Praktyczny kurs Java. Wydanie II

Autor: Marcin Lis
ISBN: 978-83-246-0876-8
Format: B5, stron: 400

Opanuj język programowania, który zmienił oblicze sieci

- Jakie elementy tworzą język Java?
- Jak wykorzystać pełnię możliwości programowania obiektowego?
- Jak tworzyć własne aplety i aplikacje?

Interesuje Cię język programowania, który zyskuje coraz większą popularność wśród twórców rozwiązań korporacyjnych? A może zamierzasz tworzyć aplikacje dla urządzeń mobilnych? Najwyższa pora poznać tajniki Javy. Ten język już dawno przestał być narzędziem do tworzenia prostych programików osadzanych na stronach WWW. Współczesna Java to potężny obiektowy język programowania wykorzystywany w aplikacjach bankowych i finansowych, portalach internetowych i wielu innych systemach. Jedna z jego wersji służy także do pisania oprogramowania dla telefonów komórkowych, terminali BlackBerry i komputerów przenośnych. Warto więc poznać Javę.

„Praktyczny kurs Java. Wydanie II" to kolejna edycja podręcznika, dzięki któremu poznasz tajniki tego niezwykłego języka programowania. Znajdziesz tu omówienie elementów najnowszej wersji Javy, słów kluczowych tego języka, konstrukcji sterujących i zasad programowania. Dowiesz się, na czym polega projektowanie i programowanie obiektowe. Nauczysz się korzystać z mechanizmów obsługi wyjątków, implementować w programach operacje wejścia i wyjścia oraz budować własne aplikacje i aplety.

- Instalacja Javy w Windows i Linuksie
- Instrukcje Javy
- Operacje na tablicach
- Podstawy programowania obiektowego
- Obsługa wyjątków
- Zaawansowane zagadnienia programowania obiektowego
- Operacje wejścia i wyjścia
- Obsługa myszy i klawiatury
- Tworzenie interfejsów użytkownika
- Korzystanie z komponentów

Zostań profesjonalnym programistą Javy

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

Spis treści

Rozdział 8.

Aplikacje i aplety

Programy w Javie mogą być apletami (ang. *applet*) lub aplikacjami (ang. *application*). Różnica jest taka, że aplikacja jest programem samodzielnym uruchamianym z poziomu systemu operacyjnego (a dokładniej: maszyny wirtualnej Javy operującej na poziomie systemu operacyjnego) i tym samym ma pełny dostęp do zasobów udostępnianych przez system, natomiast aplet jest uruchamiany pod kontrolą innego programu, najczęściej przeglądarki internetowej, i ma dostęp jedynie do środowiska, które mu ten program udostępni. Inaczej aplet to program zagnieżdżony w innej aplikacji. Zazwyczaj nie ma on dostępu np. do dysku twardego (i innych urządzeń), tak aby ściągnięty nieświadomie z internetu nie mógł zaszkodzić użytkownikowi, kasując dane, chyba że zostanie wyposażony w specjalny podpis cyfrowy i użytkownik zgodzi się na udostępnienie mu chronionych zasobów. Osobną już sprawą są wykrywane co jakiś czas błędy w mechanizmach bezpieczeństwa przeglądarek, które niekiedy powodują, że złośliwie napisane aplety mogą obejść restrykcje i dostać się do chronionych zasobów systemu. Nie są to jednak sytuacje bardzo częste.

W rzeczywistości różnice w konstrukcji apletu i aplikacji nie są bardzo duże, jednak rozdział ten został podzielony na dwie sekcje. W pierwszej omówimy właśnie działające pod kontrolą przeglądarek aplety, a w drugiej samodzielne aplikacje.

Aplety

Lekcja 37. Podstawy apletów

Wszystkie przykłady programów prezentowane we wcześniejszych lekcjach pracowały w trybie tekstowym, najwyższy więc czas przejść w świat aplikacji pracujących w trybie graficznym. Lekcja 37. rozpoczyna omawianie apletów, czyli niewielkich aplikacji uruchamianych pod kontrolą przeglądarek internetowych. Zobaczymy, jak wygląda ogólna konstrukcja apletu, w jaki sposób jest on wywoływany przez przeglądarkę oraz jak przekazać mu parametry, które mogą sterować sposobem jego wykonania.

Pierwszy aplet

Utworzenie apletu wymaga użycia klasy `Applet` lub, jeśli miałyby się w nim znaleźć komponenty pakietu *Swing*, `JApplet`. Co prawda w pierwszych przykładach nie będziemy wykorzystywać tej biblioteki, jednak aplety będziemy wyprowadzać z nowocześniejszej klasy `JApplet`, komponenty *Swing* zostaną natomiast przedstawione w lekcji 42. Zanim jednak zaczniemy dokładne omawianie budowy tego typu programów, spróbujmy na rozgrzewkę napisać prosty aplet, który, jakżeby inaczej, będzie wyświetlał na ekranie dowolny napis. Jest on widoczny na listingu 8.1.

Listing 8.1.

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

public class PierwszyAplet extends JApplet {
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString ("Pierwszy aplet", 100, 50);
  }
}
```

Na początku importujemy klasę `JApplet` z pakietu `javax.swing` oraz pakiet `java.awt`. Pakiet `java.awt` jest potrzebny, gdyż zawiera definicję klasy `Graphics`, dzięki której można wykonywać operacje graficzne. Aby utworzyć własny aplet, trzeba wyprowadzić klasę pochodną od `JApplet`, nasza klasa nazywa się po prostu `PierwszyAplet`. Co dalej? Otóż `aplet` będzie wyświetlany w przeglądarce i zajmie pewną cześć jej okna. Chcemy na powierzchni apletu wyświetlić napis, musimy więc w jakiś sposób uzyskać do niej dostęp. Pomaga nam w tym metoda `paint`. Jest ona automatycznie wywoływana za każdym razem, kiedy zaistnieje konieczność odrysowania powierzchni apletu. Metoda ta otrzymuje w argumencie wskaźnik do specjalnego obiektu udostępniającego metody pozwalające na wykonywanie operacji na powierzchni apletu. Wywołujemy więc metodę `clearRect`, która wyczyści obszar okna apletu[1], a następnie `drawString` rysującą napis we wskazanych współrzędnych. W powyższym przypadku będzie to napis `Pierwszy aplet` we współrzędnych `x = 100` i `y = 50`. Metoda `clearRect` przyjmuje cztery argumenty określające prostokąt, który ma być wypełniony kolorem tła. Dwa pierwsze określają współrzędne lewego górnego rogu, a dwa kolejne — szerokość i wysokość. Szerokość uzyskiwana jest przez wywołanie `getSize().width`[2], a wysokość — `getSize().height`.

Do techniki rysowania na powierzchni apletu wrócimy już niebawem, teraz jednak zajmiemy się jego umieszczeniem w przeglądarce. Będzie to wymagać napisania fragmentu kodu HTML i umieszczenia w nim odpowiedniego znacznika. Znacznikiem historycznym służącym do umieszczania apletów był `<applet>`, o postaci:

[1] W praktyce przed wywołaniem `clearRect` należałoby użyć metody `setColor` (patrz lekcja „Czcionki i kolory") tak, aby na każdej platformie uruchomieniowej uzyskać taki sam kolor tła. W różnych systemach domyślny kolor tła może być bowiem inny.

[2] Jest to więc odwołanie do pola `width` obiektu zwróconego przez wywołanie metody `getSize`. Jest to obiekt klasy `Dimension`.

```
<applet
  code = "nazwa_klasy"
  width = "szerokość apletu"
  height = "wysokość apletu"
>
</applet>
```

W przypadku naszego pierwszego apletu znacznik ten przyjąłby postać widoczną na listingu 8.2.

Listing 8.2.

```
<applet
  code = "PierwszyAplet.class"
  width = "szerokość apletu"
  height = "100"
>
</applet>
```

Wciąż jest on rozpoznawany, choć nie należy do standardu HTML 4. Obecnie zamiast niego lepiej stosować znacznik <object>. Pakiet JDK zawiera jednak narzędzie do konwersji kodu HTML o nazwie *HtmlConverter*, które potrafi automatycznie wygenerować odpowiednią wersję kodu dla różnych przeglądarek. Przykładowo dla Internet Explorera kod przyjąłby postać widoczną na listingu 8.3.

Listing 8.3.

