jest wymagane w tej deklaracji, ponieważ obiekt `listData` wykorzystywany jest przez klasę wewnętrzną).

javax.swing.JList

- `ListModel getModel()` pobiera model listy.

javax.swing.DefaultListModel

- `void addElement(Object obj)` umieszcza obiekt na końcu danych modelu.
- `boolean removeElement(Object obj)` usuwa pierwsze wystąpienie obiektu z modelu. Zwraca wartość `true`, jeśli obiekt został odnaleziony w modelu, wartość `false` — w przeciwnym razie.

Odrysowywanie zawartości listy

Jak dotąd wszystkie przykłady wykorzystywały listy zawierające łańcuchy znaków. W praktyce równie łatwo możemy utworzyć listę ikon, dostarczając konstruktorowi tablicę lub wektor wypełniony obiektami klasy `Icon`. W ogóle możemy reprezentować elementy listy za pomocą dowolnych rysunków.

Klasa `JList` automatycznie wyświetla łańcuchy znaków oraz ikony, ale w pozostałych przypadkach należy dostarczyć jej *obiekt odrysowujący zawartość komórek* listy. Obiekt taki musi należeć do klasy, która implementuje poniższy interfejs:

```
interface ListCellRenderer
{
   Component getListCellRendererComponent(JList list,
      Object value, int index,
      boolean isSelected, boolean cellHasFocus);
}
```

Jeśli liście dostarczymy taki obiekt, to jego metoda wywoływana będzie dla każdego elementu listy, aby:

- ustalić jego wymiary (w przypadku gdy nie wybraliśmy listy o stałych wymiarach komórek),
- narysować go.

Obiekt ten musi tworzyć i zwracać obiekt typu `Component`, którego metody `getPreferredSize` oraz `paintComponent` wykonają odpowiednie operacje wymagane dla prezentacji elementów listy.

Najprostszym sposobem spełnienia tego wymagania jest wykorzystanie klasy wewnętrznej dysponującej tymi metodami:

```
class MyCellRenderer implements ListCellRenderer
{
   public Component getListCellRendererComponent(final JList list,
      final Object value, final int index,
      final boolean isSelected, final boolean cellHasFocus)
   {
      return new
         JPanel()
         {
            public void paintComponent(Graphics g)
            {
               // kod rysujący element
            }
            public Dimension getPreferredSize()
            {
                // kod wyznaczający rozmiary
            }
         };
   }
}
```

Program, którego kod źródłowy zawiera listing 6.3, umożliwia wybór czcionki, korzystając z jej rzeczywistego przedstawienia na liście (patrz rysunek 6.3). Metoda `paintComponent` wyświetla nazwę danej czcionki, wykorzystując ją samą. Musi także dopasować kolorystykę do aktualnego wyglądu klasy `JList`. Uzyskujemy ją, posługując się metodami `getForeground/getBackground` oraz `getSelectionForeground/getSelectionBackground` klasy `JList`. Metoda `getPreferredSize` mierzy łańcuch znaków w sposób opisany w książce *Java 2. Podstawy* w rozdziale 7.

Rysunek 6.3. *Lista o komórkach rysowanych przez program*

Obiekt rysujący komórki instalujemy za pomocą metody `setCellRenderer`:

```
fontList.setCellRenderer(new FontCellRenderer());
```

Od tego momentu wszystkie komórki listy będą rysowane przez ten obiekt.

W wielu przypadkach sprawdza się prostsza metoda tworzenia obiektów rysujących komórki list. Jeśli komórka składa się z tekstu, ikony i zmienia swój kolor, to wszystkie te możliwości uzyskać możemy, korzystając z obiektu klasy `JLabel`. Aby na przykład pokazać nazwę czcionki za jej pomocą, możemy skorzystać z następującego obiektu:

```
class FontCellRenderer implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list,
      Object value, int index, boolean isSelected,
         boolean cellHasFocus)
   {
      JLabel label = new JLabel();
      Font font = (Font)value;
      label.setText(font.getFamily());
      label.setFont(font);
      label.setOpaque(true);
      label.setBackground(isSelected
         ? list.getSelectionBackground()
         : list.getBackground());
      label.setForeground(isSelected
         ? list.getSelectionForeground()
         : list.getForeground());
      return label;
   }
}
```

Zwróćmy uwagę, że w tym przypadku nie implementujemy wcale metod `paintComponent` oraz `getPreferredSize`. Implementacje tych metod posiada bowiem klasa `JLabel`. Nasze zadanie polega jedynie na skonfigurowaniu tekstu, czcionki i koloru etykiety klasy `JLabel` zgodnie z naszymi wymaganiami.

Klasa `FontCellRenderer` może być nawet klasą pochodną klasy `JLabel` i sama konfigurować swoje parametry w wywołaniu metody `getListCellRendererComponent`, a następnie zwracać wartość `this`:

```
class FontCellRenderer extends JLabel implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list,
      Object value, int index, boolean isSelected,
         boolean cellHasFocus)
   {
      Font font = (Font)value;
      setText(font.getFamily());
      setFont(font);
      setOpaque(true);
      setBackground(isSelected
         ? list.getSelectionBackground()
         : list.getBackground());
      setForeground(isSelected
         ? list.getSelectionForeground()
         : list.getForeground());
      return this;
   }
}
```

Kod taki stanowi wygodny skrót w sytuacjach, w których istnieje komponent (`JLabel` w naszym przypadku) o funkcjonalności wystarczającej do narysowania zawartości komórki listy.

Listing 6.3. *ListRenderingTest.java*

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

/**
   Program demonstrujący wykorzystanie obiektów
   rysujących komórki listy.
*/
public class ListRenderingTest
{
   public static void main(String[] args)
   {
      JFrame frame = new ListRenderingFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca listę czcionek i pole tekstowe, którego tekst
   pokazywany jest wybraną z listy czcionką.
```

```
*/
class ListRenderingFrame extends JFrame
{
   public ListRenderingFrame()
   {
      setTitle("ListRenderingTest");
      setSize(WIDTH, HEIGHT);

      ArrayList fonts = new ArrayList();
      final int SIZE = 24;
      fonts.add(new Font("Serif", Font.PLAIN, SIZE));
      fonts.add(new Font("SansSerif", Font.PLAIN, SIZE));
      fonts.add(new Font("Monospaced", Font.PLAIN, SIZE));
      fonts.add(new Font("Dialog", Font.PLAIN, SIZE));
      fonts.add(new Font("DialogInput", Font.PLAIN, SIZE));
      fontList = new JList(fonts.toArray());
      fontList.setVisibleRowCount(4);
      fontList.setSelectionMode
         (ListSelectionModel.SINGLE_SELECTION);
      fontList.setCellRenderer(new FontCellRenderer());
      JScrollPane scrollPane = new JScrollPane(fontList);

      JPanel p = new JPanel();
      p.add(scrollPane);
      fontList.addListSelectionListener(new
         ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent evt)
            {
               Font font = (Font)fontList.getSelectedValue();
               text.setFont(font);
            }

         });

      Container contentPane = getContentPane();
      contentPane.add(p, BorderLayout.SOUTH);
      text = new JTextArea(
         "The quick brown fox jumps over the lazy dog");
      text.setFont((Font)fonts.get(0));
      text.setLineWrap(true);
      text.setWrapStyleWord(true);
      contentPane.add(text, BorderLayout.CENTER);
   }

   private JTextArea text;
   private JList fontList;
   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;
}

/**
   Obiekt rysujący komórki listy przy użyciu czcionki, której nazwę zawiera komórka.
*/
class FontCellRenderer implements ListCellRenderer
{
```

```
   public Component getListCellRendererComponent
      (final JList list, final Object value,
      final int index, final boolean isSelected,
      final boolean cellHasFocus)
   {
      return new
         JPanel()
         {
            public void paintComponent(Graphics g)
            {
               Font font = (Font)value;
               String text = font.getFamily();
               FontMetrics fm = g.getFontMetrics(font);
               g.setColor(isSelected
                  ? list.getSelectionBackground()
                  : list.getBackground());
               g.fillRect(0, 0, getWidth(), getHeight());
               g.setColor(isSelected
                  ? list.getSelectionForeground()
                  : list.getForeground());
               g.setFont(font);
               g.drawString(text, 0, fm.getAscent());
            }

            public Dimension getPreferredSize()
            {
               Font font = (Font)value;
               String text = font.getFamily();
               Graphics g = getGraphics();
               FontMetrics fm = g.getFontMetrics(font);
               return new Dimension(fm.stringWidth(text),
                  fm.getHeight());
            }
         };
   }
}
```

javax.swing.JList

- `Color getBackground()` **zwraca kolor tła komórki listy, która nie jest wybrana.**
- `Color getSelectionBackground()` **zwraca kolor tła komórki listy, która została wybrana.**
- `void setCellRenderer(ListCellRenderer cellRenderer)` **instaluje obiekt wykorzystywany do rysowania zawartości komórek listy.**

javax.swing.ListCellRenderer

- `Component getListCellRendererComponent(JList list, Object item, int index, boolean isSelected, boolean hasFocus)` **zwraca komponent, którego metoda** `paint` **rysuje zawartość komórek. Jeśli komórki listy nie posiadają stałych rozmiarów, to komponent ten musi także implementować metodę** `getPreferredSize`.

Parametry:	`list`	lista, której komórki mają zostać narysowane,
	`item`	rysowany element,
	`index`	indeks elementu w modelu,
	`isSelected`	wartość `true`, jeśli komórka jest wybrana,
	`hasFocus`	wartość `true`, jeśli komórka jest bieżąca.

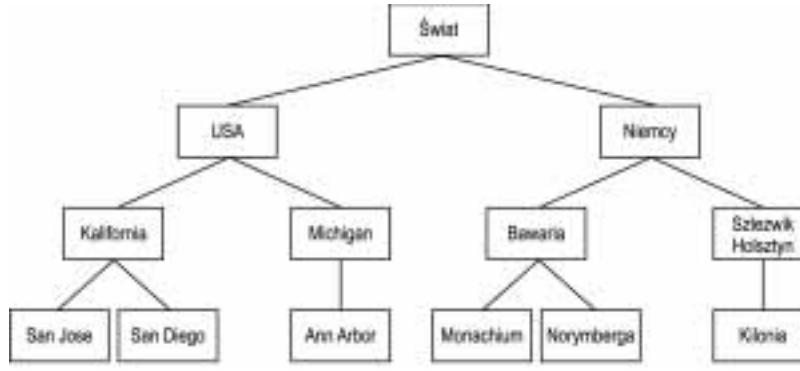
Drzewa

Każdy użytkownik komputera, którego system operacyjny posiada hierarchicznie zbudowany system plików, spotkał się w praktyce z jego reprezentacją za pomocą *drzewa*, taką jak na przykład na rysunku 6.4. Katalogi i pliki tworzą tylko jedną z wielu możliwych struktur drzewiastych. Programiści stosują drzewa również do opisu hierarchii dziedziczenia klas. Także w życiu codziennym często spotykamy struktury drzewiaste, takie jak hierarchie administracyjne państw, stanów, miast itd. (patrz rysunek 6.5).

Rysunek 6.4.
Drzewo katalogów

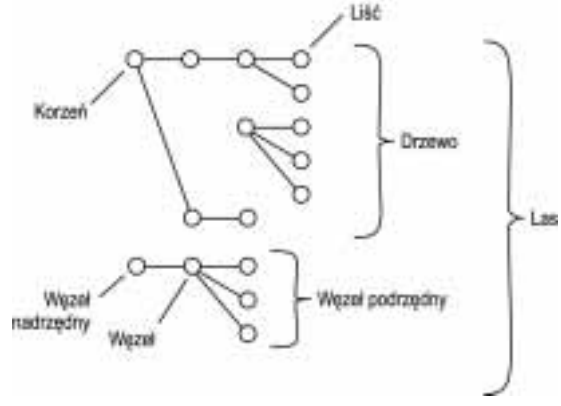
W programach często trzeba zaprezentować dane w postaci struktury drzewiastej. Biblioteka Swing dostarcza w tym celu klasę `JTree`. Klasa `JTree` (wraz z klasami pomocniczymi) służy do tworzenia reprezentacji graficznej drzewa i przetwarzania akcji użytkownika polegających na rozwijaniu i zwijaniu węzłów drzewa. W podrozdziale tym nauczymy się korzystać z możliwości klasy `JTree`. Podobnie jak w przypadku innych złożonych komponentów biblioteki Swing, skoncentrujemy się na omówieniu najczęstszych i najbardziej przydatnych przypadków zastosowań klasy `JTree`. Jeśli spotkasz się z nietypowym zastosowaniem drzewa, to polecamy książkę *Core Java Foundation Classes* autorstwa Kim Topley (Prentice-Hall, 1998) lub *Graphic Java 2* napisaną przez Davida M. Geary'ego (Prentice-Hall, 1999).

Rysunek 6.5.
Hierarchia państw, stanów i miast



Zanim przejdziemy do konkretów, warto usystematyzować terminologię związaną z opisem drzew (patrz rysunek 6.6). Drzewo składa się z *węzłów*. Każdy węzeł jest albo *liściem*, albo posiada *węzły podrzędne*. Każdy węzeł drzewa z wyjątkiem jego *korzenia*, posiada dokładnie jeden *węzeł nadrzędny*. Każde drzewo posiada jeden korzeń. Czasami występuje kolekcja drzew, z których każde posiada własny korzeń. Nazywamy ją *lasem*.

Rysunek 6.6.
Terminologia drzew



Najprostsze drzewa

Pierwszy przykładowy program wyświetlać będzie drzewo o kilku węzłach (patrz rysunek 6.8). Podobnie jak inne komponenty biblioteki Swing, także i klasa `JTree` jest przykładem zastosowania wzorca model-widok-nadzorca. W praktyce oznacza to, że komponentowi interfejsu użytkownika musimy dostarczyć model danych. W przypadku klasy `JTree` będzie to parametr konstruktora:

```
TreeModel model = . . . ;
JTree tree = new JTree(model);
```

Dostępne są także konstruktory tworzące drzewo na podstawie kolekcji elementów:

```
JTree(Object[] nodes)
JTree(Vector nodes)
JTree(Hashtable nodes)  // wartości tablicy stają się węzłami drzewa
```

Konstruktory te są jednak mało przydatne, gdyż tworzą las drzew, z których każde posiada jeden węzeł. Ostatni z konstruktorów jest wyjątkowo nieprzydatny, ponieważ węzły umieszczane są w drzewie w praktycznie przypadkowy sposób określony przez kody mieszające elementów.

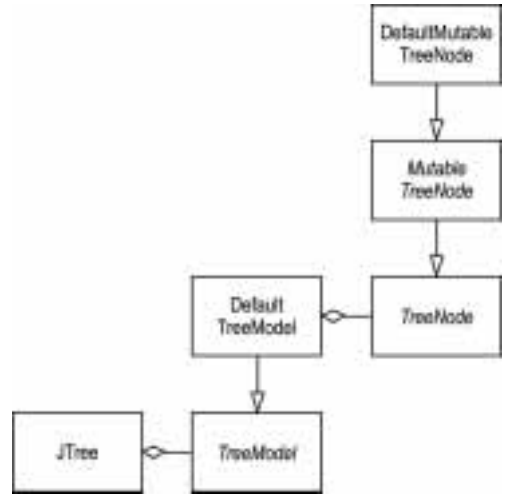
Model drzewa tworzymy, definiując klasę implementującą interfejs `TreeModel`. Z możliwości tej skorzystamy w dalszej części rozdziału, a na początku użyjemy klasy `DefaultTreeModel` dostarczanej przez bibliotekę Swing.

Kreując model tej klasy, musimy dostarczyć mu korzeń.

```
TreeNode root = . . . ;
DefaultTreeModel model = new DefaultTreeModel(root);
```

`TreeNode` jest kolejnym z interfejsów związanych z drzewami. Drzewo powstaje z węzłów dowolnych klas implementujących interfejs `TreeNode`. Na razie wykorzystamy w tym celu konkretną klasę `DefaultMutableTreeNode` udostępnianą przez bibliotekę Swing. Klasa ta implementuje interfejs `MutableTreeNode` będący specjalizacją interfejsu `TreeNode` (patrz rysunek 6.7).

Rysunek 6.7. *Zależności pomiędzy klasami drzewa*



Węzeł klasy `DefaultMutableTreeNode` przechowuje obiekt zwany *obiektem użytkownika*. Drzewo rysuje reprezentację obiektów użytkownika dla wszystkich węzłów. Dopóki nie zostanie zainstalowany specjalizowany obiekt rysujący węzły, to drzewo wyświetla po prostu łańcuch znaków będący wynikiem wywołania metody `toString`.

Nasz pierwszy przykład wykorzystywać będzie łańcuchy znaków jako obiekty użytkownika. W praktyce drzewo tworzą zwykle bardziej złożone obiekty, na przykład drzewo reprezentujące system plików składać się może z obiektów klasy `File`.

Obiekt użytkownika możemy przekazać jako parametr konstruktora węzła bądź później za pomocą metody `setUserObject`.

```
DefaultMutableTreeNode node
   = new DefaultMutableTreeNode(" Texas ")
node.setUserObject(" California ");
```

Następnie musimy utworzyć powiązania pomiędzy węzłami nadrzędnymi i podrzędnymi. Konstrukcję drzewa rozpoczniemy od korzenia, a później dodamy do niego węzły podrzędne:

```
DefaultMutableTreeNode root
   = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country
   = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state
   = new DefaultMutableTreeNode("California");
country.add(state);
```

Rysunek 6.8 pokazuje drzewo utworzone przez program.

Rysunek 6.8. *Najprostsze drzewo*

Po skonstruowaniu i połączeniu wszystkich węzłów możemy utworzyć model drzewa, przekazując mu korzeń, a następnie wykorzystać model do opracowania komponentu `JTree`.

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree = new JTree(treeModel);
```

Możemy nawet przekazać korzeń bezpośrednio konstruktorowi klasy `JTree`, który w takim przypadku sam utworzy model drzewa:

```
JTree = new JTree(root);
```

Listing 6.4 zawiera kompletny tekst źródłowy programu.

Listing 6.4. *SimpleTree.java*

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

/**
   Program wyświetlający najprostsze drzewo.
*/
public class SimpleTree
{
   public static void main(String[] args)
   {
      JFrame frame = new SimpleTreeFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}
```

```
/**
   Ramka zawierająca drzewo wyświetlające dane
   umieszczone przez program w modelu.
*/
class SimpleTreeFrame extends JFrame
{
   public SimpleTreeFrame()
   {
      setTitle("SimpleTree");
      setSize(WIDTH, HEIGHT);

      // buduje model drzewa

      DefaultMutableTreeNode root
         = new DefaultMutableTreeNode("World");
      DefaultMutableTreeNode country
         = new DefaultMutableTreeNode("USA");
      root.add(country);
      DefaultMutableTreeNode state
         = new DefaultMutableTreeNode("California");
      country.add(state);
      DefaultMutableTreeNode city
         = new DefaultMutableTreeNode("San Jose");
      state.add(city);
      city = new DefaultMutableTreeNode("Cupertino");
      state.add(city);
      state = new DefaultMutableTreeNode("Michigan");
      country.add(state);
      city = new DefaultMutableTreeNode("Ann Arbor");
      state.add(city);
      country = new DefaultMutableTreeNode("Germany");
      root.add(country);
      state = new DefaultMutableTreeNode("Schleswig-Holstein");
      country.add(state);
      city = new DefaultMutableTreeNode("Kiel");
      state.add(city);

      // tworzy drzewo i umieszcza je w przewijalnym panelu

      JTree tree = new JTree(root);
      Container contentPane = getContentPane();
      contentPane.add(new JScrollPane(tree));
   }

   private static final int WIDTH = 300;
   private static final int HEIGHT = 200;
}
```

Po uruchomieniu programu powstanie drzewo, które zobaczymy na rysunku 6.9. Widoczny będzie jedynie korzeń drzewa oraz jego węzły podrzędne. Wybranie myszą uchwytu węzła spowoduje rozwinięcie poddrzewa. Odcinki wystające z ikony uchwytu węzła skierowane są w prawo, gdy poddrzewo jest zwinięte i w dół w przeciwnym razie (patrz rysunek 6.10). Nie wiemy, co mieli na myśli projektanci wyglądu interfejsu zwanego Metal, tworząc ikonę uchwytu węzła, ale nam przypomina ona w działaniu klamkę drzwi. Kierujemy ją w dół, aby rozwinąć poddrzewo.

Rysunek 6.9.
Początkowy wygląd drzewa

Rysunek 6.10.
Zwinięte i rozwinięte poddrzewa

Wygląd drzewa zależy od wybranego wyglądu komponentów interfejsu użytkownika. Opisując dotąd drzewo, korzystaliśmy ze standardowego dla aplikacji Java wyglądu interfejsu Metal. W przypadku interfejsów Motif lub Windows uchwyty węzłów posiadają postać kwadratów zawierających znaki plus lub minus (patrz rysunek 6.11).

Rysunek 6.11.
Drzewo o wyglądzie Windows

Do wersji SDK 1.3 włącznie linie łączące węzły drzewa domyślnie nie były rysowane (patrz rysunek 6.12). Począwszy od SKD 1.4, rysowane są domyślnie.

Rysunek 6.12.
Drzewo bez linii łączących węzły

Korzystając z SDK 1.4., możemy wyłączyć rysowanie linii łączących węzły:

```
tree.putClientProperty("JTree.lineStyle", "None");
```

lub włączyć je z powrotem:

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

Istnieje także styl linii `Horizontal` pokazany na rysunku 6.13. Poziome linie oddzielają wtedy węzły podrzędne korzenia. Nie jesteśmy przekonani o celowości takiego rozwiązania.

Rysunek 6.13. *Drzewo wykorzystujące styl linii Horizontal*

Domyślnie korzeń drzewa nie posiada uchwytu, który umożliwiałby zwinięcie całego drzewa. Możemy go dodać następująco:

```
tree.setShowsRootHandles(true);
```

Rysunek 6.14 pokazuje rezultat. Możemy teraz zwinąć całe drzewo do korzenia.

Rysunek 6.14. *Drzewo posiadające uchwyt korzenia*

Możemy także w ogóle usunąć reprezentację korzenia drzewa, uzyskując w ten sposób *las* drzew, z których każde posiada własny korzeń. Tworząc taki las, nadal musimy jednak połączyć wszystkie drzewa wspólnym korzeniem, który następnie ukrywamy, wywołując

```
tree.setRootVisible(false);
```

Efekt przedstawia rysunek 6.15. Występują na nim dwa korzenie oznaczone „USA" i „Germany". W rzeczywistości są to węzły posiadające wspólny, ukryty korzeń.

Rysunek 6.15. *Las*

Zajmijmy się teraz liśćmi drzewa. Zwróćmy uwagę, że ikony liści różnią się od ikon pozostałych węzłów drzewa (patrz rysunek 6.16).

Rysunek 6.16.
Ikony liści

Każdy węzeł drzewa reprezentowany jest za pomocą pewnej ikony. Wyróżnić można trzy rodzaje takich ikon: ikona liścia, ikona węzła, którego poddrzewo jest rozwinięte i ikona węzła, którego poddrzewo jest zwinięte. Dla uproszczenia dwa ostatnie rodzaje ikon będziemy nazywać wspólnie ikonami folderu.

Obiekt rysujący węzły drzewa musi otrzymać informację, jakiej ikony powinien użyć dla danego węzła. Domyślny proces decyzyjny przebiega następująco: jeśli metoda `isLeaf` zwraca wartość `true`, to rysowana jest ikona liścia. W przeciwnym razie używana jest ikona folderu.

Metoda `isLeaf` klasy `DefaultMutableTreeNode` zwraca wartość `true` dla każdego węzła, który nie posiada węzłów podrzędnych. W ten sposób węzły posiadające węzły podrzędne otrzymują ikonę folderu, a pozostałe węzły — ikonę liścia.

Czasami rozwiązanie takie nie jest jednak właściwe. Załóżmy, że do naszego drzewa dodaliśmy węzeł reprezentujący stan Montana i dopiero zastanawiamy się, jakie miasta powinniśmy umieścić jako jego węzły podrzędne. Nie chcemy przy tym, aby węzeł reprezentujący stan posiadał ikonę liścia, ponieważ koncepcyjnie przysługuje ona tylko miastom.

Klasa `JTree` nie wie, które węzły powinny być liśćmi. Decyduje o tym model drzewa. Jeśli węzeł, który nie posiada węzłów podrzędnych, nie jest liściem z punktu widzenia koncepcji drzewa, to model drzewa może zastosować inne kryterium rozpoznawania liści. Polega ono na wykorzystaniu właściwości węzła zezwalającej na posiadanie węzłów podrzędnych.

Dla węzłów, które nie będą posiadać węzłów podrzędnych, należy wtedy wywołać:

```
node.setAllowsChildren(false);
```

oraz poinformować model drzewa, by, decydując czy danym węzeł jest liściem, sprawdzał, czy może on posiadać węzły podrzędne. W tym celu wywołujemy metodę `setAsksAllowsChildren` klasy `DefaultTreeModel`:

```
model. setAsksAllowsChildren(true);
```

Od tego momentu węzły, które mogą posiadać węzły podrzędne, otrzymują ikonę folderu, a pozostałe ikonę liścia.

Takie kryterium doboru ikony możemy także uzyskać, tworząc obiekt klasy `JTree` za pomocą odpowiedniego konstruktora:

```
JTree tree = new JTree(root, true);
   // węzły, które nie mogą posiadać węzłów podrzędnych otrzymają ikonę liścia
```

javax.swing.JTree

- `JTree(TreeModel model)` tworzy drzewo na podstawie modelu.
- `JTree(TreeNode root)`
- `JTree(TreeNode root, boolean asksAllowChildren)`

  Tworzą drzewo, korzystając z domyślnego modelu i wyświetlając początkowo korzeń i jego węzły podrzędne.

  Parametry: `root` korzeń,

  `asksAllowChildren`, jeśli posiada wartość `true`, to węzeł jest liściem, gdy może posiadać węzły podrzędne.

- `void setShowsRootHandles(boolean b)`, jeśli `b` posiada wartość `true`, to korzeń posiada uchwyt umożliwiający zwinięcie drzewa.
- `void setRootVisible(boolean b)`, jeśli `b` posiada wartość `true`, to korzeń jest wyświetlany. W przeciwnym razie jest ukryty.

javax.swing.tree.TreeNode

- `boolean isLeaf()` zwraca wartość `true`, jeśli dany węzeł reprezentuje liść na poziomie koncepcji.
- `boolean getAllowsChildren()` zwraca wartość `true`, jeśli dany węzeł może posiadać węzły podrzędne.

javax.swing.tree.MutableTreeNode

- `void setUserObject(Object userObject)` określa obiekt użytkownika dla danego węzła.

javax.swing.tree.TreeModel

- `boolean isLeaf(Object node)` zwraca wartość `true`, jeśli węzeł `node` zostanie wyświetlony jako liść.

javax.swing.tree.DefaultTreeModel

- `void setAsksAllowsChildren(boolean b)`, jeśli `b` posiada wartość `true`, to węzły wyświetlane są jako liście, w przypadku gdy metoda `getAllowsChildren` zwraca wartość `false`. Gdy `b` posiada wartość `false`, to węzły wyświetlane są jako liście, jeśli metoda `isLeaf` zwraca wartość `true`.

javax.swing.tree.DefaultMutableTreeNode

- `DefaultMutableTreeNode(Object userObject)` tworzy węzeł drzewa zawierający podany obiekt użytkownika.

- `void add(MutableTreeNode child)` dodaje do węzła węzeł podrzędny.
- `void setAllowsChioldren(boolean b)`, jeśli `b` posiada wartość `true`, to do węzła mogą być dodawane węzły podrzędne.

javax.swing.JComponent

- `void putClientProperty(Object key, Object value)` dodaje parę `key/value` do niewielkiej tablicy, którą zarządza każdy komponent. Mechanizm ten jest stosowany przez niektóre komponenty Swing w celu przechowywania specyficznych właściwości związanych z wyglądem komponentu.

Modyfikacje drzew i ścieżki drzew

Następny przykład programu ilustrować będzie sposób modyfikacji drzew. Rysunek 6.17 przedstawia potrzebny interfejs użytkownika. Jeśli wybierzemy przycisk *Add Sibling* lub *Add Child*, to program doda do drzewa nowy węzeł opisany jako *New*. Jeśli wybierzemy przycisk *Delete*, to usuniemy wybrany węzeł.

Rysunek 6.17.
Modyfikowanie drzewa

Aby zaimplementować takie zachowanie, musimy uzyskać informację o tym, który z węzłów drzewa jest aktualnie wybrany. Klasa `JTree` zaskoczy nas z pewnością sposobem identyfikacji węzłów w drzewie. Wykorzystuje ona *ścieżki do obiektów* nazywane *ścieżkami drzewa*. Ścieżka taka zaczyna się zawsze od korzenia i zawiera sekwencje węzłów podrzędnych (patrz rysunek 6.18).

Rysunek 6.18.
Ścieżka drzewa

Zastanawiać może, w jakim celu klasa `JTree` potrzebuje całej ścieżki. Czy nie wystarczyłby jej obiekt klasy `TreeNode` i możliwość wywołania metody `getParent`? Okazuje się, że klasa `JTree` nie ma pojęcia o istnieniu interfejsu `TreeNode`. Nie jest on wykorzystywany przez interfejs `TreeModel`, a dopiero przez implementującą go klasę `DefaultTreeModel`. Możemy tworzyć też inne modele drzewa, które nie będą wykorzystywać interfejsu `TreeNode`. Więc

zdarza się, że obiekty w takim modelu nie będą posiadać metod `getParent` i `getChild`, ale wykorzystywać będą inny sposób połączeń pomiędzy węzłami. Łączenie węzłów jest zadaniem modelu drzewa. Klasa `JTree` nie posiada żadnej wiedzy na temat natury tych połączeń. I to jest właśnie przyczyną, dla której klasa `JTree` musi wykorzystywać kompletne ścieżki drzewa.

Klasa `TreePath` zarządza sekwencją referencji do obiektów klasy `Object` (nie `TreeNode`!). Wiele metod klasy `JTree` zwraca obiekty klasy `TreePath`. Gdy dysponujemy już ścieżką, to możemy pobrać węzeł znajdujący się na jej końcu, korzystając z metody `getLastpathComponent`. Aby na przykład odnaleźć aktualnie wybrany węzeł drzewa, korzystamy z metody `getSelectionPath` klasy `JTree`. W rezultacie otrzymujemy obiekt klasy `TreePath`, za pomocą którego możemy z kolei uzyskać aktualnie wybrany węzeł.

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
   selectionPath.getLastPathComponent();
```

Ponieważ potrzeba taka pojawia się bardzo często, to udostępniono dodatkową metodę, która natychmiast zwraca wybrany węzeł drzewa.

```
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
   tree.getLastSelectedPathComponent();
```

Metody tej nie nazwano `getSelectedNode`, ponieważ drzewo nie operuje na węzłach, a jedynie na ścieżkach.