```
<object
    classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
    codebase = "http://java.sun.com/update/1.6.0/
    ↳jinstall-6-windows-i586.cab#Version=6,0,0,105"
    width = "300" height = "100" >
    <param name = code value = "PierwszyAplet.class" >
    <param name = "type" value = "application/x-java-applet;version=1.6">
    <param name = "scriptable" value = "false">
</object>
```

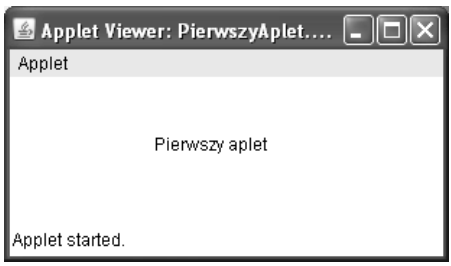
Tak skonstruowany plik z kodem HTML można już wczytać do przeglądarki. Pakiet JDK zawiera swoją własną przeglądarkę służącą do testowania apletów — *Applet Viewer*. Uruchamiamy ją, pisząc w wierszu poleceń:

```
appletviewer nazwa_pliku.html
```

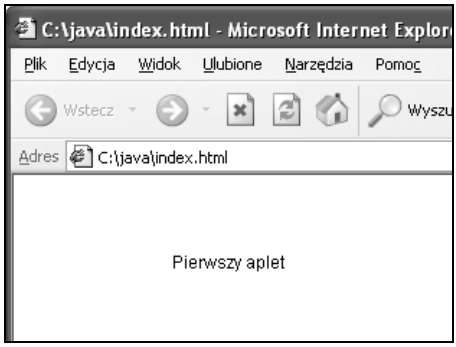
Wynik działania naszego apletu w tej przeglądarce jest widoczny na rysunku 8.1. Na rysunku 8.2 zaprezentowany został natomiast ten sam aplet po wczytaniu do Internet Explorera.

Konstrukcja apletu

Kiedy przeglądarka obsługująca Javę odnajdzie w kodzie HTML osadzony aplet, wykonuje określoną sekwencję czynności. Zaczyna oczywiście od sprawdzenia, czy w podanej lokalizacji znajduje się plik zawierający kod danej klasy. Jeśli tak, kod ten jest ładowany do



Rysunek 8.1. *Pierwszy aplet w przeglądarce Applet Viewer*



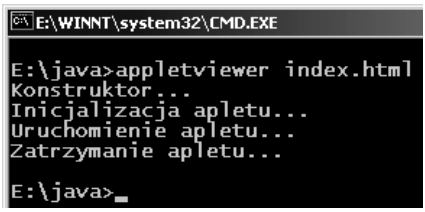
Rysunek 8.2. *Pierwszy aplet w przeglądarce Internet Explorer*

pamięci i tworzona jest instancja (czyli obiekt) znalezionej klasy, a tym samym wywoływany jest konstruktor. Następnie jest wykonywana metoda `init` utworzonego obiektu informująca go o tym, że został załadowany do systemu. W metodzie `init` można zatem umieścić, o ile zachodzi taka potrzeba, procedury inicjacyjne.

Kiedy aplet jest gotowy do uruchomienia, przeglądarka wywołuje jego metodę `start`, informując, że powinien rozpocząć swoje działanie. Przy pierwszym ładowaniu apletu metoda `start` jest zawsze wykonywana po metodzie `init`. W momencie, kiedy aplet powinien zakończyć swoje działanie, jest z kolei wywoływana jego metoda `stop`. O tym, że faktycznie występuje taka sekwencja instrukcji, możemy się przekonać, uruchamiając (czy też wczytując do przeglądarki) kod z listingu 8.4. Wynik działania dla *appletviewera* uruchamianego z konsoli systemowej jest widoczny na rysunku 8.3. Na rysunku 8.4 jest z kolei widoczny obraz konsoli Javy w przeglądarce korzystającej z wtyczki JRE 1.6.

Listing 8.4.

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

public class Aplet extends JApplet {
  public Aplet() {
    System.out.println("Konstruktor...");
  }
  public void init() {
    System.out.println("Inicjalizacja apletu...");
  }
  public void start() {
    System.out.println("Uruchomienie apletu...");
  }
  public void stop() {
```



Rysunek 8.3. *Kolejność wykonywania metod w aplecie na konsoli systemowej*

Rysunek 8.4. *Kolejność wykonywania metod w aplecie na konsoli Java Plugin*

```
    System.out.println("Zatrzymanie apletu...");
  }
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString ("Pierwszy aplet", 100, 50);
  }
}
```

Parametry apletu

W przypadku niektórych apletów istnieje potrzeba przekazania im parametrów z kodu HTML — na przykład wartości sterujących ich pracą. Taka możliwość oczywiście istnieje, znacznik `<applet>` (a także `<object>`) pozwala na zastosowanie argumentów o nazwie `param`. Ogólnie konstrukcja taka będzie miała postać:

```
<applet
  <!--parametry znacznika-->
>
<param name = "nazwa1" value = "wartość1">
...
<param name = "nazwan" value = "wartośćn">
</applet>
```

Znaczniki `<param>` umożliwiają właśnie przekazywanie różnych parametrów. Załóżmy dla przykładu, że chcielibyśmy przekazać apletowi tekst, który będzie następnie wyświetlany na jego powierzchni. Powinniśmy zatem w kodzie HTML zastosować następującą konstrukcję:

```
<applet
  <!--parametry znacznika-->
>
<poaram name="tekst" value="Testowanie parametrów apletu">
</applet>
```

Oznacza ona, że aplet będzie miał dostęp do parametru o nazwie `tekst` i wartości `Testowanie parametrów apletu`. Napiszmy teraz kod odpowiedniej klasy, która będzie potrafiła ten parametr odczytać. Jest on widoczny na listingu 8.5.

Listing 8.5.

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

public class Aplet extends JApplet {
  private String tekst;
  public void init() {
    if((tekst = getParameter("tekst")) == null)
      tekst = "Nie został podany parametr: tekst";
  }
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString (tekst, 60, 50);
  }
}
```

Do odczytu wartości parametru wykorzystaliśmy metodę `getParameter`, której jako argument należy przekazać nazwę parametru (w naszym przypadku `tekst`). Metoda ta zwróci wartość wskazanego argumentu lub wartość `null`, jeżeli parametr nie został uwzględniony w kodzie HTML.

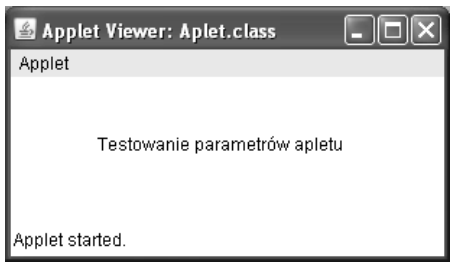
W klasie `Aplet` deklarujemy zatem pole typu `String` o nazwie `tekst`. W metodzie `init`, która zostanie wykonana po załadowaniu apletu przez przeglądarkę, odczytujemy wartość parametru i przypisujemy ją polu `tekst`. Sprawdzamy jednocześnie, czy pole to nie otrzymało wartości `null` — jeśli tak, przypisujemy mu własny tekst. Działanie metody `paint` jest analogiczne jak we wcześniejszych przykładach, wyświetla ona po prostu zawartość pola `tekst` na ekranie, w miejscu o wskazanych współrzędnych (`x = 60`, `y = 50`). Pozostaje nam więc napisanie kodu HTML zawierającego taki aplet, został on przedstawiony na listingu 8.6. Po jego uruchomieniu (nie zapomnijmy o wcześniejszej kompilacji klasy `Aplet`) zobaczymy widok zaprezentowany na rysunku 8.5.

Listing 8.6.

```
<applet
  code = "Aplet.class"
  width = "300"
  height = "100"
>
<param name="tekst" value="Testowanie parametrów apletu">
</applet>
```

Lekcja 38. Czcionki i kolory

Wiemy już, w jaki sposób wyświetlać w apletach napisy, ten temat był poruszany w pierwszej lekcji niniejszego rozdziału. W lekcji 38. zajmiemy się zatem sposobami umożliwiającymi wykorzystanie dostępnych w systemie czcionek. Poznamy bliżej klasę



Rysunek 8.5.
Napis przekazany apletowi w postaci parametru

`Font`. W drugiej części lekcji zajmiemy się natomiast kolorami, czyli klasą `Color` i takim jej wykorzystaniem, które umożliwi pokolorowanie wyświetlanych w apletach napisów.

Czcionki

Przykłady z poprzedniej lekcji pokazywały, w jaki sposób wyświetlić na powierzchni apletu napis. Bardzo przydałaby się nam w takim razie możliwość zmiany wielkości i, ewentualnie, kroju czcionki. Jest to jak najbardziej możliwe. W pakiecie `java.awt` znajduje się klasa `Font`, która umożliwia wykonywanie tego typu manipulacji. Musimy jednak wiedzieć, że czcionki w Javie dzielą się na dwa rodzaje: czcionki logiczne oraz czcionki fizyczne. W uproszczeniu możemy powiedzieć, że czcionki fizyczne są zależne od systemu, na którym działa maszyna wirtualna, natomiast czcionki logiczne to rodziny czcionek, które muszą być obsługiwane przez każde środowisko uruchomieniowe Javy (JRE). W środowisku Java 2 jest jedynie dostępnych pięć czcionek logicznych: Serif, SansSerif, Monospaced, Dialog, DialogInput.

Należy jednak pamiętać, że nie są to czcionki wbudowane, które będą tak samo wyglądały w każdym systemie. Zamiast tego jest wykorzystywana technika mapowania, tzn. dobierana jest ta czcionka systemowa (z systemu, na którym działa środowisko uruchomieniowe Javy), która najlepiej pasuje do jednej z wymienionych rodzin.

Aby w praktyce zobaczyć, jakie są różnice pomiędzy wymienionymi krojami, napiszemy teraz aplet, który wyświetli je na ekranie. Jest on widoczny na listingu 8.7.

Listing 8.7.

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

public class Aplet extends JApplet {
  Font serif;
  Font sansSerif;
  Font monospaced;
  Font dialog;
  Font dialogInput;
  public void init() {
    serif = new Font("Serif", Font.BOLD, 24);
    sansSerif = new Font("SansSerif", Font.BOLD, 24);
    monospaced = new Font("Monospaced", Font.BOLD, 24);
    dialog = new Font("Dialog", Font.BOLD, 24);
    dialogInput = new Font("DialogInput", Font.BOLD, 24);
  }
```

```
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.setFont(serif);
    gDC.drawString ("Czcionka Serif", 60, 40);

    gDC.setFont(sansSerif);
    gDC.drawString ("Czcionka SansSerif", 60, 80);

    gDC.setFont(monospaced);
    gDC.drawString ("Czcionka Monospaced", 60, 120);

    gDC.setFont(dialog);
    gDC.drawString ("Czcionka Dialog", 60, 160);

    gDC.setFont(dialogInput);
    gDC.drawString ("Czcionka DialogInput", 60, 200);
  }
}
```

W klasie `Aplet` przygotowujemy pięć pól klasy `Font` odpowiadających poszczególnym krojom pisma: `serif`, `sansSerif`, `monospaced`, `dialog` i `dialogInput`. W metodzie `init` tworzymy pięć obiektów klasy `Font` i przypisujemy je przygotowanym polom. Konstruktor klasy `Font` przyjmuje trzy parametry. Pierwszy z nich określa nazwę rodziny czcionek, drugi styl czcionki, natomiast trzeci jej wielkość. Styl czcionki ustalamy, posługując się stałymi (zmienne finalne) z klasy `Font`. Do dyspozycji mamy trzy różne wartości:

- `Font.BOLD` — czcionka pogrubiona,
- `Font.ITALIC` — czcionka pochylona,
- `Font.PLAIN` — czcionka zwyczajna.