> Ścieżki są jedną z dwu metod wykorzystywanych przez klasę `JTree` do opisu węzłów. Istnieje kilka metod klasy `JTree`, które wykorzystują lub zwracają wartość indeksu określającą *pozycję wiersza*. Jest to po prostu numer wiersza (numeracja rozpoczyna się od 0), w którym znajduje się dany węzeł. Numerowane są jedynie węzły widoczne w danym momencie i w związku z tym numer wiersza danego węzła ulega zmianie podczas operacji, takich jak wstawianie, zwijanie i rozwijanie, wykonywanych dla węzłów poprzedzających dany węzeł w drzewie. Dlatego też należy unikać korzystania z numerów węzłów. Wszystkie metody klasy `JTree` używające numerów węzłów posiadają odpowiedniki posługujące się ścieżkami.

Po uzyskaniu wybranego węzła możemy dodać do niego węzły podrzędne, ale nie w poniższy sposób:

```
selectedNode.add(newNode);   // NIE!
```

Zmieniając strukturę węzła, dokonujemy zmiany jedynie modelu, ale jego widok nie uzyskuje o tym informacji. Możemy sami wysłać odpowiednie zawiadomienie, ale jeśli skorzystamy z metody `insertNodeInto` klasy `DafaultTreeModel`, to zrobi to za nas automatycznie model drzewa. Na przykład poniższe wywołanie doda nowy węzeł jako ostatni węzeł podrzędny wybranego węzła i powiadomi o tym widok drzewa:

```
model.insertNodeInto(newNode, selectedNode,
   selectedNode.getChildCount());
```

Natomiast metoda `removeNodeFromParent` usunie węzeł i powiadomi widok drzewa:

```
model.removeNodeFromParent(selectedNode);
```

Jeśli struktura drzewa pozostaje zachowana, a zmienił się jedynie obiekt użytkownika, wystarczy wywołać:

```
model.nodeChanged(changedNode);
```

Automatyczne powiadamianie widoku drzewa jest główna zaletą korzystania z klasy `DefaultTreeModel`. Jeśli utworzymy własny model drzewa, to sami musimy zawiadamiać jego widok o zmianach. (Patrz *Core Java Foundation Classes* autorstwa Kim Topley).

> Klasa `DefaultTreeModel` posiada metodę `reload`, która powoduje przeładowanie modelu. Nie należy jednak z niej korzystać w celu aktualizacji widoku drzewa po każdej przeprowadzonej zmianie modelu. Przeładowanie modelu powoduje, że zwijane są wszystkie węzły drzewa z wyjątkiem węzłów podrzędnych korzenia. Użytkownik będzie więc zmuszony rozwijać drzewo po każdej wprowadzonej zmianie.

Gdy widok drzewa jest zawiadamiany o zmianie w strukturze węzłów, to aktualizuje odpowiednio reprezentację graficzną drzewa. Nie rozwija jednak przy tym automatycznie węzłów, jeśli nowe węzły zostały dodane jako węzły podrzędne do zwiniętego węzła. Szczególnie jeśli użytkownik doda nowy węzeł do węzła, który jest zwinięty, to nie wywoła żadnej zmiany w bieżącej prezentacji drzewa. Użytkownik nie będzie więc wiedzieć, czy nowy węzeł został faktycznie dodany, dopóki sam nie rozwinie odpowiedniego poddrzewa. W takim przypadku program powinien postarać o rozwinięcie odpowiednich węzłów, tak by widoczny był nowo dodany węzeł. W tym celu wykorzystać można metodę `makeVisible` klasy `JTree`. Jej parametrem jest ścieżka prowadząca do węzła, który powinien być widoczny.

Musimy więc skonstruować ścieżkę prowadzącą od korzenia do nowego węzła. Wywołamy w tym celu metodę `getPathToRoot` klasy `DefaultTreeModel`, która zwróci tablicę `TreeNode[]` wszystkich węzłów ścieżki od danego węzła do korzenia. Tablicę tę przekażemy jako parametr konstruktora klasy `TreePath`.

Poniżej demonstrujemy przykładowy kod rozwijający ścieżkę do nowego węzła.

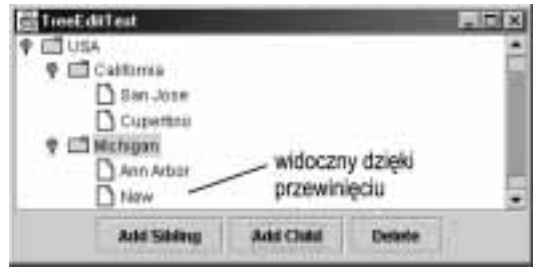
```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```

> Ciekawe, że klasa `DefaultTreeModel` ignoruje zupełnie istnienie klasy `TreePath`, mimo że musi komunikować się z klasą `JTree`. Klasa `JTree` wykorzystuje ścieżki, ale nigdy nie używa tablic węzłów.

Załóżmy teraz, że drzewo nasze umieszczone jest wewnątrz przewijalnego panelu. Po dodaniu nowego węzła może on nadal nie być widoczny, ponieważ znajdzie się poza widocznym fragmentem drzewa. Zamiast wywołać metodę `makeVisible`, wykorzystamy wtedy:

```
tree.scrollPathToVisible(path);
```

Wywołanie to spowoduje nie tylko rozwinięcie węzłów wzdłuż ścieżki prowadzącej do nowego węzła, ale i takie przewinięcie zawartości panelu, że nowy węzeł będzie widoczny (patrz rysunek 6.19).



Rysunek 6.19. *Przewinięcie panelu w celu prezentacji nowego węzła*

Domyślnie węzły drzewa nie mogą być modyfikowane. Jeśli jednak wywołamy:

```
tree.setEditable(true);
```

to użytkownik może edytować węzły, klikając je dwukrotnie myszą, zmieniając łańcuch opisujący węzeł i zatwierdzając zmianę klawiszem *Enter*. Dwukrotne kliknięcie węzła myszą powoduje wywołanie *domyślnego edytora komórki* implementowanego przez klasę `DefaultCellEditor` (patrz rysunek 6.20). Można zainstalować własny edytor, ale temat ten omówimy podczas przedstawiania tabel, w przypadku których zastosowanie edytorów komórek jest bardziej naturalne.

Rysunek 6.20. *Domyślny edytor komórek*

Listing 6.5 zawiera kompletny tekst źródłowy programu edycji drzewa. Umożliwia on wstawianie węzłów i edytowanie ich. Zwróćmy przy tym uwagę, w jaki sposób rozwijane są węzły drzewa i przewijany panel, tak aby można było zobaczyć nowe węzły dodane do drzewa.

Listing 6.5. *TreeEditTest.java*

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

/**
   Program demonstrujący edycję drzewa.
*/
public class TreeEditTest
{
   public static void main(String[] args)
   {
      JFrame frame = new TreeEditFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}
```

```
/**
   Ramka zawierająca trzy przyciski i edytowane drzewo.
*/
class TreeEditFrame extends JFrame
{
   public TreeEditFrame()
   {
      setTitle("TreeEditTest");
      setSize(WIDTH, HEIGHT);

      // tworzy drzewo

      TreeNode root = makeSampleTree();
      model = new DefaultTreeModel(root);
      tree = new JTree(model);
      tree.setEditable(true);

      // umieszcza drzewo w przewijalnym panelu

      JScrollPane scrollPane = new JScrollPane(tree);
      getContentPane().add(scrollPane, BorderLayout.CENTER);

      makeButtons();
      // tworzy panel przycisków

   }

   public TreeNode makeSampleTree()
   {
      DefaultMutableTreeNode root
         = new DefaultMutableTreeNode("World");
      DefaultMutableTreeNode country
         = new DefaultMutableTreeNode("USA");
      root.add(country);
      DefaultMutableTreeNode state
         = new DefaultMutableTreeNode("California");
      country.add(state);
      DefaultMutableTreeNode city
         = new DefaultMutableTreeNode("San Jose");
      state.add(city);
      city = new DefaultMutableTreeNode("Cupertino");
      state.add(city);
      state = new DefaultMutableTreeNode("Michigan");
      country.add(state);
      city = new DefaultMutableTreeNode("Ann Arbor");
      state.add(city);
      country = new DefaultMutableTreeNode("Germany");
      root.add(country);
      state = new DefaultMutableTreeNode("Schleswig-Holstein");
      country.add(state);
      city = new DefaultMutableTreeNode("Kiel");
      state.add(city);
      return root;
   }

   /**
      Tworzy przyciski umożliwiające dodanie węzła siostrzanego,
      dodanie węzła podrzędnego oraz usunięcie wybranego węzła.
```

```
*/
public void makeButtons()
{
   JPanel panel = new JPanel();
   JButton addSiblingButton = new JButton("Add Sibling");
   addSiblingButton.addActionListener(new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            DefaultMutableTreeNode selectedNode
               = (DefaultMutableTreeNode)
               tree.getLastSelectedPathComponent();

            if (selectedNode == null) return;

            DefaultMutableTreeNode parent
               = (DefaultMutableTreeNode)
               selectedNode.getParent();

            if (parent == null) return;

            DefaultMutableTreeNode newNode
               = new DefaultMutableTreeNode("New");

            int selectedIndex = parent.getIndex(selectedNode);
            model.insertNodeInto(newNode, parent,
               selectedIndex + 1);

            // wyświetla nowy węzeł

            TreeNode[] nodes = model.getPathToRoot(newNode);
            TreePath path = new TreePath(nodes);
            tree.scrollPathToVisible(path);
         }
      });
   panel.add(addSiblingButton);

   JButton addChildButton = new JButton("Add Child");
   addChildButton.addActionListener(new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            DefaultMutableTreeNode selectedNode
               = (DefaultMutableTreeNode)
               tree.getLastSelectedPathComponent();

            if (selectedNode == null) return;

            DefaultMutableTreeNode newNode
               = new DefaultMutableTreeNode("New");
            model.insertNodeInto(newNode, selectedNode,
               selectedNode.getChildCount());

            // wyświetla nowy węzeł

            TreeNode[] nodes = model.getPathToRoot(newNode);
            TreePath path = new TreePath(nodes);
```

```
                tree.scrollPathToVisible(path);
            }
         });
      panel.add(addChildButton);

      JButton deleteButton = new JButton("Delete");
      deleteButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               DefaultMutableTreeNode selectedNode
                  = (DefaultMutableTreeNode)
                  tree.getLastSelectedPathComponent();

               if (selectedNode != null &&
                  selectedNode.getParent() != null)
                  model.removeNodeFromParent(selectedNode);
            }
         });
      panel.add(deleteButton);
      getContentPane().add(panel, BorderLayout.SOUTH);
   }

   private DefaultTreeModel model;
   private JTree tree;
   private static final int WIDTH = 400;
   private static final int HEIGHT = 200;
}
```

javax.swing.JTree

- `TreePath getSelectionPath()` zwraca ścieżkę do aktualnie wybranego węzła (lub pierwszego wybranego, jeśli wybranych zostało wiele węzłów) bądź wartość `null`, jeśli żaden węzeł nie jest wybrany.
- `Object getLastSelectedPathComponent()` zwraca obiekt aktualnie wybranego węzła (lub pierwszego wybranego, jeśli wybranych zostało wiele węzłów) bądź wartość `null`, jeśli żaden węzeł nie jest wybrany.
- `void makeVisible(TreePath path)` rozwija wszystkie węzły wzdłuż ścieżki.
- `void scrollPathToVisible(TreePath path)` rozwija wszystkie węzły wzdłuż ścieżki oraz, jeśli drzewo jest umieszczone w panelu przewijalnym, przewija panel, tak by widoczny był ostatni węzeł ścieżki.

javax.swing.tree.TreePath

- `Object getLastPathComponent()` zwraca ostatni obiekt ścieżki czyli węzeł, do którego dostęp reprezentuje ścieżka.

javax.swing.tree.TreeNode

- `TreeNode getParent()` zwraca węzeł nadrzędny danego węzła.
- `TreeNode getChildAt(int index)` zwraca węzeł podrzędny o danym indeksie. Wartość indeksu musi być z przedziału od 0 do `getChildCount() - 1`.
- `int getChildCount()` zwraca liczbę węzłów podrzędnych danego węzła.
- `Enumeration children()` zwraca obiekt wyliczenia umożliwiający przeglądanie wszystkich węzłów podrzędnych danego węzła.

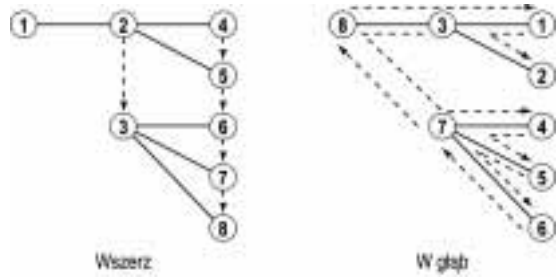
javax.swing.DefaultTreeModel

- `void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)` wstawia `newChild` jako nowy węzeł podrzędny węzła `parent` o podanym indeksie.
- `void removeNodeFromParent(MutableTreeNode node)` usuwa węzeł `node` z modelu.
- `void nodeChanged(TreeNode node)` zawiadamia obiekty nasłuchujące modelu o modyfikacji węzła `node`.
- `void nodesChanged(TreeNode parent, int[] changedChildIndexes)` zawiadamia obiekty nasłuchujące modelu, że uległy modyfikacji węzły podrzędne węzła `parent` o podanych indeksach.
- `void reload()` ładuje wszystkie węzły do modelu. Operacja ta wykonywana powinna być jedynie, gdy zaszła zasadnicza modyfikacja węzłów spowodowana zewnętrzną przyczyną.

Przeglądanie węzłów

Często, aby odnaleźć poszukiwany węzeł, musimy przejrzeć wszystkie jego węzły. Klasa `DefaultMutableTreeNode` posiada kilka metod przydatnych w tym celu.

Metody `breadthFirstEnumeration` i `depthFirstEnumeration` zwracają obiekty wyliczeń, których metoda `nextElement` umożliwia przeglądanie wszystkich węzłów podrzędnych bieżącego węzła, korzystając z metody przeglądania wszerz i w głąb. Rysunek 6.21 pokazuje porządek przeglądania węzłów przykładowego drzewa w obu metodach.



Rysunek 6.21. *Kolejność przeglądania węzłów drzewa*

Przeglądanie wszerz jest nieco łatwiejsze do wyjaśnienia. Drzewo przeglądane jest warstwami. Najpierw odwiedzany jest korzeń drzewa, potem jego wszystkie węzły podrzędne, a następnie ich węzły podrzędne itd.

W przypadku przeglądania w głąb sposób poruszania się po drzewie przypomina mysz biegającą po labiryncie w kształcie drzewa. Porusza się ona wzdłuż ścieżek drzewa aż do napotkania liścia, po czym wycofuje się i wybiera następną ścieżkę.

Taki sposób przeglądania drzewa nazywany bywa także *przeglądaniem od końca*, ponieważ węzły podrzędne są przeglądane przed węzłami nadrzędnymi. Zgodnie z tym nazewnictwem dodano metodę przeszukiwania od końca `postOrderTraversal` będącą synonimem metody `depthFirstTraversal` przeszukiwania w głąb. Aby zestaw ten był kompletny, uzupełniono go metodą `preOrderTraversal`, która przegląda drzewo również w głąb, ale najpierw węzły nadrzędne, a potem podrzędne.

Poniżej typowy przykład użycia metod przeglądania drzewa:

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
   operacje na węźle breadthFirst.nextElement();
```

Z przeglądaniem drzewa związana jest także metoda `pathFromAncestorEnumeration`, która wyznacza ścieżkę od jednego z przodków węzła do danego węzła i tworzy wyliczenie węzłów znajdujących się na ścieżce. Jej implementacja polega na wywoływaniu metody `getParent` do momentu znalezienia przodka, a następnie przejściu ścieżki z powrotem.

Nasz następny program ma przeglądać drzewo. Będzie on wyświetlał drzewo dziedziczenia klas. Po wprowadzeniu nazwy klasy w polu tekstowym w dolnej części okna klasa ta zostanie dodana do drzewa wraz z jej wszystkimi klasami bazowymi (patrz rysunek 6.22).

Rysunek 6.22.
Drzewo dziedziczenia

W przykładzie tym wykorzystamy fakt, że obiekt użytkownika danego węzła może być dowolnego typu. Ponieważ węzły naszego drzewa reprezentować będą klasy, umieścimy w nich obiekty klasy `Class`.

Ponieważ nie chcemy, by drzewo zawierało wiele wystąpień tej samej klasy, to najpierw będziemy musieli je przeszukać, aby sprawdzić, czy dana klasa już występuje. Poniższa metoda znajduje węzeł zawierający podany obiekt użytkownika pod warunkiem, że istnieje on w drzewie.

```
public DefaultMutableTreeNode findUserObject(Object obj)
{
   Enumeration e = root.breadthFirstEnumeration();
   while (e.hasMoreElements())
   {
      DefaultMutableTreeNode node
         = (DefaultMutableTreeNode)e.nextElement();
      if (node.getUserObject().equals(obj))
         return node;
   }
   return null;
}
```

Rysowanie węzłów

Specyfika aplikacji często wymaga zmiany sposobu prezentacji drzewa. Najczęściej polega ona na zmianie ikon folderów i liści, zmianie czcionki opisującej węzeł lub zastąpieniu opisu obrazkiem. Wszystkie te zmiany są osiągalne przez zainstalowanie nowego *obiektu rysującego komórki* danego drzewa. Domyślnie klasa `JTree` wykorzystuje obiekt klasy `DefaultTreeCellRenderer`, która jest klasą pochodną klasy `JLabel`. Etykieta reprezentowana przez obiekt klasy `JLabel` składa się w tym przypadku z ikony węzła i jego opisu.

> **Obiekt rysujący komórki drzewa nie jest odpowiedzialny za narysowanie uchwytów węzłów umożliwiających zwijanie i rozwijanie poddrzew. Uchwyty te są częścią ogólnego wyglądu komponentów interfejsu użytkownika i nie powinny być modyfikowane.**

Wygląd drzewa zmodyfikować możemy na trzy różne sposoby.

1. Zmieniamy ikony, czcionkę oraz kolor tła wykorzystywany przez obiekt klasy `DefaultTreeCellRenderer`. Ustawienia te wykorzystywane będą podczas rysowania wszystkich węzłów drzewa.

2. Instalujemy własny obiekt rysujący komórki drzewa, który należeć będzie do klasy pochodnej klasy `DefaultTreeCellRenderer`. Może on zmieniać ikony, czcionkę i kolor tła, rysując poszczególne węzły.

3. Instalujemy własny obiekt rysujący komórki drzewa, który implementować będzie interfejs `TreeCellRenderer` i rysować dowolną reprezentację węzłów drzewa.

Przyjrzyjmy się po kolei tym sposobom. Najprostszy z nich wymaga utworzenia obiektu klasy `DefaultTreeCellRenderer`, zmiany ikony i zainstalowania obiektu dla danego drzewa:

```
DefaultTreeCellRenderer renderer
   = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new imageIcon("blue-ball.gif "));
   // ikona liści
renderer.setClosedIcon(new imageIcon("red-ball.gif "));
   // ikona zwiniętego węzła
renderer.setOpenIcon(new imageIcon("yellow-ball.gif "));
      // ikona rozwiniętego węzła
tree.setCellRenderer(renderer);
```

Efekt takiego rozwiązania przedstawia rysunek 6.22.

Nie zalecamy zmiany czcionki bądź koloru tła dla całego drzewa, ponieważ to jest zadaniem wyglądu komponentów interfejsu użytkownika.

Możemy jednak zmieniać czcionkę pojedynczych węzłów drzewa dla ich wyróżnienia. Jeśli przyjrzymy się uważnie rysunkowi 6.22, to zauważymy, że nazwy klas *abstrakcyjnych* pisane są kursywą.

Aby zmieniać wygląd poszczególnych węzłów, musimy zainstalować własny obiekt rysujący komórki drzewa. Przypomina on obiekty rysujące komórki list, które omówiliśmy w 9. rozdziale książki *Java 2. Podstawy*. Interfejs `TreeCellRenderer` posiada pojedynczą metodę:

```
Component getTreeCellRendererComponent(JTree tree,
   Object value, boolean selected, boolean expanded,
   boolean leaf, int row, boolean hasFocus)
```

Metoda ta wywoływana jest dla każdego węzła. Zwraca ona *komponent*, którego metoda `paint` rysuje reprezentację węzła drzewa. Metodzie `paint` przekazywany jest odpowiedni obiekt `Graphics`, który zawiera informację o współrzędnych umożliwiającą narysowanie reprezentacji węzła we właściwym miejscu drzewa.

> Zastanawiać może, dlaczego nie wystarczy po prostu umieścić metody `paint` w klasie implementującej interfejs `TreeCellRenderer`. Przyczyna tego jest bardzo prozaiczna. Często łatwiej wykorzystać dla prezentacji węzłów drzewa istniejące już komponenty interfejsu, niż programować metodę `paint` od podstaw. Z metody tej skorzystali także projektanci biblioteki Swing, ponieważ domyślny obiekt rysujący komórki drzewa stanowi rozszerzenie komponentu `JLabel`, który zapewnia odpowiednie rozmieszczenie ikony i tekstu.

Metoda `getTreeCellRendererComponent` klasy `DefaultTreeCellRenderer` zwraca po prostu wartość `this`, czyli innymi słowy komponent etykiety. (Przypomnijmy, że klasa `DefaultTreeCellRenderer` jest pochodną klasy `JLabel`). Aby zmodyfikować komponent, musimy sami utworzyć klasę pochodną klasy `DefaultTreeCellRenderer` i zastąpić jej metodę `getTreeCellRendererComponent` implementacją, która:

- wywoła metodę klasy nadrzędnej w celu przygotowania danych komponentu etykiety,
- zmodyfikuje właściwości etykiety,
- zwróci `this`.

```
class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
{
   public Component getTreeCellRendererComponent(JTree tree,
      Object value, boolean selected, boolean expanded,
      boolean leaf, int row, boolean hasFocus)
   {
      super.getTreeCellRendererComponent(tree, value,
         selected, expanded, leaf, row, hasFocus);
      DefaultMutableTreeNode node
         = (DefaultMutableTreeNode)value;
      sprawdź node.getUserObject();
```

```
        Font font = odpowiednia czcionka ;
        setFont(plainFont);
        return this;
    }
```

Parametr `value` **metody** `getTreeCellRendererComponent` **nie jest obiektem użytkownika, ale obiektem reprezentującym** ***węzeł****! Przypomnijmy w tym miejscu, że obiekty użytkownika wykorzystywane są przez klasę* `DefaultMutableTreeNode`*, natomiast klasa* `JTree` *może zawierać węzły dowolnego typu. Jeśli drzewo składa się z węzłów klasy* `DefaultMutableTreeNode`*, to obiekt użytkownika możemy pobrać z nich dopiero, wywołując metodę* `getUserObject`*, tak jak uczyniliśmy to w powyższym przykładzie.*

Obiekt klasy `DefaultTreeCellRenderer` **wykorzystuje ten sam obiekt klasy** `JLabel` **dla wszystkich węzłów, zmieniając jedynie tekst etykiety. Jeśli więc dokonamy zmiany czcionki dla danego węzła, to musimy przywrócić czcionkę domyślną, gdy metoda zostanie wywołana po raz kolejny. W przeciwnym razie wszystkie kolejne węzły zostaną opisane zmienioną czcionką! Kod programu w listingu 6.6 pokazuje sposób przywrócenia domyślnej czcionki.**

Nie będziemy pokazywać osobnego przykładu obiektu rysującego komórkę drzewa dowolnej postaci. Sposób jego działania jest analogiczny do pracy obiektu rysującego komórki listy przedstawionego na początku tego rozdziału.

Zaprzęgnijmy zatem do pracy obiekty rysujące komórki drzewa. Listing 6.6 zawiera tekst źródłowy programu tworzącego drzewo klas. Program prezentuje drzewo dziedziczenia klas, wyróżniając klasy abstrakcyjne kursywą. W polu tekstowym w dolnej części okna programu użytkownik może wpisać nazwę dowolnej klasy, a następnie wybrać klawisz *Enter* lub przycisk *Add*, aby dodać klasę i jej klasy bazowe do drzewa. Należy podać wyłącznie pełną nazwę klasy, czyli na przykład `java.util.ArrayList`.

Działanie programu jest trochę skomplikowane, ponieważ wykorzystuje on przy tworzeniu drzewa refleksję klas, co odbywa się wewnątrz metody `addClass`. (Szczegóły nie są w tym przypadku istotne. Przykładowy program tworzy akurat drzewo klas, ponieważ drzewo dziedziczenia jest dobrym przykładem struktury drzewiastej. Zwykle programy reprezentują jednak za pomocą drzewa inne struktury danych). Metoda `addClass` wywołuje metodę `findUserObject`, aby sprawdzić, czy klasa znajduje się już w drzewie. Metoda `findUserObject` przegląda drzewo wszerz. Jeśli klasa nie znajduje się jeszcze w drzewie, to program dodaje najpierw do drzewa jej klasy bazowe, a na końcu daną klasę i troszczy się o to, by jej węzeł był widoczny.

Obiekt klasy `ClassNameTreeCellRenderer` prezentuje nazwę klasy czcionką prostą lub pochyłą w zależności od modyfikatora `ABSTRACT` obiektu klasy `Class`. Program korzysta z czcionki, którą dla reprezentacji etykiet drzewa przewidział bieżący wygląd komponentów i tworzy na jej podstawie wersję pochyłą. Ponieważ wszystkie wywołania zwracają ten sam obiekt klasy `JLabel`, to kolejne wywołanie metody `getTreeCellRendererComponent` musi odtworzyć oryginalną czcionkę, jeśli wcześniej użyta była jej wersja pochyła.

Konstruktor klasy `ClassTreeFrame` zmienia dodatkowo ikony reprezentujące węzły drzewa.

Listing 6.6. *ClassTree.java*

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
   Program demonstrujący sposób rysowania
   komórek drzewa na przykładzie drzewa klas i ich
   klas bazowych.
*/
public class ClassTree
{
   public static void main(String[] args)
   {
      JFrame frame = new ClassTreeFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca drzewo klas oraz pole tekstowe
   i przycisk umożliwiające dodawanie klas do drzewa.
*/
class ClassTreeFrame extends JFrame
{
   public ClassTreeFrame()
   {
      setTitle("ClassTree");
      setSize(WIDTH, HEIGHT);

      // korzeniem drzewa jest klasa Object
      root = new DefaultMutableTreeNode(java.lang.Object.class);
      model = new DefaultTreeModel(root);
      tree = new JTree(model);

      // dodaje klasę do drzewa
      addClass(getClass());

      // tworzy ikony węzłów
      ClassNameTreeCellRenderer renderer
         = new ClassNameTreeCellRenderer();
      renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
      renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
      renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
      tree.setCellRenderer(renderer);

      getContentPane().add(new JScrollPane(tree),
         BorderLayout.CENTER);

      addTextField();
   }
```

```
/**
   Dodaje pole tekstowe i przycisk.
*/
public void addTextField()
{
   JPanel panel = new JPanel();

   ActionListener addListener = new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            // dodaje do drzewa klasę, której nazwa znajduje się
            // w polu tekstowym
            try
            {
               String text = textField.getText();
               addClass(Class.forName(text));
               // usuwa tekst z pola tekstowego
               textField.setText("");
            }
            catch (ClassNotFoundException e)
            {
               JOptionPane.showMessageDialog(null,
                  "Class not found");
            }
         }
      };

   // tworzy pole tekstowe
   textField = new JTextField(20);
   textField.addActionListener(addListener);
   panel.add(textField);

   JButton addButton = new JButton("Add");
   addButton.addActionListener(addListener);
   panel.add(addButton);

   getContentPane().add(panel, BorderLayout.SOUTH);
}

/**
   Znajduje obiekt w drzewie.
   @param obj szukany obiekt
   @return węzeł zawierający obiekt lub null,
   jeśli obiektu nie ma w drzewie
*/
public DefaultMutableTreeNode findUserObject(Object obj)
{
   // znajduje węzeł zawierający dany obiekt użytkownika
   Enumeration e = root.breadthFirstEnumeration();
   while (e.hasMoreElements())
   {
      DefaultMutableTreeNode node
         = (DefaultMutableTreeNode)e.nextElement();
      if (node.getUserObject().equals(obj))
         return node;
   }
```

```
      return null;
   }

   /**
      Dodaje do drzewa klasę i jej klasy bazowe,
      których nie ma jeszcze w drzewie.
      @param c dodawana klas
      @return nowo dodany węzeł.
   */
   public DefaultMutableTreeNode addClass(Class c)
   {
      // dodaje klasę do drzewa

      // pomija typy, które nie są klasami
      if (c.isInterface() || c.isPrimitive()) return null;

      // jeśli klasa znajduje się już w drzewie, to zwraca jej węzeł
      DefaultMutableTreeNode node = findUserObject(c);
      if (node != null) return node;

      // klasa nie znajduje się w drzewie
      // najpierw należy dodać do drzewa jej klasy bazowe

      Class s = c.getSuperclass();

      DefaultMutableTreeNode parent;
      if (s == null)
         parent = root;
      else
         parent = addClass(s);

      // dodaje klasę jako węzeł podrzędny
      DefaultMutableTreeNode newNode
         = new DefaultMutableTreeNode(c);
      model.insertNodeInto(newNode, parent,
         parent.getChildCount());

      // sprawia, że węzeł jest widoczny
      TreePath path = new TreePath(model.getPathToRoot(newNode));
      tree.makeVisible(path);

      return newNode;
   }

   private DefaultMutableTreeNode root;
   private DefaultTreeModel model;
   private JTree tree;
   private JTextField textField;
   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;
}

/**
   Klasa opisująca węzły drzewa czcionką zwykłą lub pochyloną
   (w przypadku klas abstrakcyjnych).
*/
class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
```

```
{
   public Component getTreeCellRendererComponent(JTree tree,
      Object value, boolean selected, boolean expanded,
      boolean leaf, int row, boolean hasFocus)
   {
      super.getTreeCellRendererComponent(tree, value,
         selected, expanded, leaf, row, hasFocus);
      // pobiera obiekt użytkownika
      DefaultMutableTreeNode node
         = (DefaultMutableTreeNode)value;
      Class c = (Class)node.getUserObject();

      // przy pierwszym użyciu tworzy czcionkę
      // pochyłą odpowiadającą danej czcionce prostej
      if (plainFont == null)
      {
         plainFont = getFont();
         /*
            obiekt rysujący komórkę drzewa wywoływany jest czasami
            dla etykiety, która nie posiada określonej czcionki (null).
         */
         if (plainFont != null)
            italicFont = plainFont.deriveFont(Font.ITALIC);
      }

      // ustawia czcionkę pochyłą, jeśli klasa jest abstrakcyjna
      if ((c.getModifiers() & Modifier.ABSTRACT) == 0)
         setFont(plainFont);
      else
         setFont(italicFont);
      return this;
   }

   private Font plainFont = null;
   private Font italicFont = null;
};
```

javax.swing.tree.DefaultMutableTreeNode

- `Enumeration breadthFirstEnumeration()`
- `Enumeration depthFirstEnumeration()`
- `Enumeration preOrderEnumeration()`
- `Enumeration postOrderEnumeration()`

  Zwracają obiekt wyliczenia umożliwiający przeglądanie wszystkich węzłów drzewa w odpowiednim porządku: wszerz, gdzie węzły podrzędne leżące bliżej korzenia odwiedzane są wcześniej, w głąb, gdzie wszystkie węzły podrzędne danego węzła są odwiedzane, zanim odwiedzone zostaną jego węzły siostrzane. Metoda `postOrderEnumeration` stanowi synonim metody `depthFirstEnumeration`. Metoda `preOrderEnumeration` przegląda drzewo podobnie do niej, z tą różnicą, że węzły nadrzędne przeglądane są przed ich węzłami podrzędnymi.

javax.swing.tree.TreeCellRenderer

- `Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)` **zwraca komponent, którego metoda** `paint` **wywoływana jest w celu narysowania komórki drzewa.**

  Parametry:

`tree`	drzewo, do którego należy rysowana komórka,
`value`	rysowany węzeł,
`selected`	wartość `true`, jeśli węzeł jest wybrany,
`expanded`	wartość `true`, jeśli węzły podrzędne danego węzła są widoczne,
`leaf`	wartość `true`, jeśli węzeł jest liściem,
`row`	numer wiersza graficznej reprezentacji drzewa zawierającej węzeł,
`hasFocus`	wartość `true`, jeśli węzeł jest przeglądany w danym momencie przez użytkownika.

javax.swing.tree.DefaultTreeCellRenderer

- `void setLeafIcon(Icon icon)`
- `void setOpenIcon(Icon icon)`
- `void setClosedIcon(Icon icon)`

  Ustalają ikonę prezentującą odpowiednio: liść, węzeł rozwinięty, węzeł zwinięty.

Nasłuchiwanie zdarzeń w drzewach

Najczęściej komponent drzewa wykorzystywany jest razem z innym komponentem interfejsu użytkownika. Gdy użytkownik wybiera węzły drzewa, to inny komponent pokazuje pewną informację o nich, tak jak na przykład program przedstawiony na rysunku 6.23. Kiedy użytkownik wybiera węzeł drzewa reprezentujący klasę języka Java, to w polu tekstowym obok prezentowana jest informacja o jej zmiennych.

Rysunek 6.23. *Przeglądarka klas*

Aby uzyskać takie działanie programu, konieczne jest zainstalowanie *obiektu nasłuchującego wyboru w drzewie*. Obiekt ten musi implementować interfejs `TreeSelectionListener`, który posiada tylko jedną metodę:

```
void valueChanged(TreeSelectionEvent event)
```

Jest ona wywoływana za każdym razem, gdy węzeł drzewa zostaje wybrany lub przestaje być wybrany.

Obiekt nasłuchujący dodajemy do drzewa w zwykły sposób:

```
tree.addSelectionListener(listener);
```

Możemy określić sposób wyboru węzłów drzewa przez użytkownika. Wybrany może być tylko jeden węzeł, ciągły zakres węzłów lub dowolny, potencjalnie nieciągły zbiór węzłów. Klasa `JTree` wykorzystuje klasę `TreeSelectionModel` do zarządzania wyborem węzłów. Dla modelu, który musimy najpierw pobrać, określić możemy jeden z następujących stanów wyboru: `SINGLE_TREE_SELECTION`, `CONTIGUOUS_TREE_SELECTION` lub `DISCONTIGUOUS_TREE_SELECTION` (ten ostatni jest stanem domyślnym). Nasza przeglądarka umożliwiać będzie wybór pojedynczej klasy:

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

Po określeniu sposobu wyboru na drzewie nie musimy więcej zajmować się modelem wyboru.

> Sposób wyboru wielu węzłów drzewa zależy od bieżącego wyglądu komponentów interfejsu użytkownika. W przypadku wyglądu Metal wystarczy przytrzymać klawisz *Ctrl* podczas kliknięcia myszą, aby dokonać wyboru kolejnego węzła lub usunąć jego wybór, jeśli wcześniej był już wybrany. Podobnie przytrzymanie klawisza *Shift* umożliwia wybranie *zakresu* węzłów.

Aby uzyskać informacje o tym, które z węzłów zostały wybrane, wywołujemy metodę `getSelectionPaths`:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

W przypadku gdy możliwość wyboru ograniczyliśmy do jednego węzła, możemy skorzystać z metody `getSelectionPath`, która zwróci ścieżkę do pierwszego wybranego węzła lub wartość `null`, jeśli żaden węzeł nie został wybrany.

> Klasa `TreeSelectionEvent` dysponuje metodą `getPaths`, która zwraca tablicę obiektów klasy `TreePath` reprezentujących *zmiany wyboru*, a nie aktualnie wybrane węzły.

Program, którego kod źródłowy zawiera listing 6.7, wykorzystuje mechanizm wyboru węzła drzewa. Program ten stanowi rozbudowaną wersję programu z listingu 6.6. Jednak aby uczynić jego tekst źródłowy możliwie krótkim, zrezygnowaliśmy tym razem z użycia własnego obiektu rysującego komórki drzewa. W kodzie konstruktora ramki ograniczamy możliwość wyboru do jednego węzła i dodajemy do drzewa obiekt nasłuchujący wyboru. Gdy wywołana zostaje jego metoda `valueChanged`, ignorujemy jej parametr i korzystamy z metody `getSelectionPath`. Pobieramy ostatni węzeł uzyskanej ścieżki i zawarty w nim obiekt użytkownika. Następnie wywołujemy metodę `getFieldDescription`, która korzysta z mechanizmu refleksji, aby utworzyć łańcuch znaków opisujący wszystkie składowe danej klasy. Otrzymany łańcuch znaków wyświetlamy w polu tekstowym okna.

Listing 6.7. *ClassBrowserTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
   Program demonstrujący wybór węzła drzewa.
*/
public class ClassBrowserTest
{
   public static void main(String[] args)
   {
      JFrame frame = new ClassBrowserTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca drzewo klas, pole tekstowe pokazujące
   składowe wybranej klasy oraz pole tekstowe umożliwiające
   dodawanie nowych klas do drzewa.
*/
class ClassBrowserTestFrame extends JFrame
{
   public ClassBrowserTestFrame()
   {
      setTitle("ClassBrowserTest");
      setSize(WIDTH, HEIGHT);

      // w korzeniu drzewa znajduje się klasa Object
      root = new DefaultMutableTreeNode(java.lang.Object.class);
      model = new DefaultTreeModel(root);
      tree = new JTree(model);

      // dodaje klasę do drzewa
      addClass(getClass());

      // set up selection mode
      tree.addTreeSelectionListener(new
         TreeSelectionListener()
         {
            public void valueChanged(TreeSelectionEvent event)
            {
               // użytkownik wybrał inny węzeł drzewa
               // i należy zaktualizować opis klasy
               TreePath path = tree.getSelectionPath();
               if (path == null) return;
               DefaultMutableTreeNode selectedNode
                  = (DefaultMutableTreeNode)
                  path.getLastPathComponent();
```

```
            Class c = (Class)selectedNode.getUserObject();
            String description = getFieldDescription(c);
            textArea.setText(description);
         }
      });
   int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
   tree.getSelectionModel().setSelectionMode(mode);

   // pole tekstowe zawierające opis klasy
   textArea = new JTextArea();

   // dodaje komponenty drzewa i pola tekstowego do panelu
   JPanel panel = new JPanel();
   panel.setLayout(new GridLayout(1, 2));
   panel.add(new JScrollPane(tree));
   panel.add(new JScrollPane(textArea));

   getContentPane().add(panel, BorderLayout.CENTER);

   addTextField();
}

/**
   Dodaje pole tekstowe i przycisk "Add"
   umożliwiające dodanie nowej klasy do drzewa.
*/
public void addTextField()
{
   JPanel panel = new JPanel();

   ActionListener addListener = new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            // dodaje do drzewa klasę, której nazwa
            // znajduje się w polu tekstowym
            try
            {
               String text = textField.getText();
               addClass(Class.forName(text));
               // clear text field to indicate success
               textField.setText("");
            }
            catch (ClassNotFoundException e)
            {
               JOptionPane.showMessageDialog(null,
                  "Class not found");
            }
         }
      };

   // pole tekstowe, w którym wprowadzane są nazwy nowych klas
   textField = new JTextField(20);
   textField.addActionListener(addListener);
   panel.add(textField);
```

```
      JButton addButton = new JButton("Add");
      addButton.addActionListener(addListener);
      panel.add(addButton);

      getContentPane().add(panel, BorderLayout.SOUTH);
   }

   /**
      Wyszukuje obiekt w drzewie.
      @param obj szukany obiekt
      @return węzeł zawierający szukany obiekt lub null,
      jeśli obiekt nie znajduje się w drzewie
   */
   public DefaultMutableTreeNode findUserObject(Object obj)
   {
      // szuka węzła zawierającego obiekt użytkownika
      Enumeration e = root.breadthFirstEnumeration();
      while (e.hasMoreElements())
      {
         DefaultMutableTreeNode node
            = (DefaultMutableTreeNode)e.nextElement();
         if (node.getUserObject().equals(obj))
            return node;
      }
      return null;
   }

   /**
      Dodaje do drzewa klasę i jej klasy bazowe,
      których nie ma jeszcze w drzewie.
      @param c dodawana klasa
      @return nowo dodany węzeł.
   */
   public DefaultMutableTreeNode addClass(Class c)
   {
      /// dodaje klasę do drzewa

      // pomija typy, które nie są klasami
      if (c.isInterface() || c.isPrimitive()) return null;

      // jeśli klasa znajduje się już w drzewie, to zwraca jej węzeł
      DefaultMutableTreeNode node = findUserObject(c);
      if (node != null) return node;

      // klasa nie znajduje się w drzewie
      // najpierw należy dodać do drzewa jej klasy bazowe

      Class s = c.getSuperclass();

      DefaultMutableTreeNode parent;
      if (s == null)
         parent = root;
      else
         parent = addClass(s);

      // dodaje klasę jako węzeł podrzędny
      DefaultMutableTreeNode newNode
         = new DefaultMutableTreeNode(c);
```

```
            model.insertNodeInto(newNode, parent,
               parent.getChildCount());

            // sprawia, że węzeł jest widoczny
            TreePath path = new TreePath(model.getPathToRoot(newNode));
            tree.makeVisible(path);

            return newNode;
         }

         /**
            Zwraca opis składowych klasy.
            @param klasa
            @return łańcuch znaków zawierający nazwy i typy zmiennych
         */
         public static String getFieldDescription(Class c)
         {
            // korzysta z mechanizmu refleksji
            StringBuffer r = new StringBuffer();
            Field[] fields = c.getDeclaredFields();
            for (int i = 0; i < fields.length; i++)
            {
               Field f = fields[i];
               if ((f.getModifiers() & Modifier.STATIC) != 0)
                  r.append("static ");
               r.append(f.getType().getName());
               r.append(" ");
               r.append(f.getName());
               r.append("\n");
            }
            return r.toString();
         }

         private DefaultMutableTreeNode root;
         private DefaultTreeModel model;
         private JTree tree;
         private JTextField textField;
         private JTextArea textArea;
         private static final int WIDTH = 400;
         private static final int HEIGHT = 300;
      }
```

javax.swing.JTree

- `TreePath getSelectionPath()`
- `TreePath[] getSelectionPaths()`

  Zwracają odpowiednio ścieżkę do pierwszego wybranego węzła lub tablicę ścieżek do wybranych węzłów. Jeśli żaden węzeł nie jest wybrany, obie metody zwracają `null`.

javax.swing.event.TreeSelectionListener

- `void valueChanged(TreeSelectionEvent event)` wywoływana, gdy węzeł zostaje wybrany lub przestaje być wybrany.

`javax.swing.event.TreeSelectionEvent`

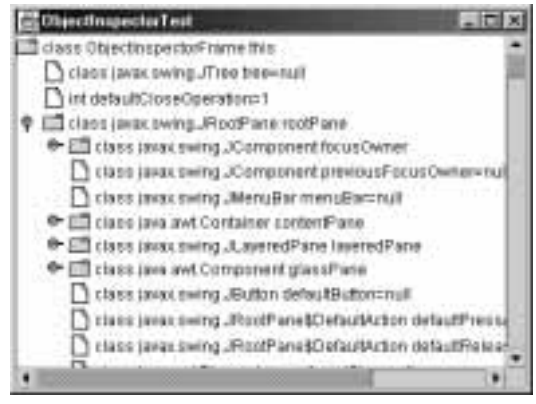
- `TreePath getPath()`
- `TreePath[] getPaths()`

  Zwracają odpowiednio ścieżkę do pierwszego obiektu lub tablicę ścieżek obiektów, których stan uległ zmianie na skutek danego zdarzenia wyboru. Jeśli interesują nas wybrane elementy, a nie zmiana ich wyboru, to powinniśmy skorzystać z metody `getSelectionPath` lub `getSelectionPaths` klasy `JTree`.

Własne modele drzew

Ostatnim przykładem wykorzystania drzew będzie program umożliwiający inspekcję wartości zmiennych, podobnie jak czynią to narzędzia uruchomieniowe (patrz rysunek 6.24).

Rysunek 6.24. *Drzewo inspekcji obiektów*



Zanim rozpoczniemy omawianie programu, zalecamy, by skompilować go, uruchomić i zapoznać się z jego działaniem. Każdy węzeł utworzonego przez program drzewa odpowiada zmiennej składowej obiektu. Jeśli z kolei ta zmienna reprezentuje także obiekt, to możemy rozwinąć jej węzeł, aby sprawdzić zmienne także i tego obiektu. Program umożliwia inspekcję obiektów składających się na jego interfejs użytkownika. Jeśli rozejrzymy się trochę po drzewie, to znajdziemy znajome obiekty odpowiadające komponentom interfejsu użytkownika. Jednocześnie nabierzemy także respektu dla złożoności mechanizmów biblioteki Swing, która nie jest zwykle widoczna dla programisty.

Istotna różnica w działaniu tego programu w stosunku do poprzednich przykładów polega na tym, że nie używa on klasy `DefaultTreeModel`. Jeśli program dysponuje już danymi zorganizowanymi w hierarchiczną strukturę, to nie ma sensu duplikować jej za pomocą nowego modelu i dodatkowo zajmować się jeszcze zapewnieniem synchronizacji obu struktur. Sytuacja taka występuje właśnie w przypadku naszego programu, ponieważ obiekty interfejsu użytkownika są już powiązane wzajemnymi referencjami.

Interfejs `TreeModel` definiuje szereg metod. Pierwsza ich grupa umożliwia klasie `JTree` odnalezienie węzłów drzewa przez pobranie najpierw jego korzenia, a później węzłów podrzędnych. Klasa `JTree` korzysta z tych metod jedynie, gdy użytkownik rozwija węzeł drzewa.

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

Przykład ten ukazuje, dlaczego interfejs `TreeModel`, podobnie jak klasa `JTree`, nie korzysta bezpośrednio z pojęcia węzłów. Korzeń i jego węzły podrzędne mogą być dowolnymi obiektami. Interfejs `TreeModel` umożliwia klasie `JTree` uzyskanie informacji o sposobie ich powiązania.

Kolejna metoda interfejsu `TreeModel` wykonuje operację odwrotną do metody `getChild`:

```
int getIndexOfChild(Object parent, Object child)
```

Metoda ta może zostać zaimplementowana za pomocą wymienionych wcześniej trzech metod — patrz kod programu w listingu 6.8.

Model drzewa informuje klasę `JTree` o tym, które węzły powinny zostać przedstawione jako liście:

```
boolean isLeaf(Object node)
```

Jeśli w wyniku działania programu dane modelu drzewa ulegają zmianie, to drzewo musi zostać o tym powiadomione, aby dokonać aktualizacji swojego widoku. Dlatego też drzewo powinno być dodane jako obiekt nasłuchujący `TreeModelListener` do modelu. Model musi więc posiadać typowe metody związane z zarządzaniem obiektami nasłuchującymi:

```
void addTreeModelListener(TreeModelListener l)
void removeTreeModelListener(TreeModelListener l)
```

Implementację tych metod pokazuje także listing 6.8.

Gdy zawartość modelu ulega zmianie, to wywołuje on jedną z czterech metod definiowanych przez interfejs `TreeModeListener`:

```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)
```

Parametr tych metod opisuje miejsce wystąpienia zmian w drzewie. Szczegóły tworzenia obiektu zdarzenia opisującego wstawienie bądź usunięcie węzła są dość skomplikowane, ale musimy się nimi zajmować tylko wtedy, gdy węzły naszego drzewa są rzeczywiście dodawane bądź usuwane. Listing 6.8 pokazuje konstrukcję obiektu zdarzenia w przypadku zastąpienia korzenia nowym obiektem.

> Programiści biblioteki Swing znużeni wysyłaniem obiektów zdarzeń utworzyli klasę `javax.swing.EventListenerList` zawierającą listę obiektów nasłuchujących. Rozdział 8. książki *Java 2. Podstawy* zawiera więcej informacji na ten temat.

Jeśli użytkownik zmodyfikuje węzeł drzewa, to model zostaje o tym poinformowany przez wywołanie jego metody:

```
void valueForPathChanged(TreePath path, Object newValue)
```

Gdy nie zezwolimy użytkownikowi na edycję drzewa, to metoda ta nigdy nie zostanie wywołana.

W przypadku takim konstrukcja modelu drzewa staje się łatwa. Należy zaimplementować trzy poniższe metody:

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

Opisują one strukturę drzewa. Następnie musimy dostarczyć jeszcze implementacji pozostałych pięciu metod, co pokazano w listingu 6.8 i będziemy gotowi do wyświetlenia własnego drzewa.

Zajmijmy się teraz implementacją naszego programu. Drzewo zawierać będzie obiekty klasy `Variable`.

> Gdybyśmy skorzystali z klasy `DefaultTreeModel`, to węzły naszego drzewa byłyby obiektami klasy `DefaultMutableTreeNode` zawierającymi *obiekty użytkownika* klasy `Variable`.

Załóżmy, że inspekcji poddać chcemy zmienną:

```
Employee joe;
```

Zmienna ta posiada *typ* `Employee.class`, *nazwę* `"joe"` i *wartość*, którą stanowi referencja do obiektu `joe`. W programie zdefiniujemy klasę `Variable` służącą reprezentacji zmiennych:

```
Variable v = new Variable(Employee.Class, "joe", joe);
```

Jeśli zmienna jest typu podstawowego, musimy użyć dla jej wartości obiektu opakowującego.

```
new Variable(double.Class, "salary", new Double(salary));
```

Jeśli typem zmiennej jest klasa, to zawierać będzie ona *pola*. Korzystając z mechanizmu refleksji, dokonujemy wyliczenia wszystkich pól i umieszczamy je w tablicy `ArrayList`. Ponieważ metoda `getFields` klasy `Class` nie zwraca pól klasy bazowej, to musimy ją dodatkowo wywołać dla wszystkich klas bazowych danej klasy. Odpowiedni kod odnajdziemy w konstruktorze klasy `Variable`. Metoda `getFields` klasy `Variable` zwraca tablicę pól, natomiast metoda `toString` — łańcuch znaków opisujący węzeł drzewa. Opis ten zawiera zawsze typ i nazwę zmiennej. Jeśli zmienna jest typu podstawowego, to opis zawiera także jej wartość.

> Jeśli typem zmiennej jest tablica, to program nie wyświetla jej elementów. Nie jest to trudne zadanie i pozostawiamy je jako ćwiczenie dla czytelników.

Przejdźmy teraz do omówienia modelu drzewa. Pierwsze dwie jego metody są bardzo proste.

```
public Object getRoot()
{
   return root;
}

public int getChildCount(Object parent)
{
   return ((Variable)parent).getFields().size();
}
```

Metoda `getChild` zwraca nowy obiekt klasy `Variable` opisujący dane pole. Metody `getType` oraz `getName` klasy `Field` udostępniają typ i nazwę pola. Korzystając z mechanizmu refleksji, możemy odczytać wartość pola za pomocą wywołania `f.get(parentValue)`. Metoda ta może wyrzucić wyjątek `IllegalAccessException`. Ponieważ w konstruktorze udostępniamy wszystkie pola, to wyjątek ten nie powinien się pojawić.

Poniżej przedstawiamy kompletny kod metody `getChild`.

```
public Object getChild(Object parent, int index)
{
   ArrayList fields = ((Variable)parent).getFields();
   Field f = (Field)fields.get(index);
   Object parentValue = ((Variable)parent).getValue();
   try
   {
      return new Variable(f.getType(), f.getName(),
         f.get(parentValue));
   }
   catch(IllegalAccessException e)
   {
      return null;
    }
}
```

Powyższe trzy metody udostępniają strukturę drzewa komponentowi klasy `JTree`. Implementacja pozostałych metod jest rutynowa (patrz listing 6.8).

Drzewo w naszym przykładzie jest strukturą *nieskończoną*. Możemy to sprawdzić, dokonując inspekcji jednego z obiektów typu `WeakReference`. Jeśli wybierzemy jego zmienną o nazwie `referent`, to powrócimy do wyjściowego obiektu. Jego poddrzewo możemy w ten sposób rozwijać w nieskończoność. Oczywiście program nie przechowuje nieskończonego zbioru węzłów, a jedynie tworzy je na żądanie, gdy użytkownik rozwija kolejne poddrzewa.

Przykład ten kończy omawianie tematyki drzew. Przejdziemy teraz do tematu tabel, kolejnego złożonego komponentu biblioteki Swing. Koncepcyjnie drzewa i tabele nie mają wiele wspólnego, ale w obu przypadkach wykorzystywane są te same mechanizmy modelu danych i rysowania komórek.

Listing 6.8. *ObjectInspectorTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
   Program demonstrujący wykorzystanie własnego modelu drzewa.
   Wyświetla pola obiektów.
*/
public class ObjectInspectorTest
{
   public static void main(String[] args)
```

```
   {
      JFrame frame = new ObjectInspectorFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca drzewo.
*/
class ObjectInspectorFrame extends JFrame
{
   public ObjectInspectorFrame()
   {
      setTitle("ObjectInspectorTest");
      setSize(WIDTH, HEIGHT);

      // jako pierwszy inspekcji poddany jest obiekt ramki

      Variable v = new Variable(getClass(), "this", this);
      ObjectTreeModel model = new ObjectTreeModel();
      model.setRoot(v);

      // tworzy i prezentuje drzewo

      tree = new JTree(model);
      getContentPane().add(new JScrollPane(tree),
         BorderLayout.CENTER);
   }

   private JTree tree;
   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;
}

/**
   Model drzewa opisującego strukturę powiązań obiektów w języku Java.
   Węzły podrzędne reprezentują zmienne składowe obiektu.
*/
class ObjectTreeModel implements TreeModel
{
   /**
      Tworzy puste drzewo.
   */
   public ObjectTreeModel()
   {
      root = null;
   }

   /**Umieszcza zmienną w korzeniu drzewa.
      @param v zmienna opisywana przez drzewo
   */
   public void setRoot(Variable v)
   {
      Variable oldRoot = v;
      root = v;
      fireTreeStructureChanged(oldRoot);
   }

   public Object getRoot()
```

```
{
   return root;
}

public int getChildCount(Object parent)
{
   return ((Variable)parent).getFields().size();
}

public Object getChild(Object parent, int index)
{
   ArrayList fields = ((Variable)parent).getFields();
   Field f = (Field)fields.get(index);
   Object parentValue = ((Variable)parent).getValue();
   try
   {
      return new Variable(f.getType(), f.getName(),
         f.get(parentValue));
   }
   catch(IllegalAccessException e)
   {
      return null;
   }
}

public int getIndexOfChild(Object parent, Object child)
{
   int n = getChildCount(parent);
   for (int i = 0; i < n; i++)
      if (getChild(parent, i).equals(child))
         return i;
   return -1;
}

public boolean isLeaf(Object node)
{
   return getChildCount(node) == 0;
}

public void valueForPathChanged(TreePath path,
   Object newValue)
{}

public void addTreeModelListener(TreeModelListener l)
{
   listenerList.add(TreeModelListener.class, l);
}

public void removeTreeModelListener(TreeModelListener l)
{
   listenerList.remove(TreeModelListener.class, l);
}

protected void fireTreeStructureChanged(Object oldRoot)
{
   TreeModelEvent event
      = new TreeModelEvent(this, new Object[] {oldRoot});
   EventListener[] listeners = listenerList.getListeners(
      TreeModelListener.class);
```

```
      for (int i = 0; i < listeners.length; i++)
         ((TreeModelListener)listeners[i]).treeStructureChanged(
            event);
   }

   private Variable root;
   private EventListenerList listenerList
      = new EventListenerList();
}

/**
   Klasa reprezentująca zmienną posiadającą typ, nazwę i wartość.
*/
class Variable
{
   /**
      Tworzy obiekt reprezentujący zmienną.
      @param aType typ zmiennej
      @param aName nazwa zmiennej
      @param aValue wartość zmiennej
   */
   public Variable(Class aType, String aName, Object aValue)
   {
      type = aType;
      name = aName;
      value = aValue;
      fields = new ArrayList();

      /*
         znajduje wszystkie pola, jeśli zmienna jest typu klasy,
         nie rozwijając jedynie łańcuchów znaków i wartości null
      */

      if (!type.isPrimitive() && !type.isArray() &&
         !type.equals(String.class) && value != null)
      {
         // pobiera pola klasy i pola wszystkich jej klas bazowych
         for (Class c = value.getClass(); c != null;
            c = c.getSuperclass())
         {
            Field[] f = c.getDeclaredFields();
            AccessibleObject.setAccessible(f, true);

            // pobiera wszystkie pola, które nie są statyczne
            for (int i = 0; i < f.length; i++)
               if ((f[i].getModifiers() & Modifier.STATIC) == 0)
                  fields.add(f[i]);
         }
      }
   }

   /**
      Zwraca wartość zmiennej.
      @return wartość
   */
   public Object getValue()
   {
      return value;
   }
```

```
   /**
      Zwraca wszystkie pola zmiennej, które nie są statyczne.
      @return tablica zmiennych opisujących pola
   */
   public ArrayList getFields()
   {
      return fields;
   }

   public String toString()
   {
      String r = type + " " + name;
      if (type.isPrimitive())
         r += "=" + value;
      else if (type.equals(String.class))
         r += "=" + value;
      else if (value == null)
         r += "=null";
      return r;
   }

   private Class type;
   private String name;
   private Object value;
   private ArrayList fields;
}
```

javax.swing.tree.TreeModel

- `Object getRoot` **zwraca korzeń drzewa**.
- `int getChildCount(Object parent)` **zwraca liczbę węzłów podrzędnych węzła** `parent`.
- `Object getChild(Object parent, int index)` **zwraca węzeł podrzędny węzła** `parent` **o danym indeksie**.
- `int getIndexOfChild(Object parent, Object child)` **zwraca indeks węzła** `child`. **Węzeł ten musi być węzłem podrzędnym węzła** `parent`.
- `boolean isLeaf(Object node)` **zwraca wartość** `true`, **jeśli węzeł** `node` **jest koncepcyjnym liściem**.
- `void addTreeModelListener(TreeModelListener l)`
- `void removeTreeModelListener(TreeModelListener l)`

  Dodają i usuwają obiekty nasłuchujące powiadamiane w momencie zmiany danych modelu.

- `void valueForPathChanged(TreePath path, Object newValue)` **metoda wywoływana, gdy edytor komórki zmodyfikował węzeł**.

  Parametry: `path` **ścieżka do zmodyfikowanego węzła,**

  `newValue` **nowa wartość zwrócona przez edytor**.

javax.swing.event.TreeModelListener

- `void treeNodesChanged(TreeModelEvent e)`
- `void treeNodesInserted(TreeModelEvent e)`
- `void treeNodesRemoved(TreeModelEvent e)`
- `void treeStructureChanged(TreeModelEvent e)`

  Wywoływane przez model drzewa, gdy jego dane uległy zmianie.

javax.swing.event.TreeModelEvent

- `TreeModelEvent(Object eventSource, TreePath node)` tworzy model zdarzeń drzewa.

  Parametry: `eventSource` model drzewa generujący zdarzenia,
  `node` ścieżka do węzła, który został zmodyfikowany.