Z instrukcji zawartych w metodzie `init` wynika zatem, że każdy krój będzie pogrubiony, o wielkości liter równej 24 punktom.

Wyświetlanie napisów odbywa się oczywiście w metodzie `paint`. Wykorzystujemy w tym celu znaną nam już metodę `drawString`, która wyświetla tekst w miejscu ekranu o współrzędnych wskazywanych przez drugi (współrzędna `x`) i trzeci (współrzędna `y`) argument. Do zmiany kroju wykorzystywanej czcionki stosujemy metodę o nazwie `setFont`. Przyjmuje ona jako argument obiekt klasy `Font` zawierający opis danej czcionki. Ostatecznie po skompilowaniu kodu i uruchomieniu apletu zobaczymy widok zaprezentowany na rysunku 8.6.

Nic nie stoi również na przeszkodzie, aby w konstruktorze klasy `Font` podać nazwę innej rodziny czcionek. Co się jednak stanie, jeśli dana czcionka nie istnieje w systemie? W takiej sytuacji zostanie wykorzystana czcionka domyślna, czyli kod będzie działał tak, jakby w konstruktorze został przekazany parametr `default`. Czcionką domyślną jest z kolei *Dialog*. Aby uniknąć takich niespodzianek, najlepiej po prostu pobrać listę wszystkich dostępnych fontów i tylko te wykorzystywać. Pomoże nam w tym klasa `GraphicsEnvironment`. Zawiera ona dwie interesujące nas metody: `getAllFonts` oraz `getAvailable-`

Rysunek 8.6. *Lista czcionek dostępnych standardowo*

`FontFamilyNames`[3]. Pierwsza z nich zwraca tablicę obiektów `Font` zawierającą wszystkie dostępne czcionki, natomiast druga tablicę obiektów klasy `String` z nazwami dostępnych rodzin czcionek. Na listingu 8.8 jest widoczny kod apletu, który wykorzystuje wymienione metody do wyświetlania na ekranie dostępnych w systemie czcionek.

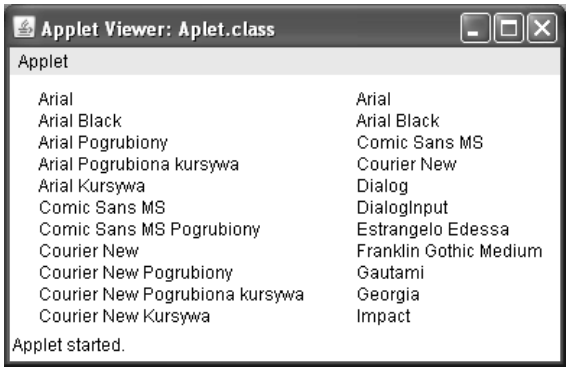
Listing 8.8.

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

public class Aplet extends JApplet {
  Font[] fonts;
  String[] fontNames;

  public void init() {
    GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
    fonts = ge.getAllFonts();
    fontNames = ge.getAvailableFontFamilyNames();
  }
  public void paint(Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    int y = 20;
    for(int i = 0; i < fonts.length; i++){
      gDC.drawString(fonts[i].getFontName(), 20, y);
      y += 15;
    }
    y = 20;
    for(int i = 0; i < fontNames.length; i++){
      gDC.drawString(fontNames[i], 240, y);
      y += 15;
    }
  }
}
```

[3] Metody te są dostępne w JDK w wersji 1.2 i wyższych.

W aplecie zostały zadeklarowane dwa pola typu tablicowego: `fonts` oraz `fontNames`. Pierwsze z nich będzie przechowywać tablicę obiektów klasy `Font`, drugie klasy `String`. W metodzie `init` musimy utworzyć obiekt klasy `GraphicsEnvironment`, który udostępnia potrzebne nam funkcje `getAllFonts` oraz `getAvailableFontFamilyNames`. Obiekt ten tworzymy, wywołując statyczną metodę `getLocalGraphicsEnvironment` klasy `GraphicsEnvironment`, i przypisujemy go zmiennej `ge`. Następnie pobieramy listę czcionek (i przypisujemy ją polu `fonts`) oraz listę nazw rodzin czcionek (i przypisujemy ją tablicy `fontNames`).

W metodzie `paint` pozostaje nam odczytać i wyświetlić na ekranie zawartość tablic. Służą do tego dwie pętle typu `for`. Do wyświetlania wykorzystujemy tradycyjnie metodę `drawString`. Ponieważ tablica `fonts` zawiera obiekt klasy `Font`, aby uzyskać nazwy czcionek, wywołujemy metodę `getFontName`. Tablica `fontNames` zawiera obiekty klasy `String`, możemy więc bezpośrednio użyć ich jako argumentu metody `drawString`. Do określenia współrzędnej `y` każdego napisu wykorzystujemy zmienną `y`, która po każdym wyświetleniu jest zwiększana o 15. Przykładowy wynik działania programu jest widoczny na rysunku 8.7. W pierwszej kolumnie są wyświetlane nazwy konkretnych czcionek, a w drugiej nazwy rodzin czcionek dostępnych w systemie, na którym aplet został uruchomiony.



Rysunek 8.7. *Lista dostępnych w systemie czcionek i rodzin czcionek*

Powróćmy teraz do przykładu, który wyświetlał na ekranie prosty napis. Zauważmy, że występuje w nim problem pozycjonowania tekstu na ekranie. Centrowanie odbywać się musi metodą prób i błędów, co z pewnością nie jest wygodne. Automatyczne wykonanie tego zadania wymaga jednak znajomości długości oraz wysokości napisu. Wiadomo też, że dla każdego rodzaju czcionki wartości te będą różne. Musimy w związku z tym posłużyć się metodami udostępnianymi przez klasę `FontMetrics`: `stringWidth` i `getHeight`. Pierwsza z nich podaje długość napisu w pikselach, natomiast druga jego przeciętną wysokość przy zastosowaniu danej czcionki. Możemy to wykorzystać do ustawienia tekstu na środku powierzchni apletu. Kod realizujący to zadanie jest widoczny na listingu 8.9.

Listing 8.9.

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

public class Aplet extends JApplet {
  String tekst = "Java";
```

```
  Font font = null;

  public void init() {
    font = new Font("Arial", Font.BOLD, 20);
  }
  public void paint (Graphics gDC) {
    gDC.setFont(font);
    FontMetrics fm = gDC.getFontMetrics();

    int strWidth = fm.stringWidth(tekst);
    int strHeight = fm.getHeight();

    int x = (getWidth() - strWidth) / 2;
    int y = (getHeight() + strHeight) / 2;
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString(tekst, x, y);
  }
}
```

W metodzie `init` tworzymy nowy obiekt klasy `Font` reprezentujący pogrubioną czcionkę Arial o wielkości 20 punktów i przypisujemy go polu `font` klasy `Aplet`. W metodzie `paint` wywołujemy metodę `setFont` obiektu `gDC`, przekazując jej jako argument obiekt `font`. Tym samym ustalamy krój czcionki, z jakiej chcemy korzystać przy wyświetlaniu napisów. Następnie pobieramy obiekt klasy `FontMetrics`, wywołując metodę `getFontMetrics`, i przypisujemy go zmiennej `fm`. Ten obiekt udostępnia nam metody niezbędne do określania wysokości i szerokości napisu. Szerokość uzyskujemy przez wywołanie metody `stringWidth`, a wysokość przez wywołanie metody `getHeight`. Kiedy pobierzemy te wartości, pozostaje nam proste wyliczenie współrzędnych, od których powinno rozpocząć się wyświetlanie napisu. Korzystamy ze wzorów:

```
x = (szerokość apletu - szerokość napisu) / 2
y = (wysokość apletu + szerokość napisu) / 2
```

Po wykonaniu obliczeń możemy już wywołać metodę `drawString`, która spowoduje, że napis zapisany w polu `tekst` pojawi się na ekranie.