Tabele

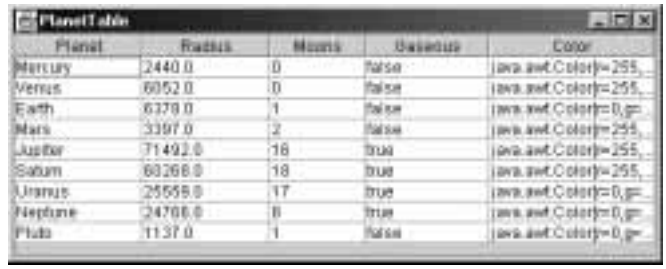
Klasa komponentu `JTable` wyświetla dwuwymiarową siatkę obiektów. Tabele stanowią często wykorzystywany komponent interfejsu użytkownika. Projektanci biblioteki Swing włożyli wiele wysiłku w uniwersalne zaprojektowanie komponentu tabel. Tabele są skomplikowanym komponentem, ale w ich przypadku projektantom klasy `JTable` udało się ukryć tę złożoność. W pełni funkcjonalne tabele o dużych możliwościach tworzymy już za pomocą kilku linijek kodu. Oczywiście kod ten możemy rozbudowywać, dostosowując wygląd i zachowanie tabeli do naszych specyficznych potrzeb.

W podrozdziale tym przedstawimy sposób tworzenia najprostszych tabel, ich interakcje z użytkownikiem i najczęstsze modyfikacje komponentu. Podobnie jak w przypadku innych złożonych komponentów biblioteki Swing, omówienie wszystkich aspektów korzystania z tabel przekracza możliwości tego rozdziału. Więcej informacji na ten temat znaleźć można w książce *Core Java Foundation Classes* autorstwa Kim Topley lub *Graphic Java 2* napisanej przez Davida M. Geary'ego.

Najprostsze tabele

Podobnie jak komponent drzewa, także i klasa `JTable` nie przechowuje danych tabeli, ale uzyskuje je, korzystając z *modelu tabeli*. Klasa `JTable` dysponuje konstruktorem, który umożliwia obudowanie dwuwymiarowej tablicy obiektów za pomocą domyślnego modelu. Z takiego rozwiązania skorzystamy w naszym pierwszym przykładzie. W dalszej części rozdziału zajmiemy się innymi modelami tabel.

Rysunek 6.25 przedstawia typową tabelę opisującą cechy planet układu słonecznego. (Planeta posiada cechę *gaseous*, jeśli składa się w większości z wodoru i helu. Kolumnę *Color* wykorzystamy dopiero w kolejnych przykładach programów).



Rysunek 6.25. *Przykład najprostszej tabeli*

Jak można zauważyć, analizując kod programu zawarty w listingu 6.9, dane tabeli przechowywane są za pomocą dwuwymiarowej tablicy obiektów:

```
private Object[][] cells =
{
   {
      "Mercury", new Double(2440),  new Integer(0),
      Boolean.FALSE, Color.yellow
   },
   {
      "Venus", new Double(6052), new Integer(0),
      Boolean.FALSE, Color.yellow
   },
   . . .
}
```

Tabela wywołuje metodę `toString` każdego z tych obiektów i wyświetla uzyskany rezultat. Wyjaśnia to między innymi sposób prezentacji danych w kolumnie *Color* w postaci `java.wat.Color[r=...,g=...,b=...]`.

Nazwy kolumn tabeli przechowywane są w osobnej tablicy łańcuchów znaków:

```
private String[] columnNames =
{
   "Planet", "Radius", "Moons", "Gaseous", "Color"
};
```

Korzystając z obu przedstawionych wyżej tablic, tworzymy tabelę. Umożliwiamy następnie jej przewijanie, obudowując ją panelem klasy `JScrollPane`.

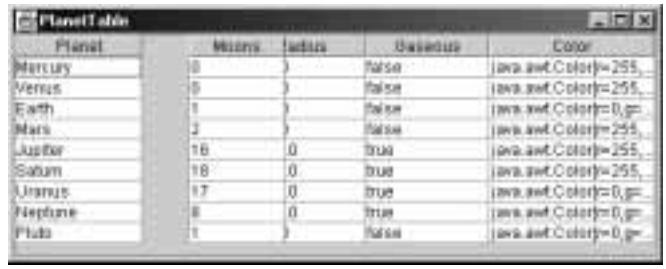
```
JTable table = new JTable(cells, columnNames);
JScrollPane pane = new JScrollPane(table);
```

Otrzymana tabela posiada zaskakująco bogate możliwości. Zmniejszymy jej rozmiar tak, by pokazały się paski przewijania. Zwróćmy uwagę, że podczas przewijania zawartości tabeli nazwy kolumn pozostają na właściwym miejscu!

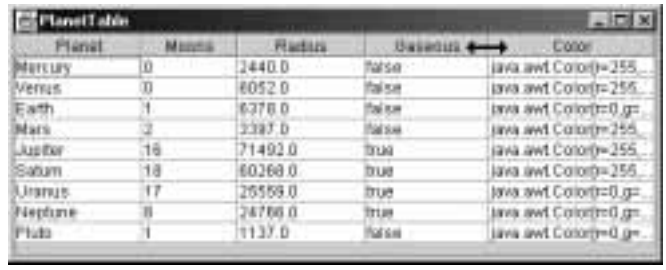
Następnie wybierzmy jeden z nagłówków kolumn i przeciągnijmy go w lewo lub w prawo. Spowoduje to przesunięcie całej kolumny (patrz rysunek 6.26), którą umieścić możemy w innym miejscu tabeli. Zmiana ta dotyczy jedynie *widoku* tabeli i pozostaje bez wpływu na dane modelu.

Aby zmienić *szerokość* kolumny, wystarczy umieścić kursor myszy na linii oddzielającej kolumny. Kursor przybierze wtedy kształt strzałki, umożliwiając przesunięcie linii oddzielającej kolumny (patrz rysunek 6.27).

Rysunek 6.26.
Przesunięcie kolumny



Rysunek 6.27.
Zmiana szerokości kolumn



Użytkownik może wybierać wiersze tabeli za pomocą myszy. Wybrane wiersze zostają podświetlone. Sposób obsługi takiego zdarzenia pokażemy w dalszej części rozdziału. Użytkownik może także wybrać komórkę tabeli i zmodyfikować ją. W bieżącym przykładzie modyfikacja ta nie spowoduje jeszcze zmiany danych modelu. W praktyce w programach należy wyłączyć możliwość edycji komórek tabeli lub obsługiwać zdarzenia edycji i odpowiednio modyfikować dane modelu. Zagadnienia te omówimy nieco później.

Listing 6.9. *PlanetTable.java*

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

/**
   Program demonstrujący przykład prostej tabeli.
*/
public class PlanetTable
{
   public static void main(String[] args)
   {
      JFrame frame = new PlanetTableFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca tabelę danych planet.
*/
class PlanetTableFrame extends JFrame
{
   public PlanetTableFrame()
   {
```

```
      setTitle("PlanetTable");
      setSize(WIDTH, HEIGHT);

      JTable table = new JTable(cells, columnNames);

      getContentPane().add(new JScrollPane(table),
         BorderLayout.CENTER);
   }

   private Object[][] cells =
   {
      {
         "Mercury", new Double(2440),  new Integer(0),
         Boolean.FALSE, Color.yellow
      },
      {
         "Venus", new Double(6052), new Integer(0),
         Boolean.FALSE, Color.yellow
      },
      {
         "Earth", new Double(6378), new Integer(1),
         Boolean.FALSE, Color.blue
      },
      {
         "Mars", new Double(3397), new Integer(2),
         Boolean.FALSE, Color.red
      },
      {
         "Jupiter", new Double(71492), new Integer(16),
         Boolean.TRUE, Color.orange
      },
      {
         "Saturn", new Double(60268), new Integer(18),
         Boolean.TRUE, Color.orange
      },
      {
         "Uranus", new Double(25559), new Integer(17),
         Boolean.TRUE, Color.blue
      },
      {
         "Neptune", new Double(24766), new Integer(8),
         Boolean.TRUE, Color.blue
      },
      {
         "Pluto", new Double(1137), new Integer(1),
         Boolean.FALSE, Color.black
      }
   };

   private String[] columnNames =
   {
      "Planet", "Radius", "Moons", "Gaseous", "Color"
   };

   private static final int WIDTH = 400;
   private static final int HEIGHT = 200;
}
```

javax.swing.JTable

- `JTable(Object[][] entries, Object[] columnNames)` **tworzy tabelę, wykorzystując domyślny model.**

  Parametry: `entries` **komórki tabeli,**

  `columnNames` **nazwy (tytuły) kolumn tabeli.**

Modele tabel

W poprzednim przykładzie obiekty tabeli przechowywane były za pomocą dwuwymiarowej tablicy. Rozwiązanie takie nie jest jednak zalecane w każdym przypadku. Jeśli kod nasz umieszcza dane w tablicy, aby zaprezentować je następnie w postaci tabeli, oznacza to, że powinniśmy zastanowić się nad implementacją własnego modelu tabeli.

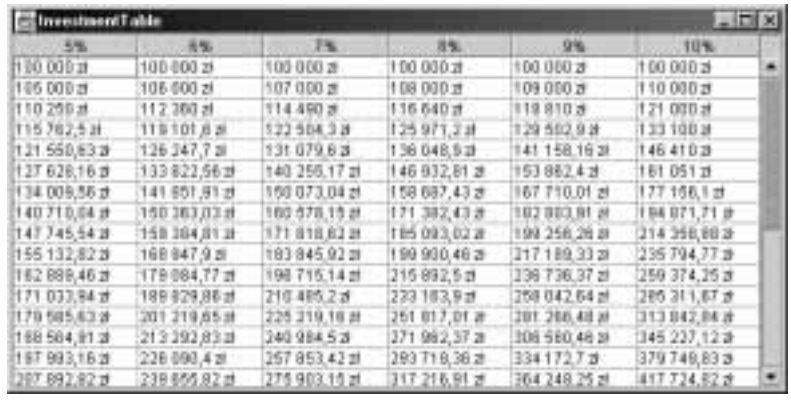
Implementacja własnego modelu tabeli nie jest trudna, ponieważ wykorzystać możemy klasę `AbstractTableModel`, która dostarcza implementacji większości potrzebnych metod. Sami zaimplementować musimy jedynie trzy poniższe metody:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Istnieje wiele sposobów implementacji metody `getValueAt`. Możemy po prostu wyliczyć odpowiednią wartość na żądanie lub pobrać konkretną wartość z bazy danych. Przyjrzyjmy się kilku przykładom.

W pierwszym z nich tabela zawierać będzie wartości, które wyliczy program. Będą one przedstawiać wzrost inwestycji w różnych scenariuszach (patrz rysunek 6.28).

Rysunek 6.28. *Tabela reprezentująca wzrost inwestycji*



Metoda `getValueAt` wyznacza odpowiednią wartość i formatuje ją:

```
public Object getValueAt(int r, int c)
{
   double rate = (c + minRate) / 100.0;
   int nperiods = r;
```

```
      double futureBalance = INITIAL_BALANCE
         * Math.pow(1 + rate, nperiods);

      return
         NumberFormat.getCurrencyInstance().format(futureBalance);
   }
```

Metody `getRowCount` i `getColumnCount` zwracają odpowiednio liczbę wierszy i kolumn tabeli.

```
   public int getRowCount()
   {
      return years;
   }

   public int getColumnCount()
   {
      return maxRate - minRate + 1;
   }
```

Jeśli nie podamy nazw kolumn, to metoda `getColumnName` klasy `AbstractTableModel` nazwie kolejne kolumny A, B, C itd. Aby zmienić nazwy kolumn, zastąpimy metodę `getColumnName` i wykorzystamy procentowy przyrost inwestycji jako nazwę kolumn.

```
   public String getColumnName(int c)
   {
      double rate = (c + minRate) / 100.0;
      return
         NumberFormat.getPercentInstance().format(rate);
   }
```

Kompletny kod źródłowy programu zawiera listing 6.10.

Listing 6.10. *InvestmentTable.java*

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

/**
   Program tworzący tabelę w oparciu o własny model.
*/
public class InvestmentTable
{
   public static void main(String[] args)
   {
      JFrame frame = new InvestmentTableFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca tabelę inwestycji.
*/
class InvestmentTableFrame extends JFrame
```

```
{
   public InvestmentTableFrame()
   {
      setTitle("InvestmentTable");
      setSize(WIDTH, HEIGHT);

      TableModel model = new InvestmentTableModel(30, 5, 10);
      JTable table = new JTable(model);
      getContentPane().add(new JScrollPane(table), "Center");
   }

   private static final int WIDTH = 600;
   private static final int HEIGHT = 300;
}

/**
   Model tabeli wyliczający wartości komórek na żądanie.
   Tabela pokazuje przyrost inwestycji w kolejnych latach
   w różnych scenariuszach.
*/
class InvestmentTableModel extends AbstractTableModel
{
   /**
      Tworzy model tabeli inwestycji.
      @param y liczba lat
      @param r1 najniższa stopa procentowa
      @param r2 najwyższa stopa procentowa
   */
   public InvestmentTableModel(int y, int r1, int r2)
   {
      years = y;
      minRate = r1;
      maxRate = r2;
   }

   public int getRowCount()
   {
      return years;
   }

   public int getColumnCount()
   {
      return maxRate - minRate + 1;
   }

   public Object getValueAt(int r, int c)
   {
      double rate = (c + minRate) / 100.0;
      int nperiods = r;

      double futureBalance = INITIAL_BALANCE
         * Math.pow(1 + rate, nperiods);

      return
         NumberFormat.getCurrencyInstance().format(futureBalance);
   }

   public String getColumnName(int c)
```

```
    {
       double rate = (c + minRate) / 100.0;
       return
          NumberFormat.getPercentInstance().format(rate);
    }

    private int years;
    private int minRate;
    private int maxRate;

    private static double INITIAL_BALANCE = 100000.0;
}
```

Prezentacja rekordów bazy danych

Prawdopodobnie najczęściej reprezentowaną przez komponent tabeli informacją jest zbiór rekordów pochodzących z bazy danych. Korzystając ze środowiska tworzenia aplikacji, dysponujemy prawie zawsze gotowymi komponentami JavaBeans dla wykorzystania bazy danych. Warto jednak zobaczyć, w jaki sposób sami prezentować możemy za pomocą tabeli dane z bazy i temu zadaniu służy kolejny przykład. Rysunek 6.29 pokazuje efekt jego działania — wynik zapytania o wszystkie rekordy wybranej tabeli bazy danych.

Rysunek 6.29. *Prezentacja wyniku zapytania za pomocą tabeli*

Przykładowy program definiuje własny model danych za pomocą klasy `ResultSetTableModel`, który pobiera dane będące wynikiem zapytania do bazy danych. (Rozdział 4. poświęcony jest dostępowi do baz danych w języku Java).

Liczbę kolumn i ich nazwy uzyskujemy, korzystając z obiektu `rsmd` reprezentującego metadane opisujące zbiór rekordów:

```
public String getColumnName(int c)
{
   try
   {
      return rsmd.getColumnName(c + 1);
   }
   catch(SQLException e)
   {
      . . .
   }
}

public int getColumnCount()
{
   try
   {
      return rsmd.getColumnCount();
      }
   catch(SQLException e)
   {
      . . .
   }
}
```

Jeśli baza danych dysponuje przewijalnymi kursorami, to wartość komórki możemy uzyskać bardzo łatwo, przesuwając kursor do danego wiersza i pobierając wartość kolumny.

```
public Object getValueAt(int r, int c)
{
   try
   {
      ResultSet rs = getResultSet();
      rs.absolute(r + 1);
      return rs.getObject(c + 1);
   }
   catch(SQLException e)
   {
     . . .
   }
}
```

Wykorzystanie w tym przypadku własnego modelu danych zamiast domyślnego modelu `DefaultTableModel` posiada szczególny sens. Jeśli utworzylibyśmy własną tablicę wartości, to niepotrzebnie duplikowalibyśmy zawartość bufora sterownika bazy danych.

Jeśli baza danych nie dysponuje przewijalnymi kursorami lub korzystamy ze sterownika zgodnego ze specyfikacją JDBC 1, to sami musimy buforować wynik zapytania. Program ukazany w przykładzie umożliwia zarządzanie takim buforem. Klasa `CachingResultSetTableModel` buforuje wynik zapytania, natomiast klasa `ScrollingResultSetTableModel` wykorzystuje przewijalny kursor. Funkcjonalność wspólną dla obu klas wyodrębniliśmy w klasie `ResultSetTableModel`.

Listing 6.11. *ResultSetTable.java*

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.sql.*;
```

```
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

/**
   Program reprezentujący wynik zapytania do bazy danych
   za pomocą komponentu tabeli.
*/
public class ResultSetTable
{
   public static void main(String[] args)
   {
      JFrame frame = new ResultSetFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca listę rozwijalną umożliwiającą wybór tabeli
   bazy danych i tabelę reprezentującą jej rekordy
*/
class ResultSetFrame extends JFrame
{
   public ResultSetFrame()
   {
      setTitle("ResultSet");
      setSize(WIDTH, HEIGHT);

      /*
         znajduje wszystkie tabele bazy danych
         i umieszcza je na liście rozwijalnej
      */

      Container contentPane = getContentPane();
      tableNames = new JComboBox();
      tableNames.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent evt)
            {
               try
               {
                  if (scrollPane != null)
                     getContentPane().remove(scrollPane);
                  String tableName
                     = (String)tableNames.getSelectedItem();
                  if (rs != null) rs.close();
                  String query = "SELECT * FROM " + tableName;
                  rs = stat.executeQuery(query);
                  if (scrolling)
                     model
                        = new ScrollingResultSetTableModel(rs);
                  else
                     model = new CachingResultSetTableModel(rs);

                  JTable table = new JTable(model);
                  scrollPane = new JScrollPane(table);
```

```
            getContentPane().add(scrollPane,
               BorderLayout.CENTER);
            pack();
            doLayout();
         }
         catch(SQLException e)
         {
            e.printStackTrace();
         }
      }
   });
JPanel p = new JPanel();
p.add(tableNames);
contentPane.add(p, BorderLayout.NORTH);

try
{
   conn = getConnection();
   DatabaseMetaData meta = conn.getMetaData();
   if (meta.supportsResultSetType(
      ResultSet.TYPE_SCROLL_INSENSITIVE))
   {
      scrolling = true;
      stat = conn.createStatement(
         ResultSet.TYPE_SCROLL_INSENSITIVE,
         ResultSet.CONCUR_READ_ONLY);
   }
   else
   {
      stat = conn.createStatement();
      scrolling = false;
   }
   ResultSet tables = meta.getTables(null, null, null,
      new String[] { "TABLE" });
   while (tables.next())
      tableNames.addItem(tables.getString(3));
   tables.close();
}
catch(IOException e)
{
   e.printStackTrace();
}
catch(SQLException e)
{
   e.printStackTrace();
}

addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent event)
      {
         try
         {
            conn.close();
         }
         catch (SQLException e)
```

```
                {
                   e.printStackTrace();
                }
             }
          });
   }

   /**
      Tworzy połączenie do bazy danych, korzystając
      z właściwości zapisanych w pliku database.properties.
      @return połączenie do bazy danych
    */
   public static Connection getConnection()
      throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in
         = new FileInputStream("database.properties");
      props.load(in);
      in.close();

      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null)
         System.setProperty("jdbc.drivers", drivers);
      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return
         DriverManager.getConnection(url, username, password);
   }

   private JScrollPane scrollPane;
   private ResultSetTableModel model;
   private JComboBox tableNames;
   private ResultSet rs;
   private Connection conn;
   private Statement stat;
   private boolean scrolling;

   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;
}

/**
   Klasa bazowa modeli tabel korzystających z przewijalnych
   kursorów lub buforujących wynik zapytania.
   Przechowuje wynik zapytania i jego metadane.
*/
abstract class ResultSetTableModel extends AbstractTableModel
{
   /**
      Tworzy model tabeli.
      @param aResultSet zbiór rekordów.
   */
   public ResultSetTableModel(ResultSet aResultSet)
   {
      rs = aResultSet;
      try
```

```
         {
            rsmd = rs.getMetaData();
         }
         catch(SQLException e)
         {
            e.printStackTrace();
         }
      }

      public String getColumnName(int c)
      {
         try
         {
            return rsmd.getColumnName(c + 1);
         }
         catch(SQLException e)
         {
            e.printStackTrace();
            return "";
         }
      }

      public int getColumnCount()
      {
         try
         {
            return rsmd.getColumnCount();
         }
         catch(SQLException e)
         {
            e.printStackTrace();
            return 0;
         }
      }

      /**
         Pobiera zbiór rekordów.
         @return zbiór rekordów
      */
      protected ResultSet getResultSet()
      {
         return rs;
      }

      private ResultSet rs;
      private ResultSetMetaData rsmd;
   }

   /**
      Klasa wykorzystująca przewijalne kursory
      dostępne w JDBC 2.
   */
   class ScrollingResultSetTableModel extends ResultSetTableModel
   {
      /**
         Tworzy model tabeli.
         @param aResultSet zbiór rekordów.
      */
```

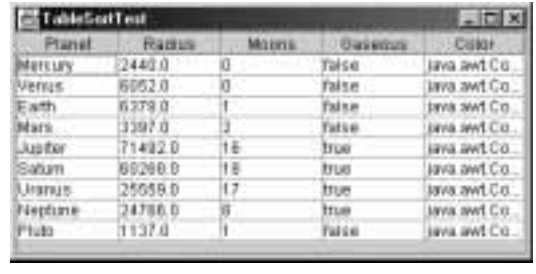
```
   public ScrollingResultSetTableModel(ResultSet aResultSet)
   {
      super(aResultSet);
   }

   public Object getValueAt(int r, int c)
   {
      try
      {
         ResultSet rs = getResultSet();
         rs.absolute(r + 1);
         return rs.getObject(c + 1);
      }
      catch(SQLException e)
      {
         e.printStackTrace();
         return null;
      }
   }

   public int getRowCount()
   {
      try
      {
         ResultSet rs = getResultSet();
         rs.last();
         return rs.getRow();
      }
      catch(SQLException e)
      {
         e.printStackTrace();
         return 0;
      }
   }
}

/*
   Klasa buforująca zbiór rekordów będący wynikiem zapytania.
   Używana, gdy kursory przewijalne są niedostępne.
*/
class CachingResultSetTableModel extends ResultSetTableModel
{
   public CachingResultSetTableModel(ResultSet aResultSet)
   {
      super(aResultSet);
      try
      {
         cache = new ArrayList();
         int cols = getColumnCount();
         ResultSet rs = getResultSet();

         /*
            Umieszcza dane w tablicy typu array list,
             której elementami są tablice typu Object[].
            Nie możemy wykorzystać tablicy Object[][],
            ponieważ nie znamy liczby rekordów.
         */
```

```
            while (rs.next())
            {
               Object[] row = new Object[cols];
               for (int j = 0; j < row.length; j++)
                  row[j] = rs.getObject(j + 1);
               cache.add(row);
            }
         }
         catch(SQLException e)
         {
            System.out.println("Error " + e);
         }
      }

      public Object getValueAt(int r, int c)
      {
         if (r < cache.size())
            return ((Object[])cache.get(r))[c];
         else
            return null;
      }

      public int getRowCount()
      {
         return cache.size();
      }

      private ArrayList cache;
   }
```

Filtry sortujące

Dwa ostatnie przykłady wykazały, że tabele nie przechowują pokazywanych za ich pomocą danych, lecz pobierają je, korzystając z modelu. Także i model nie musi przechowywać danych — może wyliczać ich wartości na żądania lub pobierać je z bazy danych.

Wprowadzimy teraz kolejny przydatny mechanizm zwany *modelem filtra*, który umożliwia prezentację informacji danej tablicy w innej formie. W naszym przykładzie forma ta polegać będzie na posortowaniu wierszy tabeli. Po uruchomieniu programu, którego tekst źródłowy zawiera listing 6.12, spróbujmy kliknąć dwukrotnie jedną z kolumn tabeli. Spowoduje to uporządkowanie tabeli według wartości wybranej kolumny (patrz rysunek 6.30).



Rysunek 6.30. *Sortowanie wierszy tabeli*

Sortowanie tabeli nie powoduje jednak uporządkowania danych modelu. Za posortowanie danych odpowiedzialny jest bowiem model filtra.

Przechowuje on referencję do modelu tabeli. Gdy komponent tabeli pobiera wiersz do prezentacji, to model filtra wyznacza rzeczywisty wiersz tabeli i pobiera go z modelu tabeli. Oto przykład

```
public Object getValueAt(int r, int c)
{
   return model.getValue( rzeczywisty indeks wiersza, c);
}
```

Wywołania pozostałych metod przekazywane są po prostu do oryginalnego modelu tabeli.

```
public String getColumnName(int c)
{
    return model. getColumnName(c);
}
```

Rysunek 6.31 pokazuje sposób, w jaki model filtra współdziała z obiektem klasy `JTable` i rzeczywistym modelem tabeli.

Rysunek 6.31.
Model filtra tabeli

Z implementacją takiego filtra związane są dwa zagadnienia. Po pierwsze, gdy użytkownik kliknie dwukrotnie jedną z kolumn, to model filtra musi otrzymać informację o tym. Nie będziemy omawiać związanych z tym szczegółów implementacji. Odpowiedni kod odnajdziemy wewnątrz metody `addMouseListener` klasy `SortFilterModel` w tekście listingu 6.12. Działa on w następujący sposób. Najpierw pobieramy komponent nagłówka tabeli i dodajemy do niego obiekt nasłuchujący zdarzeń związanych z myszą. Kiedy wykryje on dwukrotne kliknięcie, musi uzyskać informację o kolumnie, której ono dotyczy. Następnie przełożyć kolumnę komponentu tabeli na kolumnę modelu, ponieważ użytkownik mógł je poprzestawiać. Znając kolumnę, może rozpocząć sortowanie danych.

Z sortowaniem danych związane jest kolejne zagadnienie. Ponieważ nie chcemy sortować danych oryginalnego modelu tabeli, to musimy uzyskać sekwencję indeksów wierszy, która pozwoli na ich prezentację przez komponent w uporządkowanej kolejności. Jednak algorytmy sortowania dostępne w klasach `Arrays` i `Collections` nie udostępnią nam takiej informacji. Możemy oczywiście sami zaimplementować algorytm sortowania, który pozwoli śledzić sposób uporządkowania obiektów. Istnieje jednak sprytne rozwiązanie tego problemu. Polega ono na dostarczeniu własnych obiektów i własnej metody porównania bibliotecznym algorytmom sortowania.

Sortować będziemy obiekty typu `Row`. Obiekt taki zawiera indeks `r` wiersza w modelu. Dwa takie obiekty porównywać będziemy następująco: odnajdziemy je w modelu i porównamy. Innymi słowy metoda `compareTo` dla obiektów klasy `Row` zwracać będzie wynik następującego porównania:

```
model.getValueAt(r1, c).compareTo(model. getValueAt(r2, c))
```

gdzie `r1` i `r2` są indeksami obiektów klasy `Row`, a `c` jest kolumną, według której sortowana jest tabela.

Jeśli wartości danej kolumny nie można porównać, to porównujemy reprezentujące je łańcuchy znaków. Tym sposobem możemy posortować tabelę także według kolumn zawierających wartości logiczne lub definicje kolorów (choć żadna z reprezentujących je klas nie implementuje interfejsu `Comparable`).

Klasę `Row` zdefiniujemy jako klasę wewnętrzną klasy `SortFilterModel`, ponieważ metoda `compareTo` klasy `Row` potrzebuje dostępu do bieżącej kolumny modelu. Poniżej przedstawiamy odpowiedni kod:

```
class SortFilterModel extends AbstractTableModel
{ . . .
   private class Row implements Comparable
   {
      public int index;
      public int compareTo(Object other)
      {
         Row otherRow = (Row)other;
         Object a = model.getValueAt(index, sortColumn);
         Object b = model.getValueAt(otherRow.index, sortColumn);
         if (a instanceof Comparable)
            return ((Comparable)a).compareTo(b);
         else
            return a.toString().compareTo(b.toString());
         }
   }

   private TableModel model;
   private int sortColumn;
   private Row[] rows;
}
```

W konstruktorze tworzymy tablicę `rows`, którą inicjujemy w taki sposób, że `rows[i]=i`:

```
public SortFilterModel(TableModel m)
{
   model = m;
   rows = new Row[model.getRowCount()];
   for (int i = 0; i < rows.length; i++)
   {
      rows[i] = new Row();
      rows[i].index = i;
   }
}
```

Metoda `sort` korzysta z algorytmu metody `Arrays.sort` do sortowania obiektów klasy `Row`. Ponieważ metoda porównania korzysta z elementów odpowiedniej kolumny, to elementy te są uporządkowane w ten sposób, że `row[0]` zawiera indeks najmniejszego elementu w kolumnie, `row[1]` następnego najmniejszego elementu itd.

Gdy tablica jest sortowana, to powiadamiamy wszystkie obiekty nasłuchujące zmian w modelu (w szczególności `JTable`), że zawartość tabeli uległa zmianie i musi zostać narysowana od nowa.

```
public void sort(int c)
{
   sortColumn = c;
```

```
      Arrays.sort(rows);
      fireTableDataChanged();
   }
```

Poniżej prezentujemy także implementację metody `getValueAt` klasy filtra. Tłumaczy ona wartość indeksu `r` na wartość indeksu modelu `rows[r].index`:

```
public Object getValueAt(int r, int c)
{
   return model.getValueAt(rows[r].index, c);
}
```

Model filtra sortującego jest kolejnym udanym przykładem zastosowania wzorca model-widok. Ponieważ oddziela on dane od sposobu ich prezentacji, to możemy dowolnie zmieniać ich wzajemne odwzorowanie.

Listing 6.12. *TableSortTest.java*

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

/**
   Program demonstrujący sortowanie tabeli według wybranej kolumny.
   W celu przesortowania tabeli należy dwukrotnie kliknąć jedną z kolumn.
*/
public class TableSortTest
{
   public static void main(String[] args)
   {
      JFrame frame = new TableSortFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca tabelę danych o planetach.
*/
class TableSortFrame extends JFrame
{
   public TableSortFrame()
   {
      setTitle("TableSortTest");
      setSize(WIDTH, HEIGHT);

      // tworzy model tabeli oraz model filtra

      DefaultTableModel model
         = new DefaultTableModel(cells, columnNames);
      final SortFilterModel sorter = new SortFilterModel(model);

      // pokazuje tabelę

      final JTable table = new JTable(sorter);
```

```
      getContentPane().add(new JScrollPane(table),
         BorderLayout.CENTER);

      // dodaje obiekt nasłuchujący dwukrotnych kliknięć
      // myszy w nagłówku tabeli

      table.getTableHeader().addMouseListener(new
         MouseAdapter()
         {
            public void mouseClicked(MouseEvent event)
            {
               // czy dwukrotne kliknięcie?
               if (event.getClickCount() < 2) return;

               // sprawdza, dla której kolumny
               int tableColumn
                  = table.columnAtPoint(event.getPoint());

               // zamienia indeks kolumny na indeks kolumny w modelu
               // i sortuje ją
               int modelColumn
                  = table.convertColumnIndexToModel(tableColumn);
               sorter.sort(modelColumn);
            }
         });
   }

   private Object[][] cells =
   {
      {
         "Mercury", new Double(2440),  new Integer(0),
         Boolean.FALSE, Color.yellow
      },

      {
         "Venus", new Double(6052), new Integer(0),
         Boolean.FALSE, Color.yellow
      },
      {
         "Earth", new Double(6378), new Integer(1),
         Boolean.FALSE, Color.blue
      },
      {
         "Mars", new Double(3397), new Integer(2),
         Boolean.FALSE, Color.red
      },
      {
         "Jupiter", new Double(71492), new Integer(16),
         Boolean.TRUE, Color.orange
      },
      {
         "Saturn", new Double(60268), new Integer(18),
         Boolean.TRUE, Color.orange
      },
      {
         "Uranus", new Double(25559), new Integer(17),
         Boolean.TRUE, Color.blue
      },
```

```
         {
            "Neptune", new Double(24766), new Integer(8),
            Boolean.TRUE, Color.blue
         },
         {
            "Pluto", new Double(1137), new Integer(1),
            Boolean.FALSE, Color.black
         }
      };

      private String[] columnNames =
      {
         "Planet", "Radius", "Moons", "Gaseous", "Color"
      };

      private static final int WIDTH = 400;
      private static final int HEIGHT = 200;
   }

   /**
      Model tabeli wykorzystujący oryginalny model tabeli
      i sortujący wiersze tabeli według wybranej kolumny.
   */
   class SortFilterModel extends AbstractTableModel
   {
      /**
         Tworzy model filtra sortującego.
         @param m wyjściowy model tabeli
      */
      public SortFilterModel(TableModel m)
      {
         model = m;
         rows = new Row[model.getRowCount()];
         for (int i = 0; i < rows.length; i++)
         {
            rows[i] = new Row();
            rows[i].index = i;
         }
      }

      /**
         Sortuje wiersze tabeli.
         @param c kolumna, według której odbywa się sortowanie
      */
      public void sort(int c)
      {
         sortColumn = c;
         Arrays.sort(rows);
         fireTableDataChanged();
      }

      // Dla poniższych metod musi najpierw zostać wyznaczony
      // odpowiedni wiersz

      public Object getValueAt(int r, int c)
      {
         return model.getValueAt(rows[r].index, c);
      }
```

```
   public boolean isCellEditable(int r, int c)
   {
      return model.isCellEditable(rows[r].index, c);
   }

   public void setValueAt(Object aValue, int r, int c)
   {
      model.setValueAt(aValue, rows[r].index, c);
   }

   // pozostałe wywołania metod delegowane są do modelu wyjściowego

   public int getRowCount()
   {
      return model.getRowCount();
   }

   public int getColumnCount()
   {
      return model.getColumnCount();
   }

   public String getColumnName(int c)
   {
      return model.getColumnName(c);
   }

   public Class getColumnClass(int c)
   {
      return model.getColumnClass(c);
   }

   /**
      Klasa wewnętrzna przechowująca indeks wiersza w modelu.
      Wiersze porównywane są za pomocą wartości modelu dla w kolumnie,
      według której sortowana jest tabela.
   */
   private class Row implements Comparable
   {
      public int index;
      public int compareTo(Object other)
      {
         Row otherRow = (Row)other;
         Object a = model.getValueAt(index, sortColumn);
         Object b = model.getValueAt(otherRow.index, sortColumn);
         if (a instanceof Comparable)
            return ((Comparable)a).compareTo(b);
         else
            return a.toString().compareTo(b.toString());

            //            return index - otherRow.index;
      }
   }

   private TableModel model;
   private int sortColumn;
   private Row[] rows;
}
```

javax.swing.table.TableModel

- `int getRowCount()`
- `int getColumnCount()`

  Zwracają liczbę wierszy i kolumn w modelu tabeli.
- `Object getValueAt(int row, int column)` zwraca wartość w danym wierszu i kolumnie.
- `void setValueAt(Object newValue, int row, int column)` nadaje wartość obiektowi w danym wierszu i kolumnie.
- `bolean isCellEditable(int row, int column)` zwraca wartość `true`, jeśli komórka w danym wierszu i kolumnie może być edytowana.
- `String getColumnName(int column)` zwraca nazwę (tytuł) kolumny.

javax.swing.table.AbstractTableModel

- `void fireTableDataChanged()` zawiadamia wszystkie obiekty nasłuchujące danego modelu tabeli o zmianie danych.

javax.swing.JTable

- `JTableHeader getTableheader()` zwraca komponent nagłówka danej tabeli.
- `int columnAtPoint(Point p)` zwraca numer kolumny tabeli, która zawiera piksel `p`.
- `int convertColumnIndexToModel(int tableColumn)` zwraca indeks kolumny w modelu dla danej kolumny w tabeli. Wartości te są różne, jeśli kolumny tabeli zostały poprzestawiane lub ukryte.

Rysowanie i edytowanie zawartości komórek

Kolejny przykład znowu będzie prezentował w tabeli dane o planetach, ale tym razem wyposażymy tabelę w informację o *typie kolumn*. Jeśli zdefiniujemy metodę

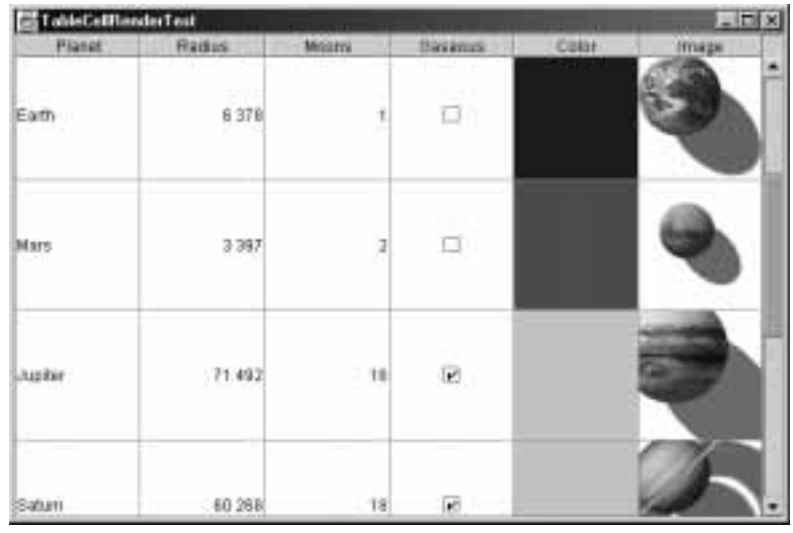
```
Class getColumnClass(int columnIndex)
```

modelu tabeli tak, by zwracała klasę opisującą typ kolumny, to klasa `JTable` będzie mogła wybrać właściwy *obiekt rysujący* dla danej klasy. Tabela 6.1 przedstawia sposób prezentacji kolumn różnych typów przez domyślne obiekty rysujące wykorzystywane przez klasę `JTable`.

Tabela 6.1. *Domyślne obiekty rysujące*

Typ	Reprezentacja w postaci
`ImageIcon`	obrazka
`Boolean`	pola wyboru
`Object`	łańcucha znaków

Pola wyboru i obrazki w komórkach tabeli zobaczyć możemy na rysunku 6.32 (dziękujemy w tym miejscu Jimowi Evinsowi, *http://www.snaught.com/JimCoolIcons/Planets*, za udostępnienie obrazków planet).



Rysunek 6.32. *Tabela wykorzystująca obiekty rysujące*

W przypadku innych typów dostarczyć możemy własnych obiektów rysujących. Przypominają one w działaniu obiekty rysujące komórki drzewa, które przedstawiliśmy, omawiając komponent drzewa. Implementują one interfejs `TableCellRenderer` posiadający pojedynczą metodę:

```
Component getTableCellRendererComponent(JTable table,
   Object value, boolean isSelected, boolean hasFocus,
   int row, int column)
```

Metoda ta wywoływana jest za każdym razem, kiedy komórka tabeli wymaga narysowania. Zwraca komponent, którego metoda `paint` wykorzystywana jest do narysowania komórki tabeli.

Aby wyświetlić komórkę typu `Color`, wystarczy, że zwrócimy panel, którego kolor tła ustawiony będzie zgodnie z kolorem określonym przez obiekt `Color` znajdujący się w komórce tabeli. Obiekt ten zostanie przekazany metodzie za pośrednictwem parametru `value`.

```
class ColorTableCellRenderer implements TableCellRenderer
{
   public Component getTableCellRendererComponent(JTable table,
      Object value, boolean isSelected, boolean hasFocus,
      int row, int column)
   {
      panel.setBackground((Color)value);
      return panel;
   }
   private JPanel panel = new JPanel();
}
```

Musimy jeszcze przekazać do tabeli informację, aby skorzystała z obiektu rysującego powyższej klasy w przypadku wszystkich komórek zawierających obiekty klasy `Color`. Użyjemy w tym celu metody `setDefaultRenderer` klasy `JTable`, której parametrami będą obiekt klasy `Class` i obiekt rysujący.

```
table.setDefaultRenderer(Color.class,
   new ColorTableCellRenderer());
```

Odtąd nowy obiekt rysujący będzie wykorzystywany dla obiektów danego typu.

> Często wykorzystuje się obiekty rysujące, które różnicują wygląd komórki w zależności od jej stanu (przeglądana, wybrana). W tym celu musimy dysponować rozmiarami komórki oraz schematem kolorów związanym ze stanami wybrania i przeglądania komponentów interfejsu.
>
> Aby uzyskać informację o rozmiarach komórki, skorzystać można z metody `getCellRect` klasy `JTable`. Kolory związane z wybraniem komponentów zwracają metody `getSelectionBackground` i `getSelectionForeground`.

> Jeśli obiekt rysujący wyświetla łańcuch znaków lub ikonę, to możemy go utworzyć, rozszerzając klasę `DefaultTableCellRenderer`, która wykona działania związane z obsługą stanu wyboru i przeglądania komórki.

Edycja komórek

Aby umożliwić edycję komórek, model tabeli musi definiować metodę `isCellEditable` wskazującą, czy dana komórka tabeli może być edytowana. Zwykle zezwala się raczej od razu na edycję całej kolumny niż poszczególnych komórek. W programie przykładowym pozwolimy na edycję komórek czterech kolumn tabeli.

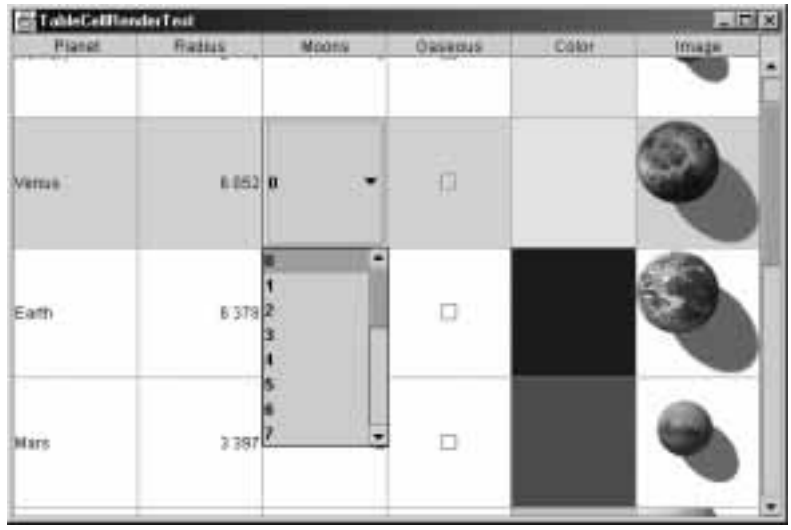
```
public boolean isCellEditable(int r, int c)
{
   return c == NAME_COLUMN
      || c == MOON_COLUMN
      || c == GASEOUS_COLUMN
      || c == COLOR_COLUMN;
}

public static final int NAME_COLUMN = 0;
public static final int MOON_COLUMN = 2;
public static final int GASEOUS_COLUMN = 3;
public static final int COLOR_COLUMN = 4;
```

> Klasa `AbstractTableModel` definiuje metodę `isCellEditable`, która zawsze zwraca wartość `false`. Klasa `DefaultTableModel` zastępuje ją z kolei implementacją, która zawsze zwraca wartość `true`.

Jeśli uruchomimy program, którego tekst źródłowy zawiera listing 6.13, to zauważymy, że w kolumnie *Gaseous* możemy edytować pole wyboru. Natomiast w kolumnie *Moons* możemy wybierać wartość z listy rozwijalnej (rysunek 6.33). Za chwilę pokażemy, w jaki sposób zainstalować listę rozwijalną jako edytor wartości komórki.

Wybierając komórki pierwszej kolumny tabeli, możemy zmieniać ich zawartość, wpisując dowolny ciąg znaków.

Rysunek 6.33.
Edytor komórki



Wszystkie powyższe przykłady stanowią odmiany klasy `DefaultCellEditor`. Obiekt klasy `DefaultCellEditor` może zostać utworzony po wykorzystaniu obiektu klasy `JTextField`, `JCheckBox` lub `JComboBox`. Klasa `JTable` sama automatycznie instaluje edytor pól wyboru dla kolumn klasy `boolean` oraz edytor pól tekstowych dla pozostałych typów kolumn, które nie posiadają własnego obiektu rysującego. Edytory pól tekstowych pozwalają w takim przypadku na modyfikację łańcucha znaków będącego wynikiem zastosowania metody `toString` do obiektu zwróconego przez metodę `getValueAt` modelu tabeli.

Po zakończeniu edycji komórki edytor przekazuje rezultat z powrotem do modelu tabeli, korzystając z metody `setValueAt`. Zadaniem programisty jest takie zaimplementowanie tej metody, aby jej wywołanie przez edytor spowodowało nadanie odpowiedniej wartości obiektowi w modelu tabeli.

> Edytor pola tekstowego może łatwo przekształcić wartość komórki w łańcuch znaków, korzystając z metody `toString` obiektu zwróconego przez wywołanie metody `getValueAt`. Jednak przekształcenie w odwrotnym kierunku spoczywa na barkach programisty. Po zakończeniu edycji komórki edytor wywoła metodę `setValueAt`, której przekaże łańcuch znaków. Metoda ta musi umieć odpowiednio sparsować ten łańcuch. Na przykład jeśli komórka zawierała numeryczną wartość całkowitą, metoda `setValueAt` może wykorzystać metodę `Integer.parseInt`.

Aby użyć w komórkach tabeli edytora listy rozwijalnej, musimy go zainstalować samodzielnie, ponieważ klasa `JTable` nie może sama ustalić zbioru wartości dla danego typu komórki. W przypadku komórek kolumny *Moons* naszej tabeli umożliwimy użytkownikowi wybór wartości z przedziału od 0 do 20. Poniżej fragment kodu inicjujący listę rozwijalną.

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
   moonCombo.addItem(new Integer(i));
```

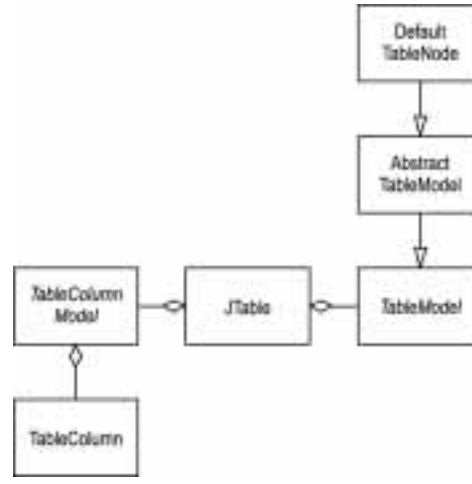
Następnie utworzymy obiekt klasy `DefaultCellEditor`, przekazując listę jako parametr jego konstruktora:

```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

Musimy jeszcze zainstalować utworzony edytor. W przeciwieństwie do edytora kolorów nie związemy go z określonym *typem*, ponieważ nie chcemy, by używany był przez tabelę dla wszystkich komórek typu `Integer`. Zamiast tego zainstalujemy go jedynie dla określonej kolumny tabeli.

Klasa `JTable` przechowuje informację o kolumnach tabeli, korzystając z obiektów typu `TableColumn`. Klasa `TableColumnModel` zarządza natomiast kolumnami. (Rysunek 6.34 przedstawia zależności między najważniejszymi klasami związanymi z tabelami). Jeśli nie planujemy dynamicznie umieszczać w tabeli nowych kolumn bądź usuwać istniejących, to nie musimy korzystać z usług modelu kolumn tabeli z wyjątkiem sytuacji, w której chcemy uzyskać obiekt `TableColumn` dla pewnej kolumny:

Rysunek 6.34.
Zależności między klasami tabel




```
TableColumnModel columnModel = table.getColumnModel();
      TableColumn moonColumn
         = columnModel.getColumn(PlanetTableModel.MOON_COLUMN);
```

Dysponując obiektem kolumny, możemy zainstalować edytor jej komórek:

```
moonColumn.setCellEditor(moonEditor);
```

Jeśli chcemy zmienić wysokość komórek tabeli, skorzystamy z poniższej metody.

```
table.setRowHeight(height);
```

Domyślnie wszystkie wiersze tabeli posiadają tę samą wysokość. Możemy jednak zmienić wysokość poszczególnych wierszy, psługując się wywołaniem:

```
table.setRowHeight(row, height);
```

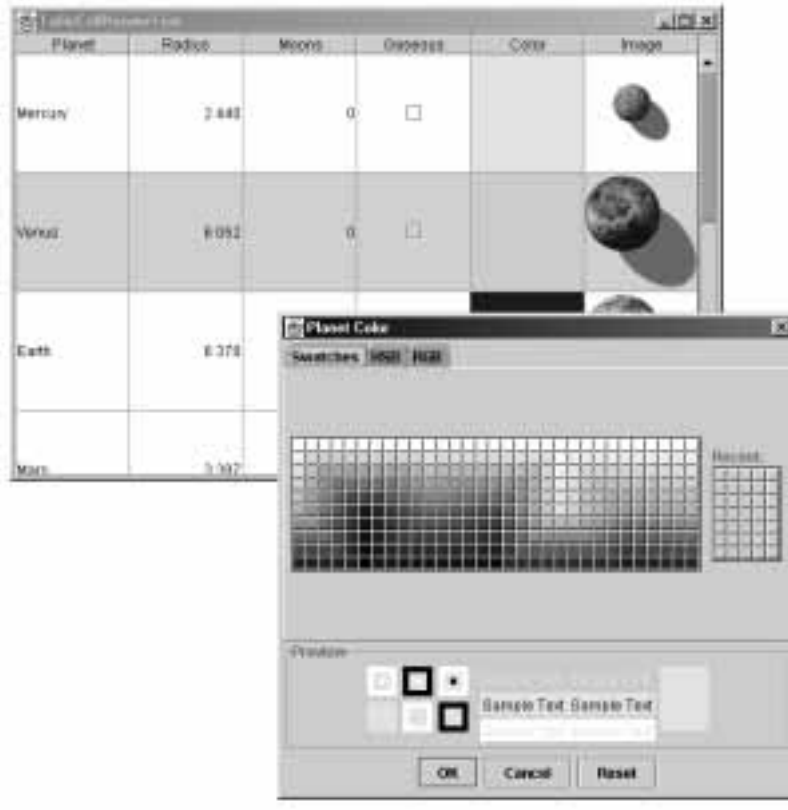
Rzeczywista wysokość komórki pomniejszona będzie o jej margines, który domyślnie wynosi 1. Wielkość marginesu możemy zmienić, używając poniższej metody.

```
table.setRowMargin(margin);
```

Tworzenie własnych edytorów

Jeśli uruchomimy przykładowy program i wybierzemy za pomocą myszy komórkę zawierającą kolor, to otworzy się okno dialogowe wyboru koloru. Użytkownik może wybrać nowy kolor i zaakceptować go przyciskiem *OK*, co spowoduje zmianę koloru komórki tabeli (patrz rysunek 6.35).

Rysunek 6.35.
Wybór koloru komórki



Edytor koloru komórek nie jest standardowym edytorem tabeli. Aby utworzyć własny edytor, należy zaimplementować interfejs `TableCellEditor`. Jest to dość pracochłonne zadanie i dlatego wersja SDK 1.3 wprowadziła klasę `AbstractCellEditor` zawierającą implementację obsługi zdarzeń.

Metoda `getTableCellEditorComponent` interfejsu `TableCellEditor` pobiera komponent rysujący komórkę tabeli. Jest zdefiniowana tak samo jak metoda `getTableCellRendererComponent` interfejsu `TableCellRenderer` z tą różnicą, że nie posiada parametru `hasFocus`. Ponieważ komórka jest edytowana, to automatycznie przyjmuje się, że wartością tego parametru jest `true`. Komponent edytora zastępuje obiekt rysujący podczas edycji komórki. W naszym przykładzie wywołanie metody zwraca pusty, niepokolorowany panel, sygnalizując w ten sposób, że komórka jest edytowana.

Edytor musi rozpocząć swoje działanie po kliknięciu komórki przez użytkownika.

Klasa `JTable` wywołuje edytor dla danego zdarzenia (na przykład kliknięcia myszą), aby sprawdzić, czy spowoduje ono rozpoczęcie procesu edycji. Klasa `AbstractCellEditor` definiuje metodę akceptującą wszystkie zdarzenia.

```
public boolean isCellEditable(EventObject anEvent)
{
   return true;
}
```

Jeśli zastąpimy tę metodę, tak by zwracała wartość `false`, to tabela w ogóle nie umieści komponentu edytora.

Po zainstalowaniu edytora wywoływana jest metoda `shouldSelectCell`, prawdopodobnie dla tego samego zdarzenia. Metoda ta rozpocząć powinna proces edycji, na przykład otwierając okno dialogowe.

```
public boolean shouldSelectCell(EventObject anEvent)
{
   colorDialog.setVisible(true);
   return true;
}
```

Jeśli proces edycji będzie musiał zostać zatrzymany lub przerwany (ponieważ użytkownik wybrał inną komórkę tabeli), to wywołana zostanie metoda `stopCellEditing` lub `cancelCellEditing`. Powinniśmy wtedy zamknąć okno dialogowe. Wywołanie metody `stopCellEditing` oznacza, że tabela chce zachować wartość zmodyfikowaną w procesie edycji. Metoda powinna zwrócić wartość `true`, jeśli wartość komórki jest dozwolona. W przypadku wyboru kolorów dozwolona będzie dowolna wartość. Jednak jeśli tworzymy edytor innych rodzajów danych, to powinniśmy zawsze sprawdzać, czy powstała w procesie edycji wartość jest dozwolona.

Po zakończeniu edycji należy także wywołać metodę klasy bazowej, która obsługuje dla nas zdarzenia.

```
public void cancelCellEditing()
{
   colorDialog.setVisible(false);
   super.cancelCellEditing);
}
```

Musimy dostarczyć także metodę, która umożliwi pobranie wartości powstałej w procesie edycji:

```
public Object getCellEditorValue()
{
   return colorChooser.getColor();
}
```

Podsumowując, edytor powinien:

1. rozszerzać klasę `AbstractCellEditor` i implementować interfejs `TableCellEitor`,
2. definiować metodę `getTableCellEditorComponent`, która zwraca nieinteraktywny komponent w przypadku gdy edytor otworzy własne okno dialogowe lub komponent umożliwiający edycję wewnątrz komórki (na przykład lista rozwijalna bądź pole tekstowe),

3. definiować metody `shouldSelectCell`, `stopCellEditing` i `cancelCellEditing` obsługujące rozpoczęcie, zakończenie i przerwanie procesu edycji; metody `stopCellEditing` i `cancelCellEditing` wywoływać powinny te same metody klasy bazowej, aby zapewnić powiadomienie obiektów nasłuchujących,

4. definiować metodę `getCellEditorValue` zwracającą nową wartość komórki powstałą w procesie edycji.

Na koniec musimy jeszcze wywołać metody `stopCellEditing` i `cancelCellEidting`, gdy użytkownik zakończy edycję. Tworząc okno dialogowe wyboru kolorów, opracujemy także obiekty nasłuchujące jego przycisków, które wywołają wspomniane metody.

```
colorDialog = JColorChooser.createDialog(null,
   "Planet Color", false, colorChooser,
   new
      ActionListener() // obiekt nasłuchujący przycisku OK
      {
         public void actionPerformed(ActionEvent event)
         {
            stopCellEditing();
         }
      },
   new
      ActionListener() // obiekt nasłuchujący przycisku Cancel
      {
         public void actionPerformed(ActionEvent event)
         {
            cancelCellEditing();
         }
});
```

Proces edycji powinien zostać przerwany także na skutek zamknięcia okna dialogowego. Osiągniemy to, instalując obiekt nasłuchujący okna:

```
colorDialog.addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent event)
      {
         cancelCellEditing();
      }
   });
```

W ten sposób zakończyliśmy implementację własnego edytora komórek.

Wiemy też, w jaki sposób umożliwić edycję komórek i zainstalować edytor. Pozostaje jeszcze tylko powiadomienie modelu tabeli o zmianie wartości edytowanej komórki. Po zakończeniu edycji komórki klasa `JTable` wywołuje następującą metodę modelu tabeli:

```
void setValueAt(Object value, int r, int c)
```

Musimy zastąpić tę metodę, aby przekazać do modelu nową wartość. Parametr `value` jest obiektem zwróconym przez edytor komórki. W przypadku edytora, który sami zaimplementowaliśmy, znamy typ obiektu zwróconego za pomocą metody `getCellEditorValue`. W przypadku edytora klasy `DefaultCellEditor` istnieją natomiast trzy możliwości. Może to być typ `Boolean`, jeśli edytor był polem wyboru lub łańcuch znaków, jeśli edytorem było pole tekstowe lub obiekt wybrany przez użytkownika z listy rozwijalnej.

Jeśli obiekt `value` nie posiada odpowiedniego typu, należy go w taki przekształcić. Sytuacja taka najczęściej zdarza się, gdy liczba edytowana jest w polu tekstowym. W naszym przykładzie lista rozwijalna wypełniona została obiektami klasy `Integer`, dlatego też nie ma potrzeby konwersji typu.