Kolory

Aby zmienić kolor, którym obiekty są domyślnie rysowane na powierzchni apletu (jak przekonaliśmy się we wcześniejszych przykładach, jest to kolor czarny), trzeba skorzystać z metody `setColor` klasy `Graphics`. Jako argument przyjmuje ona obiekt klasy `Color` określający kolor, który będzie używany w dalszych operacjach. W klasie tej zostały zdefiniowane dwadzieścia cztery statyczne i finalne pola określające łącznie dwanaście różnych kolorów. Pola te zostały zebrane w tabeli 8.1. Gdybyśmy zatem chcieli zmienić kolor wyświetlanego tekstu w aplecie z listingu 8.9 na czerwony, należałoby przed wywołaniem `drawString` dodać do metody `paint` wywołanie metody `setColor(Color.red)` lub `setColor(Color.RED)`. Metoda `paint` miałaby wtedy następującą postać:

```
public void paint (Graphics gDC) {
  gDC.setFont(font);
  FontMetrics fm = gDC.getFontMetrics();

  int strWidth = fm.stringWidth(tekst);
```

```
    int strHeight = fm.getHeight();

    int x = (getWidth() - strWidth) / 2;
    int y = (getHeight() + strHeight) / 2;

    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.setColor(Color.RED);
    gDC.drawString(tekst, x, y);
  }
```

Tabela 8.1. *Stałe określające kolory w klasie Color*

Nazwa stałej	Reprezentowany kolor
`Color.black`	Czarny
`Color.BLACK`	Czarny
`Color.blue`	Niebieski
`Color.BLUE`	Niebieski
`Color.darkGray`	Ciemnoszary
`Color.DARK_GRAY`	Ciemnoszary
`Color.gray`	Szary
`Color.GRAY`	Szary
`Color.green`	Zielony
`Color.GREEN`	Zielony
`Color.lightGray`	Jasnoszary
`Color.LIGHT_GRAY`	Jasnoszary
`Color.magenta`	Karmazynowy
`Color.MAGENTA`	Karmazynowy
`Color.orange`	Pomarańczowy
`Color.ORANGE`	Pomarańczowy
`Color.pink`	Różowy
`Color.PINK`	Różowy
`Color.red`	Czerwony
`Color.RED`	Czerwony
`Color.white`	Biały
`Color.WHITE`	Biały
`Color.yellow`	Żółty
`Color.YELLOW`	Żółty

Szybko przekonamy się, że dwanaście kolorów to jednak trochę za mało — niestety, nie ma więcej stałych w klasie `Color`. Jeśli więc w tabeli 8.1 nie odnajdziemy odpowiadającego nam koloru, będziemy musieli postąpić inaczej. Otóż należy w takim przypadku utworzyć nowy obiekt klasy `Color`. Do dyspozycji mamy siedem konstruktorów, są one dokładniej opisane w dokumentacji JDK. Najwygodniej i najprościej będzie zastosować jeden z dwóch:

```
Color(int rgb)
Color(int r, int g, int b)
```

Pierwszy z nich przyjmuje jeden argument, którym jest wartość typu `int`. Musi być ona skonstruowana w taki sposób, aby bity 0 – 7 określały wartość składowej *R* (w modelu RGB), bity 8 – 15 wartość składowej *G*, a 16 – 23 składowej *B*. Drugi z wymienionych konstruktorów przyjmuje natomiast wartość składowych w postaci trzech argumentów typu `int`. Warto może w tym miejscu przypomnieć, że w modelu RGB kolor jest określany trzema składowymi: czerwoną (*R*, ang. *red*), zieloną (*G*, ang. *green*) i niebieską (*B*, ang. *blue*). Każdy z parametrów przekazanych drugiemu konstruktorowi może przyjmować wartości z zakresu 0 – 255 (jest to więc 8-bitowy model RGB), łącznie możemy zatem przedstawić 16 777 216 różnych kolorów. W tabeli 8.2 zostały przedstawione składowe RGB dla kilkunastu przykładowych kolorów.

Tabela 8.2. *Składowe RGB dla wybranych kolorów*

Kolor	Składowa R	Składowa G	Składowa B
Beżowy	245	245	220
Biały	255	255	255
Błękitny	173	216	230
Brązowy	165	42	42
Czarny	0	0	0
Czerwony	255	0	0
Ciemnoczerwony	139	0	00
Ciemnoniebieski	0	0	139
Ciemnoszary	169	169	169
Ciemnozielony	0	100	0
Fiolet	238	130	238
Koralowy	255	127	80
Niebieski	0	0	255
Oliwkowy	128	128	0
Purpurowy	128	0	128
Srebrny	192	192	192
Stalowoniebieski	70	130	180
Szary	128	128	128
Zielony	0	255	0
Żółtozielony	154	205	50
Żółty	255	255	0

Napiszmy zatem aplet wyświetlający w centrum powierzchni swojego okna napis, którego treść oraz kolor będą zdefiniowane poprzez zewnętrzne parametry określone w kodzie HTML. Kolor będzie określany trzema parametrami odzwierciedlającymi poszczególne składowe RGB. Kod klasy realizującej takie zadanie został przedstawiony na listingu 8.10.

Listing 8.10.

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

public class Aplet extends JApplet {
  String tekst;
  Font font = null;
  Color color = null;

  public void init() {
    int paramR = 0, paramG = 0, paramB = 0;
    try{
      paramR = Integer.parseInt(getParameter("paramR"));
      paramG = Integer.parseInt(getParameter("paramG"));
      paramB = Integer.parseInt(getParameter("paramB"));
    }
    catch(NumberFormatException e){
      paramR = 0; paramG = 0; paramB = 0;
    }
    color = new Color(paramR, paramG, paramB);

    font = new Font("Arial", Font.BOLD, 20);
    tekst = getParameter("tekst");
    if(tekst == null)
      tekst = "";
  }
  public void paint (Graphics gDC) {
    gDC.setFont(font);
    FontMetrics fm = gDC.getFontMetrics();

    int strWidth = fm.stringWidth(tekst);
    int strHeight = fm.getHeight();

    int x = (getWidth() - strWidth) / 2;
    int y = (getHeight() + strHeight) / 2;

    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.setColor(color);
    gDC.drawString(tekst, x, y);
  }
}
```

W klasie `Aplet` deklarujemy trzy pola: `tekst`, które będzie zawierało wyświetlany napis, `font`, zawierające obiekt klasy `Font` opisujący czcionkę, oraz `color`, opisujące kolor napisu. W metodzie `init` odczytujemy wartości przekazanych parametrów o nazwach `paramR`, `paramG` i `paramB`. Ponieważ metoda `getParameter` zwraca wartość typu `String`, próbujemy dokonać konwersji na typ `int` za pomocą metody `parseInt` klasy `Integer`. Gdyby którakolwiek z konwersji się nie powiodła, zostanie wygenerowany wyjątek `NumberFormatException`, który przechwytujemy w bloku `try`. Jeśli taki wyjątek nastąpi, przywracamy początkowe wartości zmiennym `paramR`, `paramG` i `paramB`. Tworzymy następnie nowy obiekt klasy `Color` i nowy obiekt klasy `Font`. Odczytujemy także wartość parametru `tekst` i przypisujemy polu o takiej samej nazwie. Gdyby okazało się, że zostanie zwrócona wartość pusta `null`, czyli parametr o nazwie `tekst` nie został przekazany,

polu `tekst` przypisujemy pusty ciąg znaków. Wyświetlaniem napisu w centrum okna zajmuje się metoda `paint`, której zasada działania jest identyczna jak w przypadku tej przedstawionej na początku tego podrozdziału. Na listingu 8.11 został przedstawiony przykładowy kod HTML zawierający wywołanie apletu.

Listing 8.11.

```
<applet
  code   = "Aplet.class"
  width  = "320"
  height = "200"
>
<param name="paramr" value="168">
<param name="paramg" value="18">
<param name="paramb" value="240">
<param name="tekst" value="Przykładowy tekst">
</applet>
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 38.1.

Napisz program, który w trybie tekstowym wyświetli na konsoli systemowej listę czcionek dostępnych w systemie.

Ćwiczenie 38.2.

Napisz aplet wyświetlający na środku powierzchni okna tekst, którego treść oraz wielkość i styl czcionki będą przekazywane w postaci parametrów zawartych w kodzie HTML.

Ćwiczenie 38.3.

Napisz aplet, który będzie wyświetlał tekst o kolorze określonym przez parametr przekazywany z kodu HTML. Parametrem tym powinna być jedna liczba typu `int`.

Lekcja 39. Grafika

Aplety pracują w środowisku graficznym, czas więc poznać podstawowe operacje graficzne, jakie możemy wykonywać. Służą do tego metody klasy `Graphics`, niektóre z nich, jak `drawString` czy `setColor`, wykorzystywaliśmy już w poprzednich lekcjach. W tej lekcji dowiemy się, jak rysować proste figury geometryczne, takie jak linie, wielokąty czy okręgi, oraz jak wyświetlać obrazy z plików graficznych. Poznamy też interfejs `ImageObserver`, który pozwala na kontrolowanie postępów w ładowaniu plików graficznych do apletu.

Rysowanie figur geometrycznych

Rysowanie figur geometrycznych umożliwiają wybrane metody klasy `Graphics`. Rysować można zarówno same kontury, jak i pełne figury wypełnione zadanym kolorem. Do dyspozycji mamy zestaw funkcji pozwalających na tworzenie: linii, kół, elips i okręgów oraz wielokątów. Metody te zostały zebrane w tabeli 8.3.