Listing 6.13. *TableCellRenderTest.java*

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

/**
   Program demonstrujący wykorzystanie obiektów rysujących komórki
   i edytorów komórek tabeli.
*/
public class TableCellRenderTest
{
   public static void main(String[] args)
   {
      JFrame frame = new TableCellRenderFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca tabelę danych o planetach.
*/
class TableCellRenderFrame extends JFrame
{
   public TableCellRenderFrame()
   {
      setTitle("TableCellRenderTest");
      setSize(WIDTH, HEIGHT);

      TableModel model = new PlanetTableModel();
      JTable table = new JTable(model);

      // instaluje obiekt rysujący i edytor

      table.setDefaultRenderer(Color.class,
         new ColorTableCellRenderer());
      table.setDefaultEditor(Color.class,
         new ColorTableCellEditor());

      JComboBox moonCombo = new JComboBox();
      for (int i = 0; i <= 20; i++)
         moonCombo.addItem(new Integer(i));
      TableColumnModel columnModel = table.getColumnModel();
      TableColumn moonColumn
         = columnModel.getColumn(PlanetTableModel.MOON_COLUMN);
      moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
```

```
      // wyświetla tabelę

      table.setRowHeight(100);
      getContentPane().add(new JScrollPane(table),
         BorderLayout.CENTER);
   }

   private static final int WIDTH = 600;
   private static final int HEIGHT = 400;
}

/**
   Model tabeli planet określający wartości,
   sposób rysowania i edycji danych.
*/
class PlanetTableModel extends AbstractTableModel
{
   public String getColumnName(int c)
   {
      return columnNames[c];
   }

   public Class getColumnClass(int c)
   {
      return cells[0][c].getClass();
   }

   public int getColumnCount()
   {
      return cells[0].length;
   }

   public int getRowCount()
   {
      return cells.length;
   }

   public Object getValueAt(int r, int c)
   {
      return cells[r][c];
   }

   public void setValueAt(Object obj, int r, int c)
   {
      cells[r][c] = obj;
   }

   public boolean isCellEditable(int r, int c)
   {
      return c == NAME_COLUMN
         || c == MOON_COLUMN
         || c == GASEOUS_COLUMN
         || c == COLOR_COLUMN;
   }

   public static final int NAME_COLUMN = 0;
   public static final int MOON_COLUMN = 2;
```

```
    public static final int GASEOUS_COLUMN = 3;
    public static final int COLOR_COLUMN = 4;

    private Object[][] cells =
    {
       {
          "Mercury", new Double(2440),  new Integer(0),
          Boolean.FALSE, Color.yellow,
          new ImageIcon("Mercury.gif")
       },
       {
          "Venus", new Double(6052), new Integer(0),
          Boolean.FALSE, Color.yellow,
          new ImageIcon("Venus.gif")
       },
       {
          "Earth", new Double(6378), new Integer(1),
          Boolean.FALSE, Color.blue,
          new ImageIcon("Earth.gif")
       },
       {
          "Mars", new Double(3397), new Integer(2),
          Boolean.FALSE, Color.red,
          new ImageIcon("Mars.gif")
       },
       {
          "Jupiter", new Double(71492), new Integer(16),
          Boolean.TRUE, Color.orange,
          new ImageIcon("Jupiter.gif")
       },
       {
          "Saturn", new Double(60268), new Integer(18),
          Boolean.TRUE, Color.orange,
          new ImageIcon("Saturn.gif")
       },
       {
          "Uranus", new Double(25559), new Integer(17),
          Boolean.TRUE, Color.blue,
          new ImageIcon("Uranus.gif")
       },
       {
          "Neptune", new Double(24766), new Integer(8),
          Boolean.TRUE, Color.blue,
          new ImageIcon("Neptune.gif")
       },
       {
          "Pluto", new Double(1137), new Integer(1),
          Boolean.FALSE, Color.black,
          new ImageIcon("Pluto.gif")
       }
    };

    private String[] columnNames =
    {
       "Planet", "Radius", "Moons", "Gaseous", "Color", "Image"
    };
}
```

```
/**
   Klasa obiektu rysującego kolorowy panel wewnątrz komórki.
*/
class ColorTableCellRenderer implements TableCellRenderer
{
   public Component getTableCellRendererComponent(JTable table,
      Object value, boolean isSelected, boolean hasFocus,
      int row, int column)
   {
      panel.setBackground((Color)value);
      return panel;
   }

   // zwracany jest panel, którego kolor tła
   // określony jest przez obiekt Color komórki

   private JPanel panel = new JPanel();
}

/**
   Edytor otwierający okno dialogowe wyboru koloru
*/
class ColorTableCellEditor extends AbstractCellEditor
   implements TableCellEditor
{
   ColorTableCellEditor()
   {
      panel = new JPanel();
      // przygotowuje okno dialogowe

      colorChooser = new JColorChooser();
      colorDialog = JColorChooser.createDialog(null,
         "Planet Color", false, colorChooser,
         new
            ActionListener() // obiekt nasłuchujacy przycisku OK
            {
               public void actionPerformed(ActionEvent event)
               {
                  stopCellEditing();
               }
            },
         new
            ActionListener() // obiekt nasłuchujący przycisku Cancel
            {
               public void actionPerformed(ActionEvent event)
               {
                  cancelCellEditing();
               }
            });
      colorDialog.addWindowListener(new
         WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               cancelCellEditing();
            }
         });
   }
```

```
   public Component getTableCellEditorComponent(JTable table,
      Object value, boolean isSelected, int row, int column)
   {
      // tutaj uzyskujemy bieżącą wartość Color.
      // Przechowujemy ją w obiekcie okna.
      colorChooser.setColor((Color)value);
      return panel;
   }

   public boolean shouldSelectCell(EventObject anEvent)
   {
      // rozpoczęcie edycji
      colorDialog.setVisible(true);

      // informuje metodę wywołującą o rozpoczęciu edycji
      return true;
   }

   public void cancelCellEditing()
   {
      // edycja przerwana - zamyka okno dialogowe
      colorDialog.setVisible(false);
      super.cancelCellEditing();
   }

   public boolean stopCellEditing()
   {
      // edycja zakończona - zamyka okno dialogowe
      colorDialog.setVisible(false);
      super.stopCellEditing();

      // informuje metodę, że wartość koloru jest dozwolona
      return true;
   }

   public Object getCellEditorValue()
   {
      return colorChooser.getColor();
   }

   private Color color;
   private JColorChooser colorChooser;
   private JDialog colorDialog;
   private JPanel panel;
}
```

javax.swing.JTable

- `void setRowHeight(int height)` **nadaje wszystkim wierszom tabeli wysokość** `height` **pikseli**.
- `void setRowHeight(int row, int height)` **nadaje danemu wierszowi tabeli wysokość** `height` **pikseli**.
- `void setRowMargin(int margin)` **określa wolną przestrzeń między sąsiednimi wierszami**.

- `int getRowHeight()` pobiera domyślną wysokość wszystkich wierszy w tabeli.
- `int getRowHeight(int row)` pobiera wysokość danego wiersza tabeli.
- `int getRowMargin()` pobiera wielkość wolnej przestrzeni między sąsiednimi wierszami.
- `Rectangle getCellRect(int row, int column, boolean includeSpacing)` zwraca obszar komórki tabeli.

  Parametry: `row, column` wiersz i kolumna tabeli,

  `includeSpacing` wartość `true`, jeśli obszar uwzględniać ma marginesy.

- `Color getSelectionBackground()`
- `Color getSelectionForeground()`

  Zwracają kolory używane dla prezentacji komórki wybranej przez użytkownika.

javax.swing.table.TableModel

- `Class getColumnClass(int columnIndex)` zwraca klasę obiektów danej kolumny. Informacja ta wykorzystywana jest przez obiekty rysujące i komórki.

javax.swing.table.TableCellRenderer

- `Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)` zwraca komponent, którego metoda `paint` wywoływana jest w celu narysowania komórki tabeli.

  Parametry: `table` tabela zawierająca rysowaną komórkę,

  `value` obiekt rysowanej komórki,

  `selected` wartość `true`, jeśli komórka jest wybrana,

  `hasFocus` wartość `true`, jeśli komórka jest przeglądana,

  `row, column` wiersz i kolumna komórki.

javax.swing.table.TableColumnModel

- `TableColumn getColumn(int index)` zwraca obiekt kolumny tabeli o danym indeksie.

javax.swing.table.TableColumn

- `void setCellEditor(TableCellEditor editor)`
- `void setCellRenderer(TableCellRenderer renderer)`

  Instalują edytor i obiekt rysujący dla wszystkich komórek danej kolumny.

javax.swing.DefaultCellEditor

- `DefaultCellEditor(JComboBox comboBox)` tworzy edytor komórek wykorzystujący listę rozwijalną do wyboru wartości.

javax.swing.CellEditor

- `boolean isCellEditable(EventObject event)` zwraca wartość `true`, jeśli zdarzenie rozpocznie proces edycji komórki.
- `boolean shouldSelectCell(EventObject anEvent)` rozpoczyna proces edycji. Zwraca wartość `true`, jeśli edytowana komórka powinna zostać *wybrana*. Wartość `false` powinna być zwrócona, jeśli nie chcemy, by proces edycji zmieniał wybór komórek.
- `void cancelCellEditing()` przerywa proces edycji. Wartość powstała na skutek edycji może być porzucona.
- `boolean stopCellEditing()` kończy proces edycji. Wartość powstała na skutek edycji może być wykorzystana. Zwraca wartość `true`, jeśli wartość powstała na skutek edycji jest dozwolona i może być pobrana.
- `Object getCellEditorValue()` zwraca edytowaną wartość.
- `void addCellEditorListener(CellEditorListener l)`
- `void removeCellEditorListener(CellEditorListener l)`

  Dodają i usuwają obowiązkowy obiekt nasłuchujący edytora.

javax.swing.table.TableCellEditor

- `Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)` zwraca komponent, którego metoda `paint` wywoływana jest w celu narysowania komórki tabeli.

  Parametry:

`table`	tabela zawierająca rysowaną komórkę,
`value`	obiekt rysowanej komórki,
`selected`	wartość `true`, jeśli komórka jest wybrana,
`row`, `column`	wiersz i kolumna komórki.

javax.swing.JColorChooser

- `JColorChooser()` tworzy obiekt wyboru koloru. Początkowo wybrany jest kolor biały.
- `Color getColor()`
- `void setColor(Color c)`

  Pobierają i ustawiają kolor wybrany przez obiekt wyboru.

- `static JDialog createDialog(Component parent, String, title, boolean modal, JColorChooser chooser, ActionListener okListener, ActionListener cancelListener)` **tworzy okno dialogowe wyboru koloru.**

  Parametry:

`parent`	**komponent, nad którym pojawić ma się okno dialogowe,**
`title`	**tytuł okna dialogowego,**
`modal`	**wartość** `true`**, jeśli okno blokować ma wykonanie aplikacji do momentu jego zamknięcia,**
`chooser`	**obiekt wyboru koloru,**
`okListener`, `cancelListener`	**obiekty nasłuchujące przycisków** *OK* **i** *Cancel*.

- `static Color showDialog(Component component, String title, Color initialColor)` **tworzy i wyświetla modalne okno dialogowe wyboru koloru.**

  Parametry:

`component`	**komponent, nad którym pojawić ma się okno dialogowe,**
`title`	**tytuł okna dialogowego,**
`initialColor`	**początkowo wybrany kolor.**

Operacje na wierszach i kolumnach

W podrozdziale tym pokażemy, w jaki sposób wykonywać operacje na wierszach i kolumnach tabeli. Podczas lektury tego materiału musimy pamiętać przede wszystkim, że tabele biblioteki Swing są asymetryczne, czyli na wierszach można wykonywać inne operacje niż na kolumnach. Komponent tabeli zaprojektowano z myślą o prezentacji informacji w postaci wierszy o tej samej strukturze, takich jak na przykład rekordy będące wynikiem zapytania do bazy danych, a nie dla prezentacji dowolnej dwuwymiarowej siatki obiektów.

Zmiana szerokości kolumn

Klasa `TableColumn` udostępnia metody umożliwiające nadzór nad zmianami szerokości kolumny wykonywanymi przez użytkownika. Programista może określić preferowaną, najmniejszą i największą szerokość kolumny, korzystając z poniższych metod.

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

Informacja ta wykorzystywana jest przez komponent tabeli podczas jej wyświetlania.

Metoda

```
void setResizable(boolean resizable)
```

zezwala lub zabrania użytkownikowi zmieniać szerokość kolumny.

Szerokość kolumny można także zmieniać programowo, korzystając z poniższej metody.

```
void setWidth(int width)
```

Gdy zmieniana jest szerokość kolumny, to domyślnie całkowita szerokość tabeli nie ulega zmianie. Oznacza to, że zmiana szerokości kolumny spowoduje także zmianę szerokości kolumn tabeli, leżących od niej na prawo. Zachowanie takie jest o tyle rozsądne, że pozwala użytkownikowi dostosować szerokość kolejnych kolumn, poruszając się od lewej strony tabeli ku prawej.

Zachowanie to możemy zmienić na jedno z wymienionych w tabeli 6.2 za pomocą metody

```
void setAutoResizeMode(int mode)
```

udostępnianej przez klasę `JTable`.

Tabela 6.2. *Tryby zmiany szerokości kolumn*

Tryb	Zachowanie
`AUTO_RESIZE_OFF`	Nie zmienia szerokości innych kolumn, ale szerokość całej tabeli.
`AUTO_RESIZE_NEXT_COLUMN`	Zmienia jedynie szerokość następnej kolumny.
`AUTO_RESIZE_SUBSEQUENT_COLUMNS`	Zmienia równo szerokość wszystkich następnych kolumn. Zachowanie domyślne.
`AUTO_RESIZE_LAST_COLUMN`	Zmienia jedynie szerokość ostatniej kolumny.
`AUTO_RESIZE_ALL_COLUMNS`	Zmienia szerokość wszystkich kolumn tabeli. Zachowanie to najczęściej nie jest właściwe, ponieważ uniemożliwia użytkownikowi określenie szerokości więcej niż jednej kolumny.

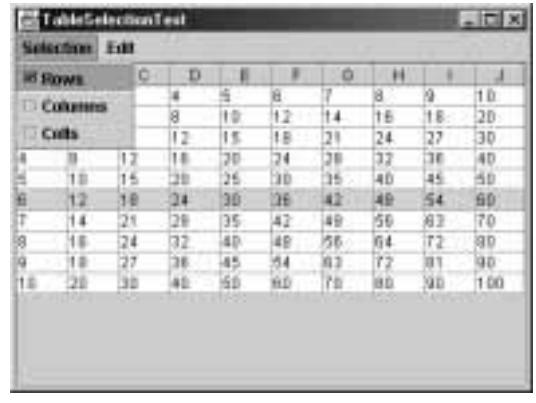
Wybór wierszy, kolumn i komórek

W zależności od trybu wyboru użytkownik wybierać może wiersze, kolumny bądź komórki tabeli. Domyślnym trybem wyboru jest tryb zezwalający na wybór wierszy tabeli. Wybranie komórki powoduje automatycznie wybranie całego wiersza (patrz rysunek 6.36). Wywołanie

```
table.setRowSelectionAllowed(false)
```

wyłącza możliwość wyboru wierszy.

Rysunek 6.36.
Wybór wiersza tabeli



Jeśli tryb wyboru wierszy jest włączony, to możemy określić, czy użytkownikowi wolno wybrać pojedynczy wiersz, ciągły zakres wierszy bądź dowolny zbiór wierszy. W tym celu musimy pobrać *model wyboru* i skorzystać z jego metody `setSelectionMode`:

```
table.getSelectionModel().setSelectionMode(mode);
```

Parametr `mode` przyjmować może jedną z trzech wartości:

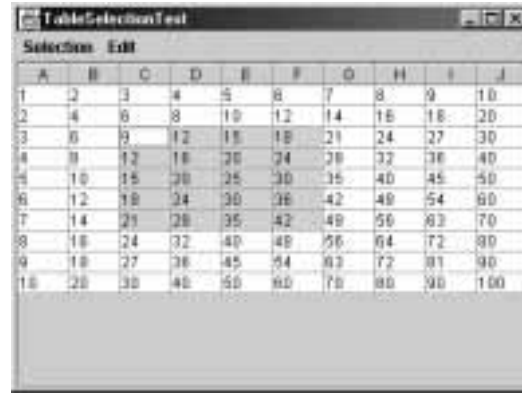
```
ListSelectionMode.SINGLE_SELECTION
ListSelectionMode.SINGLE_INTERVAL_SELECTION
ListSelectionMode.MULTIPLE_INTERVAL_SELECTION
```

Wybór kolumn jest domyślnie zabroniony. Możemy go umożliwić, wywołując

```
table.setColumnSelectionAllowed(true)
```

Zezwolenie na równoczesny wybór wierszy i kolumn oznacza możliwość wyboru komórek tabeli. Użytkownik może wtedy wybierać zakresy komórek, jak pokazano na rysunku 6.37. Wybór komórek umożliwić możemy także, korzystając z metody

```
table.setCellSelectionEnabled(true)
```



Rysunek 6.37. *Wybór zakresu komórek tabeli*

> We wczesnych wersjach biblioteki Swing dopuszczenie równoczesnego wyboru wierszy i kolumn powodowało, że wybranie komórki powodowało automatycznie wybranie zawierającego ją wiersza i kolumny.

Informację o wybranych wierszach i kolumnach możemy uzyskać, wywołując metody `getSelectedRows` i `getSelectedColumns`. Metody te zwracają tablice `int[]` zawierające indeksy wybranych wierszy bądź kolumn.

Program, którego kod źródłowy zawiera listing 6.14, umożliwia obserwację sposobu wyboru elementów tabeli. Jego menu umożliwia włączanie lub wyłączanie możliwości wyboru wierszy, kolumn i komórek tabeli.

Ukrywanie kolumn

Metoda `removeColumn` klasy `JTable` usuwa kolumnę z widoku tabeli. Dane kolumny nie są usuwane z modelu tabeli, a jedynie ukrywane przed jej widokiem. Parametrem metody

`removeColumn` jest obiekt klasy `TableColumn`. Jeśli dysponujemy numerem kolumny (na przykład zwróconym przez metodę `getSelectedColumns`), to musimy najpierw pobrać od modelu kolumn tabeli odpowiedni obiekt reprezentujący kolumnę:

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn();
table.removeColumn(column);
```

Jeśli zapamiętamy obiekt kolumny, to możemy później wstawić go z powrotem do tabeli:

```
table.addColumn(column);
```

Wywołanie tej metody spowoduje dodanie kolumny jako ostatniej kolumny tabeli. Jeśli chcemy ją umieścić w innym miejscu tabeli, to musimy skorzystać jeszcze z metody `moveColumn`.

Nową kolumnę możemy dodać także, tworząc nowy obiekt klasy `TableColumn` o indeksie odpowiadającym numerowi kolumny w modelu.

```
table.addColumn(new TableColumn(modelColumnIndex));
```

Możemy utworzyć wiele obiektów tej klasy, które stanowić będą reprezentacje jednej i tej samej kolumny modelu.

Klasa `JTable` nie dysponuje metodami, które umożliwiałyby wstawienie lub usunięcie kolumny z modelu. A także metodami, które umożliwiałyby ukrycie wierszy. Jeśli chcemy ukrywać wiersze tabeli, musimy utworzyć model filtra podobny do pokazanego wcześniej modelu filtra sortującego.

Dodawanie i usuwanie wierszy w domyślnym modelu tabeli

Klasa `DefaultTableModel` jest klasą konkretną implementującą interfejs `TableModel`. Przechowuje ona dwuwymiarową siatkę obiektów. Jeśli posiadamy już dane zorganizowane w postaci tabelarycznej, to oczywiście nie ma sensu kopiować ich do domyślnego modelu tabeli. Jego zastosowanie jest jednak bardzo wygodne, jeśli musimy szybko utworzyć tabelę reprezentującą niewielki zbiór danych. Klasa `DefaultTableModel` dysponuje przy tym metodami umożliwiającymi dodawanie wierszy oraz kolumn, a także usuwanie wierszy.

Metody `addRow` i `addColumn` dodają odpowiednio nowy wiersz lub kolumnę. Przekazujemy im tablicę `Object[]` lub wektor zawierający nowe dane. Metodzie `addColumn` musimy przekazać także nazwę nowej kolumny. Obie metody umieszczają nowe elementy tabeli na końcu siatki. Aby wstawić wiersz pomiędzy wiersze już istniejące w tabeli, możemy skorzystać z metody `insertRow`. Niestety dla kolumn nie jest dostępna analogiczna metoda.

Metoda `removeRow` usuwa wiersz z modelu tabeli. Również w tym przypadku nie istnieje taka metoda dla kolumn.

Ponieważ obiekt klasy `JTable` rejestruje się jako obiekt nasłuchujący modelu tabeli, to widok tabeli jest powiadamiany za każdym razem, gdy wstawiane są lub usuwane dane z modelu. Umożliwia to aktualizację widoku tabeli.

Program, którego kod źródłowy zamieszczamy w listingu 6.14, stanowi ilustrację operacji wyboru i edycji tabeli. W modelu tabeli umieściliśmy prosty zbiór danych (tabliczkę mnożenia). Menu *Edit* programu umożliwia:

- ukrycie wszystkich wybranych kolumn,
- przywrócenie wszystkich kolumn, które zostały kiedykolwiek ukryte,
- usunięcie wybranych wierszy z tabeli,
- dodanie wiersza danych na końcu danych modelu.

Przykład ten kończy omówienie komponentu tabel. Zrozumienie sposobów wykorzystania tabel jest nieco łatwiejsze niż w przypadku drzew, ponieważ prezentowany za pomocą tabeli model danych jest prostszy koncepcyjnie. Jednak sama implementacja komponentu tabeli jest w rzeczywistości bardziej skomplikowana niż w przypadku komponentu drzewa. Przyczyniają się do tego możliwości zmiany szerokości kolumn, dodawania własnych obiektów rysujących i edytorów. W rozdziale tym skoncentrowaliśmy się na zagadnieniach, które posiadają największą przydatność w praktyce programisty, a więc prezentacji tabel bazy danych za pomocą komponentu tabeli, sortowaniu tabel oraz wykorzystaniu własnych obiektów rysujących i edytorów. Jeśli potrzebować będziemy informacji dotyczących bardziej zaawansowanych lub nietypowych zastosowań tabeli, to po raz kolejny wrócimy do książek *Core Java Foundation Classes* autorstwa Kim Topley oraz *Graphic Java 2* napisanej przez Davida Geary'ego.

Listing 6.14. *TableSelectionTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.text.*;
import javax.swing.*;
import javax.swing.table.*;

/**
   Program demonstrujący wybór, dodawanie i usuwanie
   wierszy oraz kolumn tabeli.
*/
public class TableSelectionTest
{
   public static void main(String[] args)
   {
      JFrame frame = new TableSelectionFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca tabliczkę mnożenia
   i posiadająca menu umożliwiające określenie trybu wyboru
   (wiersze/kolumny/komórki) oraz dodawanie i usuwanie
   wierszy i kolumn.
*/
class TableSelectionFrame extends JFrame
{
```

```
public TableSelectionFrame()
{
   setTitle("TableSelectionTest");
   setSize(WIDTH, HEIGHT);

   // tworzy tabliczkę mnożenia

   model = new DefaultTableModel(10, 10);

   for (int i = 0; i < model.getRowCount(); i++)
      for (int j = 0; j < model.getColumnCount(); j++)
         model.setValueAt(
            new Integer((i + 1) * (j + 1)), i, j);

   table = new JTable(model);

   Container contentPane = getContentPane();
   contentPane.add(new JScrollPane(table), "Center");

   removedColumns = new ArrayList();

   // tworzy menu

   JMenuBar menuBar = new JMenuBar();
   setJMenuBar(menuBar);

   JMenu selectionMenu = new JMenu("Selection");
   menuBar.add(selectionMenu);

   final JCheckBoxMenuItem rowsItem
      = new JCheckBoxMenuItem("Rows");
   final JCheckBoxMenuItem columnsItem
      = new JCheckBoxMenuItem("Columns");
   final JCheckBoxMenuItem cellsItem
      = new JCheckBoxMenuItem("Cells");

   rowsItem.setSelected(table.getRowSelectionAllowed());
   columnsItem.setSelected(table.getColumnSelectionAllowed());
   cellsItem.setSelected(table.getCellSelectionEnabled());

   rowsItem.addActionListener(new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            table.clearSelection();
            table.setRowSelectionAllowed(
               rowsItem.isSelected());
            cellsItem.setSelected(
               table.getCellSelectionEnabled());
         }
      });
   selectionMenu.add(rowsItem);

   columnsItem.addActionListener(new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
```

```
      {
         table.clearSelection();
         table.setColumnSelectionAllowed(
            columnsItem.isSelected());
         cellsItem.setSelected(
            table.getCellSelectionEnabled());
      }
   });
selectionMenu.add(columnsItem);

cellsItem.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         table.clearSelection();
         table.setCellSelectionEnabled(
            cellsItem.isSelected());
         rowsItem.setSelected(
            table.getRowSelectionAllowed());
         columnsItem.setSelected(
            table.getColumnSelectionAllowed());
      }
   });
selectionMenu.add(cellsItem);

JMenu tableMenu = new JMenu("Edit");
menuBar.add(tableMenu);

JMenuItem hideColumnsItem = new JMenuItem("Hide Columns");
hideColumnsItem.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         int[] selected = table.getSelectedColumns();
         TableColumnModel columnModel
            = table.getColumnModel();

         // usuwa kolumny z widoku tabeli, począwszy od
         // najwyższego indeksu, aby nie zmieniać numerów kolumn


         for (int i = selected.length - 1; i >= 0; i--)
         {
            TableColumn column
               = columnModel.getColumn(selected[i]);
            table.removeColumn(column);

            // przechowuje ukryte kolumny do ponownej prezentacji

            removedColumns.add(column);
         }
      }
   });
tableMenu.add(hideColumnsItem);
```

```
JMenuItem showColumnsItem = new JMenuItem("Show Columns");
showColumnsItem.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         // przywraca wszystkie usunięte dotąd kolumny
         for (int i = 0; i < removedColumns.size(); i++)
            table.addColumn(
               (TableColumn)removedColumns.get(i));
         removedColumns.clear();
      }
   });
tableMenu.add(showColumnsItem);

JMenuItem addRowItem = new JMenuItem("Add Row");
addRowItem.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         // dodaje nowy wiersz tabliczki mnożenia do modelu

         Integer[] newCells
            = new Integer[model.getColumnCount()];
         for (int i = 0; i < newCells.length; i++)
            newCells[i] = new Integer((i + 1)
               * (model.getRowCount() + 1));
         model.addRow(newCells);
      }
   });
tableMenu.add(addRowItem);

JMenuItem removeRowsItem = new  JMenuItem("Remove Rows");
removeRowsItem.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         int[] selected = table.getSelectedRows();

         // usuwa wiersze z modelu, rozpoczynając od najwyższego indeksu,
         // aby nie zmieniać numerów wierszy


         for (int i = selected.length - 1; i >= 0; i--)
            model.removeRow(selected[i]);
      }
   });
tableMenu.add(removeRowsItem);

JMenuItem clearCellsItem = new  JMenuItem("Clear Cells");
clearCellsItem.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
```

```
            // nadaje wybranym komórkom wartość 0

            for (int i = 0; i < table.getRowCount(); i++)
               for (int j = 0; j < table.getColumnCount(); j++)
                  if (table.isCellSelected(i, j))
                     table.setValueAt(new Integer(0), i, j);
         }
      });
   tableMenu.add(clearCellsItem);
}

private DefaultTableModel model;
private JTable table;
private ArrayList removedColumns;

private static final int WIDTH = 400;
private static final int HEIGHT = 300;
}
```

javax.swing.JTable

- `void setAutoResizeMode(int mode)` **określa tryb zmiany szerokości kolumn.**

  Parametry: `mode` **jedna z wartości** `AUTO_RESIZE_OFF`, `AUTO_RESIZE_NEXT_COLUMN`, `AUTO_RESIZE_SUBSEQUENT_COLUMNS`, `AUTO_RESIZE_LAST_COLUMN`, `AUTO_RESIZE_ALL_COLUMNS`.

- `ListSelectionModel getSelectionModel()` **zwraca model wyboru, który umożliwia następnie określenie trybu wyboru.**
- `void setRowSelectionAllowed(boolean b)`, **jeśli** `b` **posiada wartość** `true`, **to kliknięcie komórki tabeli powoduje wybranie całego wiersza.**
- `void setColumnSelectionAllowed(boolean b)`, **jeśli** `b` **posiada wartość** `true`, **to kliknięcie komórki tabeli powoduje wybranie całej kolumny.**
- `void setRowSelectionEnabled(boolean b)`, **jeśli** `b` **posiada wartość** `true`, **to możliwy jest wybór poszczególnych komórek tabeli. Taki sam rezultat daje wywołanie kolejno metod** `setRowSelectionAllowed(b)` **i** `setColumnSelectionAllowed(b)`.
- `boolean getRowSelectionAllowed()` **zwraca wartość** `true`, **jeśli dozwolony jest wybór wierszy tabeli.**
- `boolean getColumnSelectionAllowed()` **zwraca wartość** `true`, **jeśli dozwolony jest wybór kolumn tabeli.**
- `boolean getCellSelectionEnabled()` **zwraca wartość** `true`, **jeśli dozwolony wybór wierszy i kolumn tabeli.**
- `void addColumn(TableColumn column)` **dodaje kolumnę do widoku tabeli.**
- `void moveColumn(int from, int to)` **przesuwa kolumnę o indeksie** `from` **w taki sposób, że jej indeksem staje się** `to`. **Operacja ta dotyczy jedynie widoku tabeli.**
- `void removeColumn(TableColumn column)` **usuwa kolumnę z widoku tabeli.**

javax.swing.table.TableColumn

- `TableColumn(int modelColumnIndex)` tworzy obiekt reprezentujący kolumnę tabeli o danym indeksie.
- `void setPreferredWidth(int width)`
- `void setMinWidth(int width)`
- `void setMaxWidth(int width)`

  Określają preferowaną, najmniejszą i największą szerokość danej kolumny.
- `void setWidth(int width)` ustawia bieżącą szerokość danej kolumny.
- `void setResizable(boolean b)`, jeśli `b` posiada wartość `true`, to użytkownik może zmieniać szerokość kolumny.

javax.swing.ListSelectionModel

- `void setSelectionMode(int mode)` określa tryb wyboru.

  Parametry: `mode` jedna z wartości `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` i `MULTIPLE_INTERVAL_SELECTION`.

javax.swing.table.DefaultTableModel

- `void addRow(Object[] rowData)`
- `void addColumn(Object columnName, Object[] columnData)`

  Dodaje wiersz lub kolumnę na końcu modelu danych.
- `void insertRow(int row, Object[] rowData)` dodaje wiersz danych na pozycji o indeksie `row`.
- `void removeRow(int row)` usuwa wiersz z modelu.
- `void moveRow(int start, int end, int to)` przesuwa wszystkie wiersze o indeksach z przedziału od `start` do `end` na nowe pozycje zaczynające się od indeksu `to`.

Komponenty formatujące tekst

W książce *Java 2. Podstawy* omówiliśmy podstawowe komponenty związane z edycją tekstu, takie jak `JTextField` i `JTextArea`. Klasy te są przydatne do pobierania tekstu wprowadzanego przez użytkownika. Istnieje także klasa `JEditorPane`, która wyświetla i umożliwia edycję tekstu zapisanego w formatach RTF i HTML. (Format RTF używany jest przez szereg aplikacji firmy Microsoft do wymiany dokumentów. Jest słabo udokumentowany i nawet aplikacje firmy Microsoft mają problemy z jego prawidłowym wykorzystaniem. W książce tej nie będziemy zajmować się wykorzystaniem tego formatu. Sama firma Sun twierdzi, że obsługa tego formatu w bibliotekach Java jest jedynie fragmentaryczna. Nie zdziwimy się więc, jeśli w kolejnych wersjach zostanie w ogóle usunięta. Zwłaszcza że poprawa wsparcia formatu HTML będzie kosztować jeszcze sporo wysiłku).

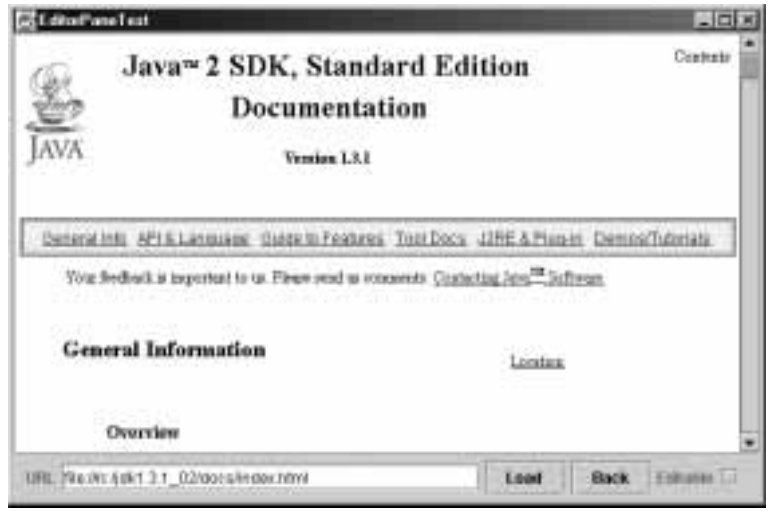
Musimy przyznać, że możliwości klasy `JEditorPane` są w obecnym stanie dość ograniczone. Potrafi ona wyświetlić proste strony w formacie HTML, ale z większością stron, które możemy spotkać obecnie w sieci, Internet ma problemy. Również edytor HTML nie posiada dużych możliwości, ale to akurat nie jest dużym ograniczeniem, ponieważ rzadko która aplikacja wymaga od użytkownika edytowania pliku w formacie HTML.

Dobrym zastosowaniem klasy `JEditorPane` może być wyświetlanie zawartości systemu pomocy zapisanej w formacie HTML. Ponieważ korzystamy wtedy z własnych plików źródłowych HTML, to możemy unikać w nich konstrukcji, z którymi klasa `JEditorPane` obecnie sobie nie radzi.

> Więcej informacji na temat tworzenia systemów pomocy dla profesjonalnych aplikacji znajdziemy pod adresem *http://java.sun.com/products/javahelp/index.html*.

> Klasa pochodna `JTextPane` klasy `JEditorPane` umożliwia przechowywanie i edycję tekstu sformatowanego przy użyciu różnych czcionek z możliwością umieszczania w nim różnych komponentów. Jeśli będziemy chcieli utworzyć aplikację umożliwiającą użytkownikowi formatowanie tekstu, to powinniśmy najpierw zapoznać się z programem demonstracyjnym StylePad dołączonym do SDK.

Program, którego kod zamieszczamy w listingu 6.15, korzysta z panelu edytora w celu wyświetlenia zawartości strony w języku HTML. W dolnej części jego okna umieszczone jest pole tekstowe, w którym wprowadzić należy adres URL. Musi ona zaczynać się od `http:` lub `file:`. Wybranie przycisku *Load* spowoduje wyświetlenie w panelu edytora strony o podanym adresie (patrz rysunek 6.38).



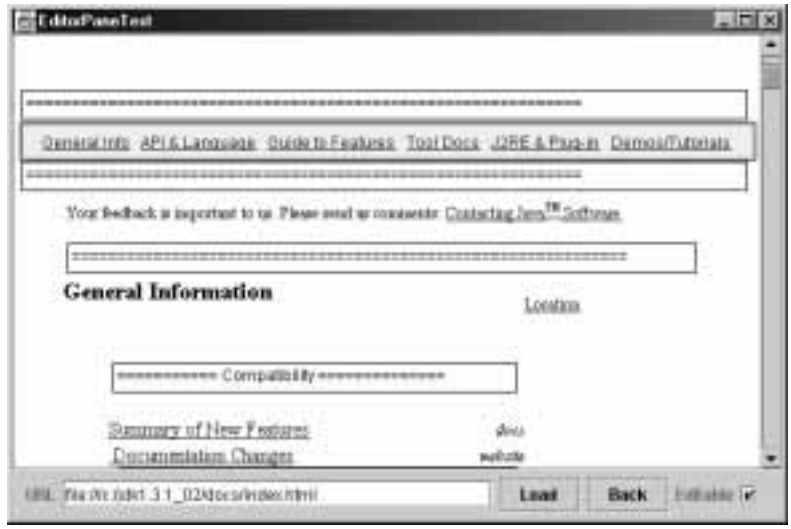
Rysunek 6.38. *Panel edytora wyświetlający stronę HTML*

Hiperłącza na wyświetlonej stronie są aktywne i wybranie jednego z nich spowoduje załadowanie kolejnej strony. Przycisk *Back* umożliwia powrót do poprzedniej strony.

Program ten jest właściwie prostą przeglądarką stron internetowych. Oczywiście nie posiada on możliwości komercyjnie dostępnych przeglądarek, takich jak listy zakładek czy buforowanie stron. Panel edytora nie umożliwia też wyświetlania apletów.

Jeśli zaznaczymy pole wyboru *Editable*, to panel edytora umożliwi nam edycję załadowanej strony. Możemy wpisywać w nim tekst lub usuwać go, korzystając z klawisza *Backspace*. Panel edytora obsługuje także kombinacje klawiszy *Ctrl+X*, *Ctrl+C*, *Ctrl+V* umożliwiające wycinanie, kopiowanie i wklejanie tekstu. Jednak umożliwienie formatowania tekstu wymagałoby jeszcze sporo pracy.

Gdy panel edytora umożliwia edycję strony, to umieszczone na niej hiperłącza nie są aktywne. Dla niektórych stron edytor pokazuje także teksty etykiet HTML, komentarze i polecenia języka Javascript (patrz rysunek 6.39). Jest to przydatne do sprawdzenia możliwości komponentu klasy `JEditorPane`, ale nie powinno być udostępniane w zwykłych programach.



Rysunek 6.39. *Panel edytora w trybie edycji*

> Domyślnie komponent klasy `JEditorPane` znajduje się w stanie edycji. Możemy to zmienić, wywołując metodę `editorPane.setEditable(false)`.

Przedstawione możliwości panelu edytora dają się łatwo wykorzystać. Do załadowania nowego dokumentu używamy metody `setPage`. Jej parametrem jest łańcuch znaków bądź obiekt klasy `URL`. Klasa `JEditorPane` jest klasą pochodną klasy `JTextComponent`. Dlatego możemy też użyć metody `setText`, jeśli chcemy umieścić w panelu zwykły, a nie sformatowany tekst.

Aby nasłuchiwać zdarzeń wyboru hiperłącza, tworzymy obiekt implementujący interfejs `HyperlinkListener`. Interfejs ten zawiera tylko jedną metodę, `hyperlinkUpdate`, która wywoływana jest, gdy użytkownik przesuwa kursor myszy ponad hiperłączem lub wybiera je. Metoda posiada parametr typu `HyperlinkEvent`.

Aby dowiedzieć się o rodzaju zdarzenia, musimy wywołać metodę `getEventType`, która zwrócić może jedną z trzech poniższych wartości:

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

Pierwsza z wartości oznacza, że użytkownik kliknął łącze. W takiej sytuacji zwykle będziemy chcieli załadować nową stronę. Pozostałe wartości możemy wykorzystać na przykład do wyświetlania wskazówek, gdy użytkownik przemieszcza kursor ponad łączem.

> Pozostaje dla nas tajemnicą powód, dla którego w interfejsie `HyperlinkListener` nie zdefiniowano osobnych metod w celu obsługi różnych rodzajów zdarzeń.

Metoda `getURL` klasy `HyperlinkEvent` zwraca adres URL dla hiperłącza. Poniżej przykład kodu obiektu nasłuchującego hiperłącza, który umożliwia przeglądanie stron ładowanych przez użytkownika przy użyciu hiperłącza.

```
editorPane.addHyperlinkListener(new
   HyperlinkListener()
   {
      public void hyperlinkUpdate(HyperlinkEvent event)
      {
         if (event.getEventType()
            == HyperlinkEvent.EventType.ACTIVATED)
         {
            try
            {
               editorPane.setPage(event.getURL());
            }
            catch(IOException e)
            {
               editorPane.setText("Exception: " + e);
            }
         }
      }
});
```

Metoda obsługi zdarzenia pobiera po prostu odpowiedni adres URL i aktualizuje zawartość panelu edytora. Metoda `setPage` może wyrzucić wyjątek `IOException`. W takiej sytuacji wyświetlamy w panelu edytora informację o błędzie w postaci zwykłego tekstu.

Program, którego tekst źródłowy zawiera listing 6.15, wykorzystuje wszystkie możliwości klasy `JEditorPane`, które przydatne są do zbudowania systemu pomocy opartego o pliki w formacie HTML. Implementacja klasy `JEditorPane` jest bardziej skomplikowana niż w przypadku komponentów drzewa bądź tabeli. Jeśli jednak nie wykorzystujemy tej klasy do tworzenia własnego edytora tekstu, to szczegółami tej implementacji nie musimy się interesować.

Listing 6.15. *EditorPaneTest.java*

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

/**
   Program demonstrujący wyświetlanie stron HTML
   za pomocą panelu edytora.
```

```
*/
public class EditorPaneTest
{
   public static void main(String[] args)
   {
      JFrame frame = new EditorPaneFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca panel edytora, pole tekstowe i przycisk Load
   umożliwiające wprowadzenie adresu URL i załadowanie strony,
   a także przycisk Back umożliwiający powrót do poprzedniej strony.
*/
class EditorPaneFrame extends JFrame
{
   public EditorPaneFrame()
   {
      setTitle("EditorPaneTest");
      setSize(WIDTH, HEIGHT);

      final Stack urlStack = new Stack();
      final JEditorPane editorPane = new JEditorPane();
      final JTextField url = new JTextField(30);

      // instaluje obiekt nasłuchujący hiperłącza

      editorPane.setEditable(false);
      editorPane.addHyperlinkListener(new
         HyperlinkListener()
         {
            public void hyperlinkUpdate(HyperlinkEvent event)
            {
               if (event.getEventType()
                  == HyperlinkEvent.EventType.ACTIVATED)
               {
                  try
                  {
                     // zapamiętuje adres URL dla potrzeb przycisku Back
                     urlStack.push(event.getURL().toString());
                     // prezentuje adres URL w polu tekstowym
                     url.setText(event.getURL().toString());

                     editorPane.setPage(event.getURL());
                  }
                  catch(IOException e)
                  {
                     editorPane.setText("Exception: " + e);
                  }
               }
            }
         });

      // pole wyboru umożliwiające włączenie trybu edycji

      final JCheckBox editable = new JCheckBox();
      editable.addActionListener(new
```

```
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            editorPane.setEditable(editable.isSelected());
         }
      });

   // tworzy obiekt nasłuchujący przycisku Load

   ActionListener listener = new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            try
            {
               // zapamiętuje adres URL dla potrzeb przycisku Back
               urlStack.push(url.getText());

               editorPane.setPage(url.getText());
            }
            catch(IOException e)
            {
               editorPane.setText("Exception: " + e);
            }
         }
      };

   JButton loadButton = new JButton("Load");
   loadButton.addActionListener(listener);
   url.addActionListener(listener);

   // tworzy obiekt nasłuchujący przycisku Back

   JButton backButton = new JButton("Back");
   backButton.addActionListener(new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            if (urlStack.size() <= 1) return;
            try
            {
               // pobiera adres URL
               urlStack.pop();
               // prezentuje adres URL w polu tekstowym
               String urlString = (String)urlStack.peek();
               url.setText(urlString);

               editorPane.setPage(urlString);
            }
            catch(IOException e)
            {
               editorPane.setText("Exception: " + e);
            }
         }
      });
```

```
        Container contentPane = getContentPane();
        contentPane.add(new JScrollPane(editorPane),
           BorderLayout.CENTER);

        // umieszcza wszystkie komponenty na głównym panelu okna

        JPanel panel = new JPanel();
        panel.add(new JLabel("URL"));
        panel.add(url);
        panel.add(loadButton);
        panel.add(backButton);
        panel.add(new JLabel("Editable"));
        panel.add(editable);

        contentPane.add(panel, BorderLayout.SOUTH);
     }

     private static final int WIDTH = 600;
     private static final int HEIGHT = 400;
  }
```

javax.swing.JEditorPane

- `void setPage(URL url)` **ładuje stronę o adresie** `url` **do panelu edytora.**
- `void addHyperlinkListener(HyperlinkListener listener)` **instaluje obiekt nasłuchujący panelu edytora.**

javax.swing.event.HyperlinkListener

- `void hyperlinkUpdate(HyperlinkEvent event)` **wywoływana, gdy hiperłącze zostanie wybrane.**

javax.swing.HyperlinkEvent

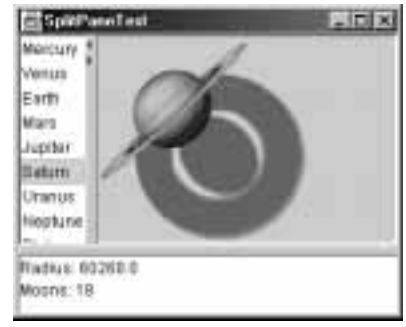
- `URL getURL()` **zwraca adres URL dla wybranego hiperłącza.**

Organizatory komponentów

Omówienie zaawansowanych możliwości biblioteki Swing zakończymy przedstawieniem komponentów, które pomagają programiście w organizacji innych komponentów. Należą do nich *panele dzielone* umożliwiające podział ich obszaru na wiele części, których rozmiary można regulować, *panele z zakładkami* pozwalające na przeglądanie wielu paneli i *panele pulpitu* ułatwiające implementację aplikacji posiadających wiele *ramek wewnętrznych*.

Panele dzielone

Panele dzielone umożliwiają podział ich obszaru na dwie części. Linia podziału panelu może być zmieniana. Rysunek 6.40 pokazuje ramkę zawierającą dwa panele dzielone. Panel zewnętrzny podzielony został poziomo, w jego dolnej części umieszczono obszar tekstowy, a w górnej — kolejny panel dzielony. Ten ostatni podzielony został pionowo. W jego lewej części umieszczono listę, a w prawej — etykietę zawierającą obrazek.



Rysunek 6.40. *Ramka zawierająca dwa zagnieżdżone panele*

Tworząc panel dzielony, musimy określić sposób jego podziału i dostarczyć komponenty, które umieszczone zostaną w poszczególnych częściach panelu.

```
JSplitPane innerPane =
   new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
```

I to wszystko. Możemy jeszcze dodać do linii podziału ikony, które umożliwią maksymalizację obszaru wybranej części panelu. Ikony te widać w górnej części linii podziału na rysunku 6.40. W przypadku wyglądu Metal są one reprezentowane za pomocą trójkątów. Wybranie jednego z nich powoduje maksymalizację obszaru części panelu w kierunku wskazywanym przez wierzchołek trójkąta.

Dodanie tej właściwości linii podziału możliwe jest za pomocą poniższej metody.

```
innerPane.setOneTouchExpandable(true);
```

Inna możliwość polega na włączeniu odrysowywania zawartości części paneli podczas przesuwania linii podziału. Jest to przydatne w niektórych sytuacjach, ale zawsze powoduje spowolnienie działania linii podziału. Możliwość tę możemy włączyć za pomocą wywołania:

```
innerPane.setContinuousLayout(true);
```

W naszym przykładowym programie dolna linia podziału posiada domyślne właściwości (brak ciągłego odrysowywania). Jej przeciąganie powoduje jedynie przesuwanie się ciemnej linii. Dopiero jej docelowe ustawienie powoduje odrysowanie komponentów.

Działanie programu z listingu 6.16 jest bardzo proste. Wypełnia on komponent listy nazwami planet. Wybór jednej z nich sprawia, że w prawej części panelu wyświetlany jest obrazek planety, a w dolnej — jej opis. Polecamy wypróbowanie i porównanie działania obu linii podziału.

Listing 6.16. *SplitPaneTest.java*

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

/**
   Program demonstrujący wykorzystanie komponentu panelu dzielonego.
*/
public class SplitPaneTest
{
   public static void main(String[] args)
   {
      JFrame frame = new SplitPaneFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca dwa panele dzielone wyświetlające opis planet.
*/
class SplitPaneFrame extends JFrame
{
   public SplitPaneFrame()
   {
      setTitle("SplitPaneTest");
      setSize(WIDTH, HEIGHT);

      // tworzy komponenty dla prezentacji nazw i opisu planet
      // oraz ich obrazków

      final JList planetList = new JList(planets);
      final JLabel planetImage = new JLabel();
      final JTextArea description = new JTextArea();

      planetList.addListSelectionListener(new
         ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent event)
            {
               Planet value
                  = (Planet)planetList.getSelectedValue();

               // aktualizuje obrazek i opis

               planetImage.setIcon(value.getImage());
               description.setText(value.getDescription());
            }
         });

      // tworzy panele dzielone

      JSplitPane innerPane
         = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
            planetList, planetImage);
```

```
      innerPane.setContinuousLayout(true);
      innerPane.setOneTouchExpandable(true);

      JSplitPane outerPane
         = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
            innerPane, description);

      getContentPane().add(outerPane, BorderLayout.CENTER);
   }

   private Planet[] planets =
      {
         new Planet("Mercury", 2440, 0),
         new Planet("Venus", 6052, 0),
         new Planet("Earth", 6378, 1),
         new Planet("Mars", 3397, 2),
         new Planet("Jupiter", 71492, 16),
         new Planet("Saturn", 60268, 18),
         new Planet("Uranus", 25559, 17),
         new Planet("Neptune", 24766, 8),
         new Planet("Pluto", 1137, 1),
      };
   private static final int WIDTH = 300;
   private static final int HEIGHT = 200;
}

/**
   Klasa reprezentująca planety.
*/
class Planet
{
   /**
      Tworzy obiekt reprezentujący planetę.
      @param n nazwa planety
      @param r promień planety
      @param m liczba księżyców
   */
   public Planet(String n, double r, int m)
   {  name = n;
      radius = r;
      moons = m;
      image = new ImageIcon(name + ".gif");
   }

   public String toString()
   {
      return name;
   }

   /**
      Pobiera opis planety.
      @return opis
   */
   public String getDescription()
   {
      return "Radius: " + radius + "\nMoons: " + moons + "\n";
   }

   /**
```

```
      Pobiera obrazek planety.
      @return obrazek
   */
   public ImageIcon getImage()
   {
      return image;
   }

   private String name;
   private double radius;
   private int moons;
   private ImageIcon image;
}
```

javax.swing.JSplitPane

- JSplitPane()
- JSplitPane(int direction)
- JSplitPane(int direction, boolean continuousLayout)
- JSplitPane(int direction, Component first, Component second)
- JSplitPane(int direction, boolean continuousLayout, Component first, Component second)

  Tworzą nowy panel dzielony.

  Parametry: direction — **jedna z wartości** JSplitPane.HORIZONTAL_SPLIT **lub** JSplitPane.VERTICAL_SPLIT,

  continuousLayout — **wartość** true, **jeśli komponenty mają być odrysowywane podczas przesuwania linii podziału,**

  first, second — **komponenty, które mają być umieszczone w częściach panelu.**

- boolean isOneTouchExpandable()
- void setOneTouchExpandable(boolean b)

  Umożliwiają sprawdzenie oraz włączenie właściwości linii podziału polegającej na możliwości maksymalizacji części panelu.

- boolean isContinuousLayout()
- void setContinuousLayout(boolean b)

  Umożliwiają sprawdzenie oraz włączenie właściwości linii podziału polegającej na odrysowywaniu zawartości komponentów panelu podczas przesuwania linii podziału.

- void setLeftComponent(Component c)
- void setTopComponent(Component c)

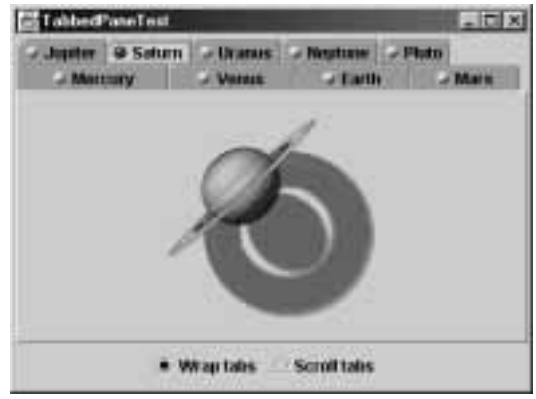
  Obie metody dają ten sam efekt — umieszczają komponent c **w pierwszej części panelu.**

- `void setRightComponent(Component c)`
- `void setBottomComponent(Component c)`

  Obie metody dają ten sam efekt — umieszczają komponent `c` w drugiej części panelu.

Panele z zakładkami

Panele z zakładkami umożliwiają uporządkowanie zawartości złożonych okien dialogowych. Pozwalają także na przeglądanie zestawu dokumentów lub obrazów (patrz rysunek 6.41). Takie będzie też ich zastosowanie w naszym przykładowym programie.



Rysunek 6.41.
Panel z zakładkami

Tworząc panel z zakładkami, konstruujemy najpierw obiekt klasy `JTabbedPane`, a następnie dodajemy do niego zakładki.

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

Ostatni parametr metody `add` jest komponentem, który umieszczony zostanie na zakładce. Jeśli chcemy, by zakładka zawierała wiele komponentów, to najpierw musimy umieścić je w kontenerze, na przykład klasy `JPanel`.

Ikona zakładki jest opcjonalna. Istnieje wersja metody `add`, która nie wymaga tego parametru:

```
tabbedPane.addTab(title, component);
```

Nową zakładkę możemy także umieścić między już istniejącymi, korzystając z metody `insertTab`:

```
tabbedPane.insertTab(title, icon, component, index);
```

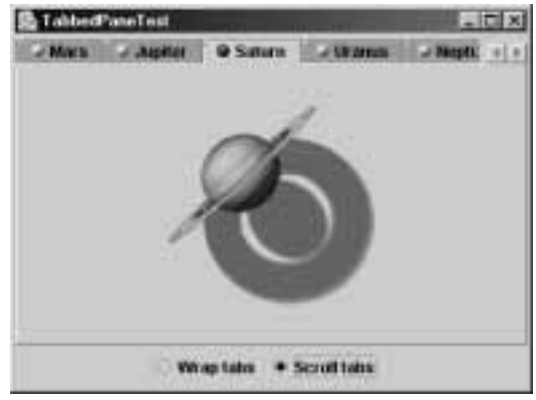
Natomiast poniższe wywołanie spowoduje usunięcie zakładki:

```
tabbedPane.removeTabAt(index);
```

Umieszczenie w panelu nowej zakładki nie powoduje, że staje się ona automatycznie widoczna. W tym celu musimy wybrać ją za pomocą metody `setSelectedIndex`. Poniższe wywołanie pokazuje, w jaki sposób spowodować wyświetlenie dodanej właśnie zakładki:

```
tabbedPane. setSelectedIndex(tabbedPane.getTabCount() - 1);
```

Jeśli panel zawiera większą liczbę zakładek, to mogą one zajmować zbyt dużo miejsca. Dlatego też w SDK 1.4 wprowadzono możliwość przewijania pojedynczego wiersza zakładek (patrz rysunek 6.42).



Rysunek 6.42. *Panel z przewijaniem zakładek*

Możemy wybrać ułożenie wszystkich zakładek w kilku wierszach bądź przewijanie ich w jednym wierszu, wywołując odpowiednio:

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

lub

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

Program przykładowy pokazuje zastosowanie pewnej techniki przydatnej w przypadku paneli z zakładkami. Polega ona na umieszczeniu komponentu na zakładce, dopiero w momencie gdy ma zostać ona pokazana. W naszym programie oznacza to, że obrazek planety zostanie umieszczony na zakładce dopiero po jej wybraniu.

Aby uzyskać powiadomienie o wyborze zakładki, musimy zainstalować obiekt nasłuchujący `ChangeListener`. Zwróćmy uwagę, że obiekt ten instalujemy dla panelu, a nie dla jednego z jego komponentów.

```
tabbedPane.addChangeListener(listener);
```

Gdy użytkownik wybierze zakładkę, to wywołana zostanie metoda `stateChanged` obiektu nasłuchującego. Korzystając z metody `getSelectedIndex`, możemy dowiedzieć się, która zakładka została wybrana:

```
public void stateChanged(ChangeEvent event)
{
   int n = tabbedPane.getSelectedIndex();
   . . .
}
```

W programie z listingu 6.17 wszystkie komponenty zakładek mają na początku wartość `null`. Kiedy zakładka jest wybierana, to sprawdzamy, czy jej komponent nadal jest wartością `null`. Jeśli tak, to ładujemy obrazek. (Dzieje się to, zanim wybrana zakładka zostanie pokazana i wobec tego użytkownik nie zobaczy pustej zakładki). Zmieniamy także ikonę zakładki z żółtej na czerwoną, aby zaznaczyć, które zakładki były już przeglądane.

Listing 6.17. *TabbedPaneTest.java*

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

/**
   Program demonstrujący zastosowanie panelu z zakładkami.
*/
public class TabbedPaneTest
{
   public static void main(String[] args)
   {
      JFrame frame = new TabbedPaneFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka zawierająca panel z zakładkami oraz przyciski
   umożliwiające przełączanie sposobu prezentacji zakładek.
*/
class TabbedPaneFrame extends JFrame
{
   public TabbedPaneFrame()
   {
      setTitle("TabbedPaneTest");
      setSize(WIDTH, HEIGHT);

      final JTabbedPane tabbedPane = new JTabbedPane();
      // załadowanie komponentów zakładek odkładamy
      // do momentu ich pierwszej prezentacji

      ImageIcon icon = new ImageIcon("yellow-ball.gif");

      tabbedPane.addTab("Mercury", icon, null);
      tabbedPane.addTab("Venus", icon, null);
      tabbedPane.addTab("Earth", icon, null);
      tabbedPane.addTab("Mars", icon, null);
      tabbedPane.addTab("Jupiter", icon, null);
      tabbedPane.addTab("Saturn", icon, null);
      tabbedPane.addTab("Uranus", icon, null);
      tabbedPane.addTab("Neptune", icon, null);
      tabbedPane.addTab("Pluto", icon, null);

      getContentPane().add(tabbedPane, "Center");

      tabbedPane.addChangeListener(new
         ChangeListener()
         {
            public void stateChanged(ChangeEvent event)
            {
```

```
                // sprawdza, czy na zakładce umieszczony jest już komponent
                if (tabbedPane.getSelectedComponent() == null)
                {
                   // ładuje obrazek

                   int n = tabbedPane.getSelectedIndex();
                   String title = tabbedPane.getTitleAt(n);
                   ImageIcon planetIcon
                      = new ImageIcon(title + ".gif");
                   tabbedPane.setComponentAt(n,
                      new JLabel(planetIcon));

                   // zaznacza, że zakładka była już przeglądana

                   tabbedPane.setIconAt(n,
                      new ImageIcon("red-ball.gif"));
                }
             }
          });

       JPanel buttonPanel = new JPanel();
       ButtonGroup buttonGroup = new ButtonGroup();
       JRadioButton wrapButton = new JRadioButton("Wrap tabs");
       wrapButton.addActionListener(new
          ActionListener()
          {
             public void actionPerformed(ActionEvent event)
             {
                tabbedPane.setTabLayoutPolicy(
                   JTabbedPane.WRAP_TAB_LAYOUT);
             }
          });
       buttonPanel.add(wrapButton);
       buttonGroup.add(wrapButton);
       wrapButton.setSelected(true);
       JRadioButton scrollButton
          = new JRadioButton("Scroll tabs");
       scrollButton.addActionListener(new
          ActionListener()
          {
             public void actionPerformed(ActionEvent event)
             {
                tabbedPane.setTabLayoutPolicy(
                   JTabbedPane.SCROLL_TAB_LAYOUT);
             }
          });
       buttonPanel.add(scrollButton);
       buttonGroup.add(scrollButton);
       getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    }

    private static final int WIDTH = 400;
    private static final int HEIGHT = 300;
}
```

javax.swing.JTabbedPane

- `JTabbedPane()`
- `JTabbedPane(int placement)`

  Tworzą panel z zakładkami.

  Parametry: `placement` **jedna z wartości** `SwingConstants.TOP`, `SwingConstants.LEFT`, `SwingConstants.RIGHT` **lub** `SwingConstants.BOTTOM`.

- `void addTab(String title, Component component)`
- `void addTab(String title, Icon icon, Component c)`
- `void addTab(String title, Icon icon, Component c, String toolTip)`

  Dodają zakładkę do panelu.

- `void insertTab(String title, Icon icon, Component c, String toolTip, int index)` **umieszcza nową zakładkę na pozycji o podanym indeksie.**
- `void removeTabAt(int index)` **usuwa zakładkę o podanym indeksie.**
- `void setSelectedIndex(int index)` **wybiera zakładkę o podanym indeksie.**
- `int getSelectedIndex()` **zwraca indeks wybranej zakładki.**
- `Component getSelectedComponent()` **zwraca komponent wybranej zakładki.**
- `String getTitleAt(int index)`
- `void setTitleAt(int index, String title)`
- `Icon getIconAt(int index)`
- `void setIconAt(int index, Icon icon)`
- `Component getComponentAt(int index)`
- `void setComponentAt(int index, Component c)`

  Pobierają lub ustawiają tytuł, ikonę lub komponent zakładki o danym indeksie.

- `int indexOfTab(Icon icon)`
- `int indexOfTab(String title)`
- `int indexOfTab(Component c)`

  Zwracają indeks zakładki o danym tytule, ikonie bądź komponencie.

- `int getTabCount()` **zwraca liczbę zakładek panelu.**
- `void setTabLayoutPolicy(int policy)` **ustala sposób prezentacji zakładek — w wielu wierszach lub jednym przewijanym.**

  Parametry: `policy` **jedna z wartości** `JTabbedPane.WRAP_TAB_LAYOUT` **lub** `JTabbedPane.SCROLL_TAB_LAYOUT`.

- `void addChangeListener(ChangeListener listener)` **instaluje obiekt nasłuchujący powiadamiany w momencie wybrania przez użytkownika zakładki.**

Panele pulpitu i ramki wewnętrzne

Aplikacje często prezentują informacje, korzystając z wielu okien umieszczonych we wspólnej ramce. Zwinięcie takiej ramki do ikony równoznaczne jest z ukryciem zawartości wszystkich jej okien. W środowisku Windows ten sposób działania interfejsu użytkownika nazwany został *MDI* (*Multiple Document Interface*). Rysunek 6.43 pokazuje typową aplikację korzystającą z interfejsu MDI.

Rysunek 6.43. *Aplikacja korzystająca z MDI*

Do niedawna był to jeden z popularniejszych sposobów tworzenia interfejsu aplikacji, ale ostatnio wykorzystywany jest rzadziej. Większość przeglądarek internetowych otwiera strony internetowe, używając ramek tego samego poziomu co główna ramka programu. (Wyjątkiem jest tutaj przeglądarka Opera pokazana na rysunku 6.43). Który ze sposobów organizacji interfejsu użytkownika jest lepszy? Oba posiadają zalety i wady. Wykorzystanie MDI pozwala ograniczyć natłok okien otwieranych przez różne programy. Natomiast użycie wielu okien programu umożliwia posłużenie się przyciskami i kombinacjami klawiszy udostępnianymi przez system okienkowy do przełączania się pomiędzy oknami.

W przypadku aplikacji w języku Java, z natury niezależnych od platformy, nie możemy polegać na usługach systemu okienkowego, a więc zarządzanie własnymi oknami przez samą aplikację ma większy sens.

Rysunek 6.44 pokazuje aplikację Java, której okno zawiera trzy wewnętrzne ramki. Dwie z nich posiadają ikony umożliwiające ich maksymalizację bądź zwinięcie do ikony. Trzecia została zwinięta do ikony.

W przypadku wyglądu komponentów Metal ramki wewnętrzne posiadają wyróżniony obszar, który umożliwia ich „uchwycenie" i przesuwanie. Uchwycenie narożnika ramki umożliwia natomiast zmianę jej rozmiarów.