Tabela 8.3. *Wybrane metody klasy Graphics*

Deklaracja metody	Opis
`void drawLine(int x1, int y1, int x2, int y2)`	Rysuje linię rozpoczynającą się w punkcie o współrzędnych `x1`, `y1` i kończącą się w punkcie `x2`, `y2`.
`void drawOval(int x, int y, int width, int height)`	Rysuje owal wpisany w prostokąt opisany parametrami `x`, `y` oraz `width` i `height`.
`void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Rysuje wielokąt o punktach wskazywanych przez tablice `xPoints` i `yPoints`. Liczbę segmentów linii, z których składa się figura, wskazuje parametr `nPoints`.
`void drawPolygon(Polygon p)`	Rysuje wielokąt opisany przez argument `p`.
`void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)`	Rysuje sekwencję połączonych ze sobą linii o współrzędnych zapisanych w tablicach `xPoints` i `yPoints`. Liczba segmentów jest określona przez argument `nPoint`.
`void drawRect(int x, int y, int width, int height)`	Rysuje prostokąt zaczynający się w punkcie o współrzędnych `x`, `y` oraz szerokości i wysokości określonej przez argumenty `width` i `height`.
`void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	Rysuje prostokąt o zaokrąglonych rogach zaczynający się w punkcie o współrzędnych `x`, `y` oraz szerokości i wysokości określonej przez argumenty `width` i `height`. Stopień zaokrąglenia rogów jest określany przez argumenty `arcWidth` i `arcHeight`.
`void fillOval(int x, int y, int width, int height)`	Rysuje koło lub elipsę wpisaną w prostokąt opisany parametrami `x`, `y` oraz `width` i `height`.
`void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Rysuje wypełniony bieżącym kolorem wielokąt o punktach wskazywanych przez tablice `xPoints` i `yPoints`. Liczbę segmentów linii, z których składa się figura, wskazuje argument `nPoints`.
`void fillPolygon(Polygon p)`	Rysuje wypełniony bieżącym kolorem wielokąt opisany przez argument `p`.
`void fillRect(int x, int y, int width, int height)`	Rysuje wypełniony bieżącym kolorem prostokąt zaczynający się w punkcie o współrzędnych `x`, `y` oraz szerokości i wysokości określonej przez argumenty `width` i `height`.
`void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	Rysuje wypełniony bieżącym kolorem prostokąt o zaokrąglonych rogach, zaczynający się w punkcie o współrzędnych `x`, `y` oraz szerokości i wysokości określonej przez argumenty `width` i `height`. Stopień zaokrąglenia rogów jest określany przez argumenty `arcWidth` i `arcHeight`.

Punkty i linie

Klasa `Graphics` nie oferuje oddzielnej metody, która umożliwiałaby rysowanie pojedynczych punktów, zamiast tego możemy jednak skorzystać z metody rysującej linie. Wystarczy, jeśli jako punkt początkowy i docelowy będą miały te same współrzędne. Tak więc aby narysować punkt o współrzędnych `x = 100` i `y = 200`, można zastosować następującą instrukcję:

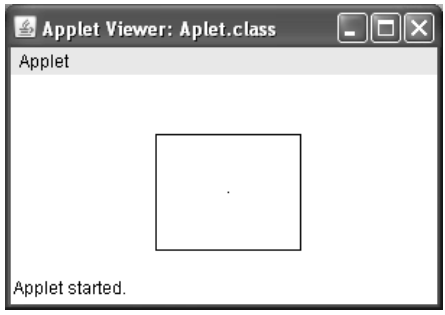
```
drawLine(100, 200, 100, 200);
```

Prosty aplet rysujący (składający się z czterech oddzielnych linii) prostokąt z pojedynczym punktem w środku jest widoczny na listingu 8.12, a efekt jego działania na rysunku 8.8.

Listing 8.12.

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

public class Aplet extends JApplet {
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawLine(100, 40, 200, 40);
    gDC.drawLine(100, 120, 200, 120);
    gDC.drawLine(100, 40, 100, 120);
    gDC.drawLine(200, 40, 200, 120);
    gDC.drawLine(150, 80, 150, 80);
  }
}
```

Rysunek 8.8.
Efekt działania apletu z listingu 8.12



Jeśli chcemy rysować punkty lub linie w różnych kolorach, musimy skorzystać z metody `setColor`, analogicznie jak w przykładach z lekcji 38. Należy zatem utworzyć nowy obiekt klasy `Color` opisujący dany kolor i użyć go jako argumentu tej metody. Po jej wybraniu wszystkie obiekty będą rysowane w wybranym kolorze.

Koła i elipsy

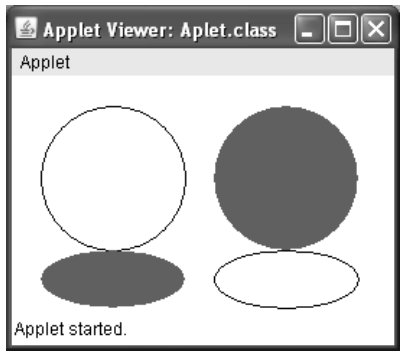
Do tworzenia kół i elips służą dwie metody klasy `Graphics`: `drawOval` oraz `fillOval`. Pierwsza rysuje sam kontur figury, druga figurę wypełnioną aktualnym kolorem. Kolor oczywiście można zmieniać przy użyciu metody `setColor`. Czemu do tworzenia koła

i elipsy została stworzona tylko jedna metoda? Ponieważ koło jest po prostu szczególnym przypadkiem elipsy, w której oba promienie są sobie równe. Nie będziemy tu zagłębiać się w niuanse matematyki, musimy tylko wiedzieć, w jaki sposób określamy rozmiary figur w wymienionych metodach. Przyjmują one bowiem cztery parametry opisujące prostokąt, w który można wpisać koło lub elipsę. Ponieważ w prostokąt o określonych rozmiarach można wpisać tylko jeden kształt owalny, zatem określenie prostokąta jednoznacznie wyznacza koło lub elipsę.

Obie metody przyjmują cztery argumenty. Pierwsze dwa określają współrzędne `x` i `y` lewego górnego rogu prostokąta, natomiast dwa kolejne jego szerokość oraz wysokość. Przykład wykorzystujący metody `drawOval` i `fillOval` do narysowania na powierzchni apletu kół i elips o różnych kolorach został zaprezentowany na listingu 8.13, a efekt jego działania jest widoczny na rysunku 8.9.

Listing 8.13.

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

public class Aplet extends JApplet {
  public void paint (Graphics gDC){
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawOval (20, 20, 100, 100);
    gDC.drawOval (140, 120, 100, 40);
    gDC.setColor(Color.RED);
    gDC.fillOval (20, 120, 100, 40);
    gDC.fillOval (140, 20, 100, 100);
  }
}
```



Rysunek 8.9. *Efekt działania metod fillOval i drawOval klasy Graphics*

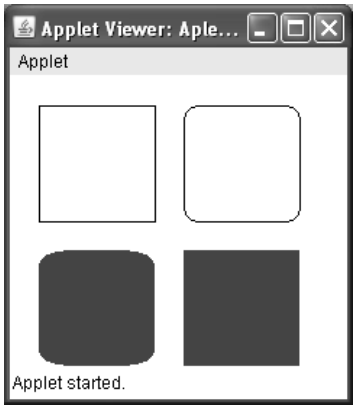
Wielokąty

Do rysowania wielokątów możemy wykorzystać kilka różnych metod — możemy rysować zarówno same kontury (metody zaczynające się słowem `draw`), jak i figury wypełnione kolorem (metody zaczynające się słowem `fill`). Do rysowania prostokątów służą metody `drawRectangle` oraz `fillRectangle`. Przyjmują one cztery argumenty — dwa pierwsze określają współrzędne lewego górnego rogu, natomiast dwa kolejne szerokość oraz wysokość figury. Istnieje również możliwość narysowania prostokąta

o zaokrąglonych rogach[4]. Służą do tego metody `drawRoundRect` oraz `fillRoundRect`. W takim przypadku do wymienionych przed chwilą argumentów dochodzą dwa dodatkowe określające średnicę łuku zaokrąglenia w poziomie oraz w pionie (deklaracje wszystkich wymienionych metod znajdują się w tabeli 8.3). Przykładowy aplet rysujący wymienione figury jest widoczny na listingu 8.14, a efekt jego działania na rysunku 8.10.

Listing 8.14.

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

public class Aplet extends JApplet {
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawRect(20, 20, 80, 80);
    gDC.drawRoundRect(120, 20, 80, 80, 20, 20);
    gDC.setColor(Color.BLUE);
    gDC.fillRoundRect(20, 120, 80, 80, 40, 20);
    gDC.fillRect(120, 120, 80, 80);
  }
}
```



Rysunek 8.10. *Efekt działania metod rysujących różne rodzaje prostokątów*

Oprócz prostokątów możemy rysować praktycznie dowolne wielokąty określone zbiorem punktów wskazujących kolejne wierzchołki. Podobnie jak w przypadku wcześniej omawianych kształtów mogą to być zarówno jedynie kontury, jak i figury wypełnione kolorem. Punkty określające wierzchołki przekazujemy w postaci dwóch tablic, pierwsza zawiera współrzędne `x`, druga współrzędne `y`. Jeśli więc chcemy narysować sam kontur, wykorzystujemy metodę:

```
drawPolygon(int[] x, int[] y, int ile)
```

jeśli natomiast ma to być pełna figura, metodę:

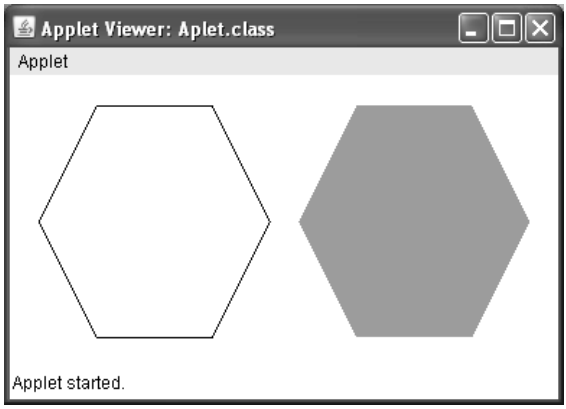
```
fillPolygon(int[] x, int[] y, int ile)
```

[4] Oczywiście, formalnie taka figura nie jest już wielokątem.

Parametr `ile` określa liczbę wierzchołków. Jeżeli ostatni punkt nie będzie się pokrywał z pierwszym, figura zostanie automatycznie domknięta. Na listingu 8.15 jest widoczny aplet rysujący dwa sześciokąty, wykorzystujący wymienione metody. Efekt jego działania prezentuje rysunek 8.11.

Listing 8.15.

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

public class Aplet extends JApplet {
  int tabX1[] = {20, 60, 140, 180, 140, 60};
  int tabY1[] = {100, 20, 20, 100, 180, 180};

  int tabX2[] = {200, 240, 320, 360, 320, 240};
  int tabY2[] = {100, 20, 20, 100, 180, 180};

  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawPolygon(tabX1, tabY1, 6);
    gDC.setColor(Color.GREEN);
    gDC.fillPolygon(tabX2, tabY2, 6);
  }
}
```



Rysunek 8.11. *Wykorzystanie metod drawPolygon i fillPolygon do wyświetlenia sześciokątów*

Istnieją również przeciążone wersje metod `drawPolygon` i `fillPolygon`, które przyjmują jako argument obiekt klasy `Polygon`. Ich deklaracje mają postać:

```
drawPolygon(Polygon p)
fillPolygon(Polygon p)
```

Aby z nich skorzystać, należy oczywiście najpierw utworzyć obiekt klasy `Polygon`. Ma ona dwa konstruktory, jeden bezargumentowy i drugi w postaci:

```
Polygon(int[] x, int[] y, int ile)
```

Pierwsze dwa argumenty to tablice współrzędnych kolejnych punktów, natomiast argument trzeci to liczba punktów. Przykładowy sześciokąt można zatem utworzyć w sposób następujący:

```
int tabX[] = {20, 60, 140, 180, 140, 60};
int tabY[] = {100, 20, 20, 100, 180, 180};
Polygon polygon = new Polygon(tabX, tabY, 6);
```

Wczytywanie grafiki

Wczytywanie obrazów graficznych umożliwia metoda klasy `Applet` o nazwie `getImage`. Wczytuje ona wskazany w argumencie plik graficzny w formacie *GIF*, *JPG* lub *PNG* i zwraca obiekt klasy `Image`, który może zostać wyświetlony na ekranie. Metoda `getImage` występuje w dwóch następujących wersjach:

```
getImage(URL url)
getImage(URL url, String name)
```

Pierwsza z nich przyjmuje jako argument obiekt klasy `URL` bezpośrednio wskazujący na plik z obrazem, druga wymaga podania argumentu klasy `URL` wskazującego na umiejscowienie pliku (np. *http://host.domena/java/obrazy/*) i drugiego, określającego nazwę pliku. Jeśli obraz znajduje się w strukturze katalogów naszego serwera, wygodne będzie użycie drugiej postaci konstruktora. Nie tworzymy wtedy nowego obiektu klasy `URL` bezpośrednio, ale wywołując jedną z dwóch metod:

```
getDocumentBase()
```

lub:

```
getCodeBase()
```

Metoda `getDocumentBase` zwraca obiekt `URL` wskazujący lokalizację dokumentu HTML, w którym znajduje się aplet, natomiast `getCodeBase` lokalizację, w której znajduje się kod apletu.

Kiedy otrzymamy zwrócony przez `getImage` obiekt klasy `Image`, można będzie go użyć jako argumentu metody `drawImage` rysującej obraz na powierzchni apletu. Metoda `drawImage` występuje w kilku przeciążonych wersjach (ich opis można znaleźć w dokumentacji JDK), w najprostszej postaci powinniśmy zastosować wywołanie:

```
gDC.drawImage (img, wspX, wspY, this);
```

Argument `img` to referencja do obiektu klasy `Image`, `wspX` to współrzędna `x`, a `wspY` — współrzędna `y`, `this` to referencja do obiektu implementującego interfejs `imageObserver` (w naszym przypadku będzie to obiekt klasy `Aplet` – za chwilę wyjaśnimy to dokładniej). Przykładowy aplet wyświetlający opisanym sposobem plik graficzny jest widoczny na listingu 8.16, natomiast efekt jego działania prezentuje rysunek 8.12.

Listing 8.16.

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

public class Aplet extends JApplet {
  Image img;
  public void init() {
    img = getImage(getDocumentBase(), "image.jpg");
  }
```

Rysunek 8.12. *Obraz wczytany za pomocą metody getImage*

```
    public void paint(Graphics gDC) {
      gDC.clearRect(0, 0, getSize().width, getSize().height);
      gDC.drawImage (img, 20, 20, this);
    }
}
```

Wyjaśnijmy teraz, do czego służy czwarty parametr metody `drawImage`, którym w przypadku apletu z listingu 8.16 było wskazanie na obiekt tego apletu (wskazanie `this`). Przede wszystkim musimy wiedzieć, co się dzieje w kolejnych fazach pracy takiego apletu. W metodzie `init` wywołujemy metodę `getImage`, przekazując jej w argumencie lokalizację pliku. Ta metoda zwraca obiekt klasy `Image` niezależnie od tego, czy wskazany plik graficzny faktycznie istnieje, czy nie. Ładowanie danych rozpocznie się dopiero w momencie pierwszego wywołania metody `drawImage`.

Sama metoda `drawImage` działa natomiast w taki sposób, że po jej wywołaniu jest wyświetlana dostępna część obrazu (czyli albo nic, albo część obrazu, albo cały obraz) i metoda kończy działanie. Jeśli cały obraz był dostępny, jest zwracana wartość `true`, jeśli nie — wartość `false`. Jeśli obraz nie był w pełni dostępny i zwrócona została wartość `false`, jest on ładowany w tle. W trakcie tego ładowania, czyli napływania kolejnych danych z sieci, jest wywoływana metoda `imageUpdate` obiektu implementującego interfejs `ImageObserver`, który został przekazany jako czwarty argument metody `drawImage`.

Ponieważ klasa `JApplet` implementuje ten interfejs, możemy jej obiekt wykorzystać jako czwarty argument metody. Osiągamy wtedy sytuację, kiedy obiekt apletu jest informowany o postępach ładowania obrazu. Gdybyśmy więc chcieli mieć możliwość kontroli procesu wczytywania i wyświetlania obrazu, należy przeciążyć metodę `imageUpdate` klasy `JApplet`. Przykładowo: jeśli w trakcie ładowania obrazu na pasku stanu miałaby być wyświetlana informacja o tym procesie, należałoby zastosować kod widoczny na listingu 8.17.

Listing 8.17.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.image.*;

public class Aplet extends JApplet {
  Image img;
  public void init() {
    img = getImage(getDocumentBase(), "image.jpg");
  }
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawImage (img, 20, 20, this);
  }
  public boolean imageUpdate(Image img, int flags, int x, int y, int width,
  ↳int height) {
    if ((flags & ImageObserver.ALLBITS) == 0){
      showStatus ("Ładowanie obrazu...");
      return true;
    }
    else{
      showStatus ("Obraz załadowany");
      repaint();
      return false;
    }
  }
}
```

Metoda `imageUpdate` przy każdym wywołaniu otrzymuje cały zestaw argumentów, czyli: `img` — referencję do rysowanego obrazu (jest to ważne, gdyż jednocześnie może być przecież ładowanych kilka obrazów), `x` i `y` — współrzędne powierzchni apletu, w których rozpoczyna się obraz, `width` i `height` — wysokość i szerokość obrazu oraz najważniejszy dla nas w tej chwili — `flags`. Argument `flags` to wartość typu `int`, w której poszczególne bity informują o stanie ładowanego obrazu. Ich dokładne znaczenie można znaleźć w dokumentacji JDK w opisie interfejsu `ImageObserver`.

Najważniejszy dla nas jest bit piąty (o nazwie `ALLBITS`), którego ustawienie na 1 oznacza, że obraz został całkowicie załadowany i może zostać wyświetlony w ostatecznej formie. Do zbadania stanu tego bitu wykorzystujemy zdefiniowaną w klasie `ImageObserver` stałą (statyczna i finalne pole typu `int`) `ImageObserver.ALLBITS`. Jeśli wynikiem operacji bitowej `AND` między argumentem `flags` oraz stałą `ALLBITS` jest 0, oznacza to, że bit ten jest wyłączony, a zatem obraz nie jest załadowany, jeśli natomiast wynik tej operacji jest różny od 0, oznacza to, że bit jest włączony i dysponujemy pełnym obrazem.

Te właściwości wykorzystujemy zatem w metodzie `imageUpdate`. Badamy wynik iloczynu logicznego `flags & ImageObserver.ALLBITS`. Kiedy jest on równy `0`, wykonujemy metodę `showStatus` ustawiającą tekst na pasku stanu przeglądarki (rysunek 8.13) informujący, że obraz jest w trakcie ładowania; kiedy jest natomiast różny od 0, wyświetlamy napis, iż obraz został załadowany. W tym drugim przypadku należy dodatkowo odświeżyć ekran apletu, za co odpowiada metoda `repaint`.

Rysunek 8.13. *Wyświetlanie informacji o stanie załadowania obrazu*

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 39.1.