Rysunek 6.44. *Aplikacja w języku Java posiadająca trzy wewnętrzne ramki*

Aby skorzystać z możliwości zarządzania wewnętrznymi ramkami należy kolejno wykonać następujące kroki.

1. Tworzymy dla aplikacji zwykłą ramkę klasy `JFrame`.

2. Umieszczamy w niej panel klasy `JDesktopPane`.

```
desktop = new JDesktopPane();
setContentPane(desktop);
```

3. Tworzymy obiekty klasy `JInternalFrame` reprezentujące wewnętrzne ramki, podając przy tym, czy mają zawierać ikony zmiany rozmiarów i zamknięcia. Zwykle będziemy chcieli, by ramki posiadały wszystkie te ikony.

```
JInternalFrame iframe = new JInternalFrame(title,
   true,   // zmiana rozmiarów
   true,   // możliwość zamknięcia
   true,   // maksymalizacja
   true); // zwinięcie do ikony
```

4. Umieszczamy komponenty w ramkach wewnętrznych.

```
iframe.getContentPane().add(c);
```

5. Przypisujemy ramkom wewnętrznym ikonę, która pokazywana będzie w lewym górnym rogu ramki.

```
iframe.setFrameIcon(icon);
```

> W obecnej wersji implementacji wyglądu Metal ikona ramki nie jest wyświetlana, gdy ramka jest zwinięta.

6. Określamy rozmiary wewnętrznych ramek. Podobnie jak w przypadku zwykłych ramek, ich początkowy rozmiar wynosi 0 na 0 pikseli. Ponieważ nie chcemy, by ramki wewnętrzne przykrywały się wzajemnie, to powinniśmy wybrać dla nich także różne pozycje początkowe. Metoda `reshape` umożliwia określenie początkowej pozycji i rozmiarów ramki:

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

7. Podobnie jak w przypadku zwykłych ramek, musimy jeszcze je pokazać.

```
iframe.setVisible(true);
```

> We wczesnych wersjach biblioteki Swing ramki wewnętrzne były pokazywane automatycznie i wywołanie metody `setVisible` nie było konieczne.

8. Dodajemy ramki do panelu `JDesktopPane`:

```
desktop.add(iframe);
```

9. Wybieramy jedną z dodanych ramek. W przypadku ramek wewnętrznych tylko wybrana ramka otrzymuje informacje o stanie klawiatury. Wygląd Metal wyróżnia wybraną ramkę za pomocą niebieskiego paska tytułu, podczas gdy w pozostałych ramkach ma on kolor szary. Metoda `setSelectedFrame` umożliwia wybranie ramki. Jednak wywołanie tej metody może zostać „zawetowane" przez aktualnie wybraną ramkę. Spowoduje to wyrzucenie przez metodę `setSelectedFrame` wyjątku `PropertyVetoException`, który musimy obsłużyć.

```
try
{
   iframe.setSelected(true);
}
catch(PropertyVetoException e)
{
   // próba wyboru została zawetowana
}
```

10. Umieszczamy kolejną ramkę poniżej, tak by nie zasłaniała istniejącej ramki. Właściwą odległością będzie zwykle wysokość paska tytułowego ramki, którą możemy uzyskać w poniższy sposób:

```
int frameDistance =
   iframe.getHeight() - iframe.getContentPane().getHeight();
```

11. Wykorzystujemy wyliczoną odległość w celu ustalenia pozycji kolejnej ramki.

```
nextFrameX += frameDistance;
nextFrameY += frameDistance;
if (nextFrameX + width > desktop.getWidth())
   nextFrameX = 0;
if (nextFrameY + height > desktop.getHeight())
   nextFrameY = 0;
```

Rozmieszczenie kaskadowe i sąsiadujące

W systemie Windows istnieją standardowe komendy umożliwiające uzyskanie rozmieszczenia *kaskadowego* lub *sąsiadującego* okien (patrz rysunki 6.45 i 6.46). Klasy `JDesktopPane` i `JInternalFrame` biblioteki Swing nie udostępniają niestety odpowiednich metod. Program, którego tekst źródłowy umieściliśmy w listingu 6.18, pokazuje sposób samodzielnej implementacji takich metod.

Rysunek 6.45. *Rozmieszczenie kaskadowe ramek wewnętrznych*

Rysunek 6.46. *Rozmieszczenie sąsiadujące ramek wewnętrznych*

Rozmieszczenie kaskadowe charakteryzuje się jednakowym rozmiarem okien i przesunięciem ich pozycji. Metoda `getAllFrames` klasy `JDesktopPane` zwraca tablicę wszystkich wewnętrznych ramek.

```
JInternalFrame[] frames = desktop.getAllFrames();
```

Musimy jednak zwrócić uwagę na stan, w jakim one się znajdują. Ramka wewnętrzna może znajdować się w jednym z trzech stanów. Oto one:

- ikona,
- pośredni, umożliwiający zmianę rozmiarów ramki,
- w pełni rozwinięty.

Korzystając z metody `isIcon`, możemy dowiedzieć się, które ramki zwinięte są do ikony i pominąć je podczas rozmieszczania. Jeśli ramka znajduje się w stanie w pełni rozwiniętym, to musimy najpierw sprowadzić ją do stanu pośredniego, wywołując `setMaximum(false)`. Jest to kolejna metoda, której wywołanie może zostać zawetowane. Trzeba więc obsłużyć wyjątek `PropertyVetoException`.

Poniższa pętla rozmieszcza kaskadowo wszystkie wewnętrzne ramki panelu pulpitu:

```
for (int i = 0; i < frames.length; i++)
{
   if (!frames[i].isIcon())
   {
      try
      {
         // próbuje przeprowadzić ramki do stanu pośredniego
         // co  może zostać zawetowane
         frames[i].setMaximum(false);
         frames[i].reshape(x, y, width, height);
         x += frameDistance;
         y += frameDistance;
         // zawija dookoła brzegu pulpitu
         if (x + width > desktop.getWidth()) x = 0;
         if (y + height > desktop.getHeight()) y = 0;
      }
      catch(PropertyVetoException e)
      {}
   }
}
```

Uzyskanie rozmieszczenia sąsiadującego okazuje się nieco bardziej skomplikowane, szczególnie jeśli liczba ramek nie jest kwadratem innej liczby. Najpierw musimy uzyskać liczbę ramek, które nie są zwinięte do ikony. Następnie obliczyć liczbę kolumn jako

```
int cols = (int)Math.sqrt(frameCount);
```

oraz liczbę wierszy jako

```
int rows = frameCount / cols;
```

z tym wyjątkiem, że ostatnia kolumna

```
extra = frameCount % cols;
```

posiadać będzie `rows + 1` wierszy.

Poniższa pętla rozmieszcza sąsiadująco wszystkie wewnętrzne ramki panelu pulpitu:

```
int width = desktop.getWidth() / cols;
int height = desktop.getHeight() / rows;
int r = 0;
```

```
int c = 0;
for (int i = 0; i < frames.length; i++)
{
   if (!frames[i].isIcon())
   {
      try
      {
       frames[i].setMaximum(false);
       frames[i].reshape(c * width,
          r * height, width, height);
       r++;
       if (r == rows)
       {
          r = 0;
          c++;
          if (c == cols - extra)
          {
             // rozpoczyna dodatkowy wiersz
             rows++;
             height = desktop.getHeight() / rows;
          }
       }
    }
    catch(PropertyVetoException e)
    {}
   }
}
```

Przykładowy program prezentuje także inną typową operację związana z ramkami: wybór kolejnych ramek, które nie są zwinięte do ikony. Klasa `JDesktopPane` nie udostępnia metody zwracającej wybraną ramkę. Musimy więc sami wywołać metodę `isSelected` dla wszystkich ramek tak długo, aż znajdziemy tę wybraną. Następnie wyszukujemy kolejną ramkę, która nie jest zwinięta do ikony i próbujemy ją wybrać.

```
frames[next].setSelected(true);
```

Także i to wywołanie może wyrzucić wyjątek `PropertyVetoException`. W takim przypadku musimy kontynuować poszukiwanie kolejnej ramki. Jeśli wrócimy w ten sposób do ramki wyjściowej, oznacza to, że żadna inna ramka nie mogła być wybrana. Poniżej kompletna pętla realizująca opisane działanie:

```
for (int i = 0; i < frames.length; i++)
{
   if (frames[i].isSelected())
   {
      // znajduje ramkę, która nie jest zwinięta do ikony
      // i może zostać wybrana
     try
     {
        int next = (i + 1) % frames.length;
        while (next != i && frames[next].isIcon())
           next = (next + 1) % frames.length;
        if (next == i) return;
           // pozostałe ramki są zwinięte do ikon lub zgłosiły weto do wyboru
        frames[next].setSelected(true);
        frames[next].toFront();
        return;
```

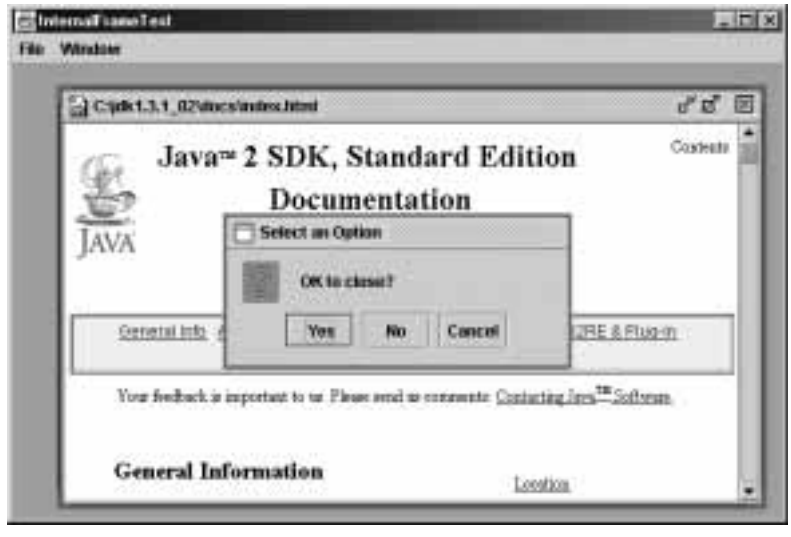
```
            }
            catch(PropertyVetoException e)
            {}
        }
    }
```

Zgłaszanie weta do zmiany właściwości

Po lekturze poprzednich przykładów możemy zastanawiać się, w jaki sposób ramka zgłasza weto. Klasa `JInternalFrame` wykorzystuje ogólny mechanizm JavaBeans w celu monitorowania zmian właściwości. Mechanizm ten omawiamy szczegółowo w rozdziale 8. Teraz będziemy chcieli jedynie pokazać, w jaki sposób ramki mogą zgłaszać weto do zmian ich właściwości.

Zwykle ramki nie zgłaszają weta, aby oprotestować zwinięcie ich do ikony bądź utratę wyboru. Typową sytuacją dla takiego zachowania będzie natomiast *zamknięcie* ramki. Ramkę zamykamy, korzystając z metody `setClosed` klasy `JInternalFrame`. Ponieważ może ona zostać zawetowana, to wywołuje najpierw wszystkie *obiekty nasłuchujące weta zmiany*. Umożliwia to tym obiektom wyrzucenie wyjątku `PropertyVetoException` i tym samym zakończenie wykonywania metody, zanim podejmie ona działania zmierzające do zamknięcia ramki.

W przykładowym programie próba zamknięcia ramki powoduje pojawienie się okna dialogowego w celu potwierdzenia zamknięcia ramki przez użytkownika (patrz rysunek 6.47). Jeśli użytkownik nie zgodzi się, to ramka pozostanie otwarta.



Rysunek 6.47. *Użytkownik może zawetować zamknięcie ramki*

A oto sposób uzyskania takiego powiadomienia.

1. Do każdej ramki dodajemy obiekt nasłuchujący. Obiekt ten musi należeć do klasy implementującej interfejs `VetoableChangeListener`. Najlepiej dodać go zaraz po utworzeniu ramki. W przykładowym programie tworzymy go i dodajemy w klasie ramki. Inną możliwością jest wykorzystanie anonimowej klasy wewnętrznej.

```
iframe.addVetoableChangeListener(listener);
```

2. Implementujemy metodę `vetoableChange`, która jest jedyną metodą definiowaną przez interfejs `VetoableChangeListener`. Jej parametrem jest obiekt klasy `PropertyChangeEvent`. Korzystając z jego metody `getPropertyName`, uzyskujemy nazwę właściwości, która ma zostać zmieniona — na przykład `"closed"`, jeśli wetowane jest wywołanie metody `setClosed(true)`. Jak pokażemy w rozdziale 8., nazwa właściwości uzyskiwana jest przez usunięcie prefiksu `"set"` z nazwy metody i zmianę wielkości następnej litery nazwy.

   Metodę `getNewValue` wykorzystujemy w celu uzyskania proponowanej wartości właściwości.

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(Boolean.TRUE))
{
   prosi użytkownika o potwierdzenie zamknięcia ramki
}
```

3. Wyrzucamy wyjątek `PropertyVetoException`, aby uniemożliwić zmianę właściwości lub w przeciwnym razie oddajemy sterowanie.

```
class DesktopFrame extends JFrame
   implements VetoableChangeListener
{
   . . .
   public void vetoableChange(PropertyChangeEvent event)
      throws PropertyVetoException
      {
         . . .
         if (not ok)
            throw new PropertyVetoException("reason", event);
         // oddaje sterowanie, jeśli ok.
      }
}
```

Okna dialogowe ramek wewnętrznych

W przypadku ramek wewnętrznych nie powinniśmy korzystać z klasy `JDialog` w celu tworzenia okien dialogowych, ponieważ:

- ich otwarcie wiąże się ze znacznym nakładem i utworzeniem nowej ramki systemu okienkowego,
- system okienkowy nie potrafi określić właściwej pozycji okna dialogowego w stosunku do ramki, która je otworzyła.

Dlatego też dla prostych okien dialogowych wykorzystywać będziemy metodę `showInternal`*`Xxx`*`Dialog` klasy `JOptionPane`. Działa ona dokładnie tak jak metoda `show`*`Xxx`*`Dialog`, ale tworzy proste okno dialogowe nad właściwą ramką wewnętrzną.

W przypadku bardziej złożonych okien dialogowych możemy skorzystać z klasy `JInternalFrame`, która nie umożliwia jednak tworzenia okien modalnych.

W naszym programie korzystamy z okna dialogowego w celu potwierdzenia przez użytkownika zamknięcia ramki.

```
int result
   = JOptionPane.showInternalConfirmDialog(
      iframe, "OK to close?");
```

> Jeśli chcemy zostać po prostu powiadomieni o zamknięciu okna, to nie musimy korzystać z mechanizmu zgłaszania weta. Wystarczy jedynie zainstalować obiekt nasłuchujący klasy `InternalFrameListener`. Zachowuje się on podobnie do obiektu nasłuchującego klasy `WindowListener`. Gdy zamykana jest wewnętrzna ramka, to wywoływana jest jego metoda `internalFrameClosing` będąca odpowiednikiem metody `windowClosing`. Pozostałe sześć powiadomień o zmianach ramki wewnętrznej (otwarcie (zamknięcie), zwinięcie do ikony (rozwinięcie), aktywacja (deaktywacja)) również odpowiada znanym metodom obiektów nasłuchujących zwykłych okien.

Przeciąganie zarysu ramki

Często krytykowaną cechą wewnętrznych ramek jest niska efektywność odrysowywania ich zawartości. Ujawnia się ona zwłaszcza podczas przeciągania ramek o złożonej zawartości.

Podobny efekt uzyskamy także dla zwykłych okien w przypadku kiepsko zaimplementowanego sterownika ekranu. Z reguły jednak przeciąganie zwykłych okien nawet z bardzo skomplikowaną zawartością jest efektywne, ponieważ obsługiwane jest sprzętowo.

Aby poprawić działanie przeciągania ramek wewnętrznych, możemy skorzystać z ich właściwości umożliwiającej przeciąganie jedynie zarysu ramki. Zawartość ramki jest w takim przypadku odrysowywana dopiero po jej umieszczeniu na pulpicie. Podczas przeciągania odrysowywany jest jedynie zarys ramki.

Aby włączyć możliwość przeciągania zarysu, wywołujemy poniższą metodę.

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

Możliwość ta stanowi odpowiednik odrysowywania linii podziału komponentów klasy `JSplitPane`.

> We wczesnych wersjach biblioteki Swing odrysowywanie zawartości ramek podczas przeciągania należało wyłączyć za pomocą poniższego wywołania.
>
> ```
> desktop.putClientProperty(
> "JDesktopPane.dragMode", "outline");
> ```

Nasz przykładowy program umożliwia zarządzanie odrysowywaniem zawartości ramek za pomocą pozycji menu *Window/Drag Outline*.

> Ramki wewnętrzne pulpitu zarządzane są przez klasę `DesktopManager`. Instalując innego menedżera pulpitu, możemy zaimplementować odmienne zachowanie pulpitu. Możliwości tej nie będziemy jednak omawiać w naszej książce.

Program z listingu 6.18 otwiera na pulpicie ramki zawierające strony HTML. Wybranie z menu opcji *File/Open* umożliwia umieszczenie zawartości wybranego pliku HTML w nowej ramce. Wybranie hiperłącza na stronie w ramce powoduje otwarcie nowej strony w osobnej ramce. Pozycje menu *Window/Cascade* i *Window/Tile* umożliwiają uzyskanie różnych rozmieszczeń ramek na pulpicie. Listing 6.18 kończy omówienie zaawansowanych możliwości pakietu Swing.

Listing 6.18. *InternalFrameTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
   Program demonstrujący wykorzystanie ramek wewnętrznych.
*/
public class InternalFrameTest
{
   public static void main(String[] args)
   {
      JFrame frame = new DesktopFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Ramka pulpitu zawierająca panele wyświetlające zawartość plików HTML.
*/
class DesktopFrame extends JFrame
{
   public DesktopFrame()
   {
      setTitle("InternalFrameTest");
      setSize(WIDTH, HEIGHT);

      desktop = new JDesktopPane();
      setContentPane(desktop);


      // tworzy menu

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      JMenu fileMenu = new JMenu("File");
      menuBar.add(fileMenu);
      JMenuItem openItem = new JMenuItem("Open");
      openItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               openFile();
```

```
            }
         });
      fileMenu.add(openItem);
      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);
      JMenu windowMenu = new JMenu("Window");
      menuBar.add(windowMenu);
      JMenuItem nextItem = new JMenuItem("Next");
      nextItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               selectNextWindow();
            }
         });
      windowMenu.add(nextItem);
      JMenuItem cascadeItem = new JMenuItem("Cascade");
      cascadeItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               cascadeWindows();
            }
         });
      windowMenu.add(cascadeItem);
      JMenuItem tileItem = new JMenuItem("Tile");
      tileItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               tileWindows();
            }
         });
      windowMenu.add(tileItem);
      final JCheckBoxMenuItem dragOutlineItem
         = new JCheckBoxMenuItem("Drag Outline");
      dragOutlineItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               desktop.setDragMode(dragOutlineItem.isSelected()
                  ? JDesktopPane.OUTLINE_DRAG_MODE
                  : JDesktopPane.LIVE_DRAG_MODE);
            }
         });
      windowMenu.add(dragOutlineItem);
   }
```

```
/**
   Tworzy wewnętrzną ramkę pulpitu.
   @param c komponent wewnątrz ramki wewnętrznej
   @param t tytuł ramki wewnętrznej.
*/
public void createInternalFrame(Component c, String t)
{
   final JInternalFrame iframe = new JInternalFrame(t,
      true, // zmiana rozmiarów
      true, // możliwość zamknięcia
      true, // maksymalizacja
      true); // zwinięcie do ikony

   iframe.getContentPane().add(c);
   desktop.add(iframe);

   iframe.setFrameIcon(new ImageIcon("document.gif"));

   // dodaje obiekt nasłuchujący, aby potwierdzić zamknięcie ramki
   iframe.addVetoableChangeListener(new
      VetoableChangeListener()
      {
         public void vetoableChange(PropertyChangeEvent event)
            throws PropertyVetoException
         {
            String name = event.getPropertyName();
            Object value = event.getNewValue();

            // sprawdza tylko próby zamknięcia ramki
            if (name.equals("closed")
               && value.equals(Boolean.TRUE))
            {
               // prosi użytkownika o potwierdzenie zamknięcia ramki
               int result
                  = JOptionPane.showInternalConfirmDialog(
                     iframe, "OK to close?");

               // jeśli użytkownik się nie zgodzi, to zgłasza weto
               if (result != JOptionPane.YES_OPTION)
                  throw new PropertyVetoException(
                     "User canceled close", event);
            }
         }
      });

   // ustala pozycje ramki
   int width = desktop.getWidth() / 2;
   int height = desktop.getHeight() / 2;
   iframe.reshape(nextFrameX, nextFrameY, width, height);

   iframe.show();

   // wybór ramki - może zostać zawetowany
   try
   {
      iframe.setSelected(true);
   }
   catch(PropertyVetoException e)
```

```
      {}

      /* jeśli pierwszy raz, to oblicza odległość pomiędzy ramkami
         rozmieszczonymi kaskadowo
      */

      if (frameDistance == 0)
         frameDistance = iframe.getHeight()
            - iframe.getContentPane().getHeight();

      // wyznacza pozycję kolejnej ramki

      nextFrameX += frameDistance;
      nextFrameY += frameDistance;
      if (nextFrameX + width > desktop.getWidth())
         nextFrameX = 0;
      if (nextFrameY + height > desktop.getHeight())
         nextFrameY = 0;
   }

   /**
      Rozmieszcza kaskadowo te ramki pulpitu, które nie są zwinięte do ikony.
   */
   public void cascadeWindows()
   {
      JInternalFrame[] frames = desktop.getAllFrames();
      int x = 0;
      int y = 0;
      int width = desktop.getWidth() / 2;
      int height = desktop.getHeight() / 2;

      for (int i = 0; i < frames.length; i++)
      {
         if (!frames[i].isIcon())
         {
            try
            {
               // próbuje przeprowadzić ramki do stanu pośredniego
               // co  może zostać zawetowane
               frames[i].setMaximum(false);
               frames[i].reshape(x, y, width, height);

               x += frameDistance;
               y += frameDistance;
               // zawija dookoła brzegu pulpitu
               if (x + width > desktop.getWidth()) x = 0;
               if (y + height > desktop.getHeight()) y = 0;
            }
            catch(PropertyVetoException e)
            {}
         }
      }
   }

   /**
     Rozmieszcza sąsiadująco te ramki pulpitu, które nie są zwiniete do ikony.
   */
   public void tileWindows()
   {
```

```
      JInternalFrame[] frames = desktop.getAllFrames();

      // zlicza ramki, które nie są zwinięte do ikony
      int frameCount = 0;
      for (int i = 0; i < frames.length; i++)
      {
         if (!frames[i].isIcon())
            frameCount++;
      }

      int rows = (int)Math.sqrt(frameCount);
      int cols = frameCount / rows;
      int extra = frameCount % rows;
         // liczba kolumn z dodatkowym wierszem

      int width = desktop.getWidth() / cols;
      int height = desktop.getHeight() / rows;
      int r = 0;
      int c = 0;
      for (int i = 0; i < frames.length; i++)
      {
         if (!frames[i].isIcon())
         {
            try
            {
               frames[i].setMaximum(false);
               frames[i].reshape(c * width,
                  r * height, width, height);
               r++;
               if (r == rows)
               {
                  r = 0;
                  c++;
                  if (c == cols - extra)
                  {
                     // rozpoczyna dodatkowy wiersz
                     rows++;
                     height = desktop.getHeight() / rows;
                  }
               }
            }
            catch(PropertyVetoException e)
            {}
         }
      }
   }

   /**
      Wybiera ramkę.
   */
   public void selectNextWindow()
   {
      JInternalFrame[] frames = desktop.getAllFrames();
      for (int i = 0; i < frames.length; i++)
      {
         if (frames[i].isSelected())
         {
```

```
            // znajduje ramkę, która nie jest zwinięta do ikony
            // i może zostać wybrana
            try
            {
               int next = (i + 1) % frames.length;
               while (next != i && frames[next].isIcon())
                  next = (next + 1) % frames.length;
               if (next == i) return;
                  // pozostałe ramki są zwinięte do ikon lub zgłosiły weto do wyboru
               frames[next].setSelected(true);
               frames[next].toFront();
               return;
            }
            catch(PropertyVetoException e)
            {}
         }
      }
   }

   /**
      Prosi użytkownika o otwarcie pliku HTML.
   */
   public void openFile()
   {
      // pozwala użytkownikowi wybrać plik

      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));
      chooser.setFileFilter(new
         javax.swing.filechooser.FileFilter()
         {
            public boolean accept(File f)
            {
               String fname = f.getName().toLowerCase();
               return fname.endsWith(".html")
                  || fname.endsWith(".htm")
                  || f.isDirectory();
            }
            public String getDescription()
            {
               return "HTML Files";
            }
         });
      int r = chooser.showOpenDialog(this);

      if (r == JFileChooser.APPROVE_OPTION)
      {
         // otwiera wybrany plik

         String filename = chooser.getSelectedFile().getPath();
         try
         {
            URL fileUrl = new URL("file:" + filename);
            createInternalFrame(createEditorPane(fileUrl),
               filename);
         }
         catch(MalformedURLException e)
         {
```

```
            }
         }
      }

      /**
         Tworzy panel edytora.
         @param u adres URL dokumentu HTML
      */
      public Component createEditorPane(URL u)
      {
         // tworzy panel edytora umożliwiający poruszanie się po hiperłączach

         JEditorPane editorPane = new JEditorPane();
         editorPane.setEditable(false);
         editorPane.addHyperlinkListener(new
            HyperlinkListener()
            {
               public void hyperlinkUpdate(HyperlinkEvent event)
               {
                  if (event.getEventType()
                     == HyperlinkEvent.EventType.ACTIVATED)
                     createInternalFrame(createEditorPane(
                        event.getURL()), event.getURL().toString());
               }
            });
         try
         {
            editorPane.setPage(u);
         }
         catch(IOException e)
         {
            editorPane.setText("Exception: " + e);
         }
         return new JScrollPane(editorPane);
      }

      private JDesktopPane desktop;
      private int nextFrameX;
      private int nextFrameY;
      private int frameDistance;

      private static final int WIDTH = 600;
      private static final int HEIGHT = 400;
   }
```

javax.swing.JDesktopPane

- `JInternalFrame[] getAllFrames()` **zwraca wszystkie ramki wewnętrzne panelu pulpitu.**
- `void setDragMode(int mode)` **określa sposób zachowania ramek wewnętrznych panelu podczas przeciągania (tylko zarys bądź także zawartość ramki).**

  Parametry: `mode` **jedna z wartości** `JDesktopPane.LIVE_DRAG_MODE` **lub** `JDesktopPane.OUTLINE_DRAG_MODE`.

javax.swing.JInternalFrame

- JInternalFrame()
- JInternalFrame(String title)
- JInternalFrame(String title, boolean resizable)
- JInternalFrame(String title, boolean resizable, boolean closable)
- JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)
- JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)

  Tworzą nową ramkę wewnętrzną.

  Parametry: `title` tytuł ramki,

  `resizable` wartość `true`, jeśli rozmiary ramki mogą być zmieniane,

  `closable` wartość `true`, jeśli ramka może być zamykana,

  `maximizable` wartość `true`, jeśli ramka może być maksymalizowana,

  `iconifiable` wartość `true`, jeśli ramka może być zwijana do ikony,

- boolean isResizable()
- boolean isClosable()
- boolean isMaximizable()
- boolean isIconifiable()

  Sprawdzają odpowiednie właściwości ramki. Jeśli właściwość posiada wartość `true`, oznacza to także obecność odpowiedniej ikony w pasku tytułu ramki.

- boolean isIcon()
- void setIcon(boolean b)
- boolean isMaximum()
- void setMaximum(boolean b)
- boolean isClosed()
- void setClosed(boolean b)

  Sprawdzają lub ustawiają właściwości ramki. Jeśli właściwość posiada wartość `true`, oznacza to, że ramka jest zwinięta do ikony, zmaksymalizowana bądź zamknięta.

- boolean isSelected()
- void setSelected(boolean b)

  Sprawdza lub ustawia właściwość wyboru ramki. Jeśli właściwość posiada wartość `true`, oznacza to, że ramka jest wybraną ramką pulpitu.

- void moveToFront()

- `void moveToBack()`

  Umieszcza ramkę na wierzchu lub spodzie pulpitu.

- `void reshape(int x, int y, int width, int height)` przesuwa ramkę i zmienia jej rozmiar.

  Parametry: `x, y` nowe współrzędne lewego górnego narożnika ramki,

  `width, height` szerokość i wysokość ramki.

- `Container getContentPane()`
- `void setContentPane(Container c)`

  Pobierają i zwracają panel ramki wewnętrznej.

- `JDesktopPane getDesktopPane()` pobiera pulpit dla danej ramki wewnętrznej.
- `Icon getFrameIcon()`
- `void setFrameIcon(Icon icon)`

  Pobierają i nadają ikonę ramki umieszczoną w jej pasku tytułowym.

- `boolean isVisible()`
- `void setVisible(boolean b)`

  Sprawdzają i ustawiają właściwość „widoczności" ramki.

- `void show()` sprawia, że ramka staje się widoczna i pojawia się na wierzchu pulpitu.

javax.swing.JComponent

- `void addVetoableChangeListener(VetoableChangeListener listener)` instaluje obiekt nasłuchujący zmiany, która może zostać zawetowana. Jest on zawiadamiany, gdy ma miejsce próba zmiany ograniczonej właściwości.

java.beans.VetoableChangeListener

- `void vetoableChange(PropertyChangeEvent event)` metoda wywoływana, gdy metoda `set` ograniczonej właściwości zawiadamia obiekt nasłuchujący zmiany, która może być zawetowana.

java.beans.PropertyChangeEvent

- `String getPropertyName()` zwraca nazwę zmienianej właściwości.
- `Object getNewValue()` zwraca proponowaną nową wartość właściwości.

java.beans.PropertyVetoException

- `PropertyVetoException(String reason, PropertyChangeEvent event)` tworzy wyjątek weta zmiany właściwości.

  Parametry: `reason` powód weta,

  `event` wetowane zdarzenie.