Napisz aplet rysujący owal, którego rozmiar i położenie będą przekazywane w postaci argumentów z kodu HTML.

Ćwiczenie 39.2.

Zmodyfikuj kod z listingu 8.15 tak, aby wykorzystywał metody rysujące wielokąty, które przyjmują jako argumenty obiekty klasy `Polygon`.

Ćwiczenie 39.3.

Zmień kod apletu z listingu 8.17 tak, aby informacja o stanie ładowania obrazu była wyświetlana nie w wierszu statusu przeglądarki, ale na środku jego powierzchni.

Lekcja 40. Dźwięki i obsługa myszy

Niektóre aplety wymagają reakcji na działania użytkownika, zwykle chodzi o zdarzenia związane z obsługą myszy. Jeśli aplet ma reagować na zmiany położenia kursora czy kliknięcia, może bezpośrednio implementować odpowiedni interfejs, bądź też korzystać z dodatkowego obiektu. Jak taka obsługa wygląda w praktyce, zobaczymy właśnie w trakcie lekcji 40. W drugiej części lekcji zajmiemy się odtwarzaniem dźwięków przez aplety i poznamy interfejs `AudioClip`.

Interfejs MouseListener

Interfejs `MouseListener` jest zdefiniowany w pakiecie `java.awt.event`, a zatem klasa, która będzie go implementowała, musi zawierać odpowiednią dyrektywę `import`. Jest on dostępny we wszystkich JDK, począwszy od wersji 1.1. Znajdują się w nim deklaracje pięciu metod zebranych w tabeli 8.4.

Każda klasa implementująca ten interfejs musi zatem zawierać definicję wszystkich wymienionych metod, nawet jeśli nie będzie ich wykorzystywała. Szkielet apletu będzie więc miał postać widoczną na listingu 8.18. Jak widzimy, mamy możliwość reagowania na pięć różnych zdarzeń opisanych w tabeli 8.4 i w komentarzach w zaprezentowanym kodzie.

Tabela 8.4. *Metody interfejsu MouseListener*

Deklaracja metody	Opis	Od JDK
`void mouseClicked(MouseEvent e)`	Metoda wywoływana po kliknięciu przyciskiem myszy.	1.1
`void mouseEntered(MouseEvent e)`	Metoda wywoływana, kiedy kursor myszy wejdzie w obszar komponentu.	1.1
`void mouseExited(MouseEvent e)`	Metoda wywoływana, kiedy kursor myszy opuści obszar komponentu.	1.1
`void mousePressed(MouseEvent e)`	Metoda wywoływana po naciśnięciu przycisku myszy.	1.1
`void mouseReleased(MouseEvent e)`	Metoda wywoływana po puszczeniu przycisku myszy.	1.1

Listing 8.18.

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

public class Aplet extends JApplet implements MouseListener {
  public void mouseClicked(MouseEvent e) {
    //kod wykonywany po kliknięciu myszą
  }
  public void mouseEntered(MouseEvent e) {
    //kod wykonywany, kiedy kursor wejdzie w obszar komponentu
  }
  public void mouseExited(MouseEvent e) {
    //kod wykonywany, kiedy kursor opuści z obszaru komponentu
  }
  public void mousePressed(MouseEvent e) {
    //kod wykonywany, kiedy wciśnięty zostanie przycisk myszy
  }
  public void mouseReleased(MouseEvent e) {
   //kod wykonywany, kiedy przycisk myszy zostanie zwolniony
  }
}
```

Każda metoda otrzymuje jako argument obiekt klasy `MouseEvent` pozwalający określić rodzaj zdarzenia oraz współrzędne kursora. Jeśli chcemy dowiedzieć się, który przycisk został wciśnięty, należy skorzystać z metody `getButton`[5], współrzędne natomiast otrzymamy, wywołując metody `getX` i `getY`. Metoda `getButton` zwraca wartość typu `int`, którą należy porównywać ze stałymi zdefiniowanych w klasie `MouseEvent`:

- `MouseEvent.BUTTON1`,
- `MouseEvent.BUTTON2`,
- `MouseEvent.BUTTON3`,
- `MouseEvent.NOBUTTON`.

Pierwsze trzy określają numer przycisku, natomiast ostatnia informuje, że żaden przycisk podczas danego zdarzenia nie był wciśnięty. Na listingu 8.19 jest widoczny przykładowy

[5] Metoda ta jest dostępna, począwszy od JDK 1.4, wcześniejsze wersje JDK jej nie zawierają.

aplet, który wyświetla współrzędne ostatniego kliknięcia myszą oraz informację o tym, który przycisk został użyty.

Listing 8.19.

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

public class Aplet extends JApplet implements MouseListener {
  String tekst = "";
  public void init() {
    addMouseListener(this);
  }
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString(tekst, 20, 20);
  }
  public void mouseClicked(MouseEvent evt) {
    int button = evt.getButton();
    switch(button){
      case MouseEvent.BUTTON1 : tekst = "Przycisk 1, ";break;
      case MouseEvent.BUTTON2 : tekst = "Przycisk 2, ";break;
      case MouseEvent.BUTTON3 : tekst = "Przycisk 3, ";break;
      default : tekst = "";
    }
    tekst += "współrzędne: x = " + evt.getX() + ", ";
    tekst += "y = " + evt.getY();
    repaint();
  }
  public void mouseEntered(MouseEvent evt){}
  public void mouseExited(MouseEvent evt){}
  public void mousePressed(MouseEvent evt){}
  public void mouseReleased(MouseEvent evt){}
}
```

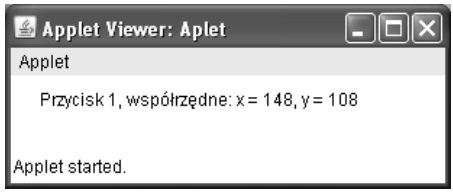
Ponieważ interesuje nas jedynie zdarzenie polegające na kliknięciu myszą, treść metod niezwiązanych z nim, czyli `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`, pozostaje pusta. Niemniej ich definicje muszą znajdować się w klasie `Aplet`, gdyż wymusza to interfejs `MouseListener` (por. lekcje 26. i 27.).

W metodzie `mouseClicked`, wywołując metodę `getButton`, odczytujemy kod naciśniętego przycisku i przypisujemy go zmiennej `button`. Następnie sprawdzamy wartość tej zmiennej za pomocą instrukcji wyboru `switch` i, w zależności od tego, czy jest to wartość `BUTTON1`, `BUTTON2` czy `BUTTON3`, przypisujemy odpowiedni ciąg znaków zmiennej `tekst`, która odpowiada za napis, jaki będzie wyświetlany na ekranie. W dalszej części kodu dodajemy do zmiennej `tekst` określenie współrzędnej `x` i współrzędnej `y`, w której nastąpiło kliknięcie. Wartości wymienionych współrzędnych odczytujemy dzięki metodom `getX` i `getY`. Na końcu metody `mouseClicked` wywołujemy metodę `repaint`, która spowoduje odświeżenie ekranu. Odświeżenie ekranu wiąże się oczywiście z wywołaniem metody `paint`, w której wykorzystujemy znaną nam dobrze metodę `drawString` do wyświetlenia tekstu zawartego w polu `tekst`.

Bardzo ważny jest również fragment kodu, który wykonujemy w metodzie `init`. Otóż wywołujemy tam metodę `addMouseListener`, której w argumencie przekazujemy wskazanie do apletu. Takie wywołanie oznacza, że wszystkie zdarzenia związane z obsługą myszy, określone przez interfejs `MouseListener`, będą obsługiwane przez obiekt przekazany tej metodzie jako argument. Ponieważ argumentem jest sam obiekt apletu (choć może być to obiekt dowolnej klasy implementującej interfejs `MouseListener`), to w tym przypadku informujemy po prostu maszynę wirtualną, że nasz aplet samodzielnie będzie obsługiwał zdarzenia związane z myszą.

O wywołaniu metody `addMouseListener` koniecznie musimy pamiętać, gdyż jeśli jej zabraknie, kompilator nie zgłosi żadnego błędu, a program po prostu nie będzie działał. Jest to błąd stosunkowo często popełniany przez początkujących programistów. Jeśli jednak będziemy o tej instrukcji pamiętać, to po skompilowaniu i uruchomieniu apletu oraz kliknięciu pierwszego przycisku na ekranie zobaczymy widok zaprezentowany na rysunku 8.14.

Rysunek 8.14. *Wynik działania apletu wyświetlającego informacje o kliknięciach myszą*



Skoro jednak metoda `addActionListener` może przekazać obsługę zdarzeń dowolnemu obiektowi implementującemu interfejs `MouseListener`, zobaczmy, jak by to wyglądało w praktyce. Napiszemy dodatkową klasę pakietową współpracującą z klasą `Aplet` i odpowiedzialną za obsługę myszy. Aby nie komplikować kodu, jej zadaniem będzie wyświetlenie na konsoli współrzędnych ostatniego kliknięcia. Kod obu klas został zaprezentowany na listingu 8.20.

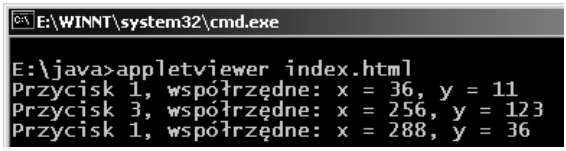
Listing 8.20.

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

public class Aplet extends JApplet {
  public void init() {
    addMouseListener(new MyMouseListener());
  }
}

class MyMouseListener implements MouseListener {
  public void mouseClicked(MouseEvent evt) {
    String tekst = "";
    int button = evt.getButton();
    switch(button){
      case MouseEvent.BUTTON1 : tekst = "Przycisk 1, ";break;
      case MouseEvent.BUTTON2 : tekst = "Przycisk 2, ";break;
      case MouseEvent.BUTTON3 : tekst = "Przycisk 3, ";break;
      default : tekst = "";
    }
    tekst += "współrzędne: x = " + evt.getX() + ", ";
```

```
        tekst += "y = " + evt.getY();
        System.out.println(tekst);
      }
      public void mouseEntered(MouseEvent evt){}
      public void mouseExited(MouseEvent evt){}
      public void mousePressed(MouseEvent evt){}
      public void mouseReleased(MouseEvent evt){}
    }
```

Klasa `Aplet` nie implementuje w tej chwili interfejsu `MouseListener`, gdyż obsługa myszy jest przekazywana innej klasie. Pozostała w niej jedynie metoda `init`, w której wywołujemy metodę `addActionListener`. Argumentem przekazanym `addActionListener` jest nowy obiekt klasy `MyMouseListener`. Oznacza to, że aplet ma reagować na zdarzenia myszy, ale ich obsługa została przekazana obiektowi klasy `MyMouseListener`.

Klasa `MyMouseListener` jest klasą pakietową, jest zatem zdefiniowana w tym samym pliku, co klasa `Aplet` (por. lekcja 17.). Implementuje ona oczywiście interfejs `MouseListener`, inaczej obiekt tej klasy nie mógłby być argumentem metody `addActionListener`. Wewnątrz klasy `MyMouseListener` zostały zdefiniowane metody z interfejsu, jednak kod wykonywalny zawiera jedynie metoda `mouseClicked`. Jej treść jest bardzo podobna do kodu metody `mouseClicked` z poprzedniego przykładu. Jedyną różnicą jest to, że zmienna `tekst` jest zdefiniowana wewnątrz tej metody i zamiast metody `repaint` jest wykonywana instrukcja `System.out.println` wyświetlająca na konsoli współrzędne kliknięcia. Uruchomienie takiego apletu spowoduje, że współrzędne kliknięcia będą się pojawiały na konsoli, tak jak jest to widoczne na rysunku 8.15.



Rysunek 8.15. *Współrzędne kliknięcia pojawiają się na konsoli*

Nie możemy tym razem wyświetlać tekstu bezpośrednio w obszarze apletu, gdyż klasa `MyMouseListener` nie ma do niego dostępu. Jak go uzyskać? Można by na przykład przekazać referencję do obiektu apletu w konstruktorze klasy `MyMouseListener` (por. ćwiczenie 40.4 z sekcji „Ćwiczenia do samodzielnego wykonania"). O wiele jednak lepszym rozwiązaniem byłoby zastosowanie klasy wewnętrznej, a jeszcze lepiej anonimowej klasy wewnętrznej (por. lekcje 28 – 30). Takie rozwiązanie zostanie pokazane już w kolejnej lekcji.

Interfejs MouseMotionListener

Interfejs `MouseMotionListener` pozwala na obsługiwanie zdarzeń związanych z ruchem myszy. Definiuje on dwie metody, które są widoczne w tabeli 8.5. Pierwsza z nich jest wywoływana, kiedy przycisk myszy został wciśnięty i mysz się porusza, natomiast druga przy każdym ruchu myszy bez wciśniętego przycisku.

Opierając się zatem na przykładzie z listingu 8.19, bez problemu powinniśmy napisać aplet, który będzie wyświetlał aktualne współrzędne położenia kursora myszy. Kod realizujący to zadanie został przedstawiony na listingu 8.21.

Tabela 8.5. *Metody interfejsu MouseMotionListener*

Deklaracja metody	Opis	Od JDK
`void mouseDragged(MouseEvent e)`	Metoda wywoływana podczas ruchu myszy, kiedy wciśnięty jest jeden z klawiszy.	1.1
`void mouseMoved(MouseEvent e)`	Metoda wywoływana przy każdym ruchu myszy, o ile nie jest wciśnięty żaden klawisz.	1.1

Listing 8.21.

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

public class Aplet extends JApplet implements MouseMotionListener {
  String tekst = "";
  public void init() {
    addMouseMotionListener(this);
  }
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString(tekst, 20, 20);
  }
  public void mouseMoved(MouseEvent evt) {
    tekst = "zdarzenie mouseMoved, ";
    tekst += "współrzędne: x = " + evt.getX() + ", ";
    tekst += "y = " + evt.getY();
    repaint();
  }
  public void mouseDragged(MouseEvent evt) {
    tekst = "zdarzenie mouseDragged, ";
    tekst += "współrzędne: x = " + evt.getX() + ", ";
    tekst += "y = " + evt.getY();
    repaint();
  }
}
```

Sposób postępowania jest tu identyczny jak w przypadku interfejsu `MouseListener`. Klasa `Aplet` implementuje interfejs `MouseMotionListener`, zawiera zatem metody `mouseMoved` i `mouseDragged`. W metodzie `init` jest wywoływana metoda `addMouseMotionListener` (analogicznie jak we wcześniejszych przykładach `addMouseListener`), dzięki czemu wszystkie zdarzenia związane z poruszaniem myszy będą obsługiwane przez klasę `Aplet`. W metodach `mouseMoved` oraz `mouseDragged` odczytujemy współrzędne kursora myszy dzięki funkcjom `getX` i `getY`, przygotowujemy treść pola `tekst` oraz wywołujemy metodę `repaint` odświeżającą ekran. W metodzie `paint` wyświetlamy zawartość pola `tekst` na ekranie.

Dodatkowe parametry zdarzenia

Niekiedy przy przetwarzaniu zdarzeń związanych z obsługą myszy zachodzi potrzeba sprawdzenia stanu klawiszy specjalnych *Alt*, *Shift* lub *Ctrl*. Z taką sytuacją mamy do czynienia np. wtedy, kiedy program ma inaczej reagować, kiedy kliknięcie nastąpiło

z równoczesnym wciśnięciem jednego z wymienionych klawiszy. Musi zatem istnieć sposób pozwalający na sprawdzenie, czy mamy do czynienia z taką specjalną sytuacją. Tym sposobem jest dokładniejsze zbadanie obiektu klasy `MouseEvent`, który jest przekazywany funkcji obsługującej każde zdarzenie związane z obsługą myszy.

Klasa `MouseEvent` (a dokładniej klasa `InputEvent`, z której `MouseEvent` dziedziczy) udostępnia metodę o nazwie `getModifiers`, zwracającą wartość typu `int`, której poszczególne bity określają dodatkowe parametry zdarzenia. Stan tych bitów badamy poprzez porównanie z jedną ze stałych[6] zdefiniowanych w klasie `MouseEvent`. W sumie jest dostępnych kilkadziesiąt stałych, w większości odziedziczonych z klas bazowych, ich opis można znaleźć w dokumentacji JDK. Dla nas interesujące są trzy wartości:

- `MouseEvent.SHIFT_DOWN_MASK`,
- `MouseEvent.ALT_DOWN_MASK`,
- `MouseEvent.CTRL_DOWN_MASK`.

Pierwsza z nich oznacza, że został wciśnięty klawisz *Shift*, druga, że został wciśnięty klawisz *Alt*, a trzecia, że został wciśnięty klawisz *Ctrl*[7]. Porównania z wartością zwróconą przez `getModifiers` dokonujemy przez wykonanie operacji bitowej `AND`, czyli iloczynu bitowego. Jeśli więc wynikiem operacji:

```
getModifiers() & MouseEvent.SHIFT_DOWN
```

jest wartość 0, oznacza to, że klawisz *Shift* nie był wciśnięty, a jeśli wartość tej operacji jest różna od 0, oznacza to, że *Shift* był wciśnięty. Przykładowy aplet wykorzystujący opisaną technikę do stwierdzenia, które z klawiszy funkcyjnych były wciśnięte podczas przesuwania kursora myszy, jest widoczny na listingu 8.22, a przykładowy efekt jego działania przedstawia rysunek 8.16.

Listing 8.22.

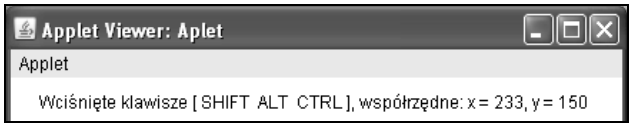
```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;

public class Aplet extends JApplet implements MouseMotionListener {
  String tekst = "";
  public void init() {
    addMouseMotionListener(this);
  }
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString(tekst, 20, 20);
  }
  public void mouseMoved(MouseEvent evt) {
    tekst = "Wciśnięte klawisze [";
    int modifiers = evt.getModifiersEx();
    if((modifiers & MouseEvent.SHIFT_DOWN_MASK) != 0){
      tekst += " SHIFT ";
```

[6] Przez stałą rozumiemy statyczne i finalne pole klasy.

[7] W wersjach JDK poniżej 1.4 wykorzystywane były stałe `SHIFT_DOWN`, `ALT_DOWN` i `CTRL_DOWN`.

```
    }
    if((modifiers & MouseEvent.ALT_DOWN_MASK) != 0){
      tekst += " ALT ";
    }
    if((modifiers & MouseEvent.CTRL_DOWN_MASK) != 0){
      tekst += " CTRL ";
    }
    tekst += "], ";

    tekst += "współrzędne: x = " + evt.getX() + ", ";
    tekst += "y = " + evt.getY();
    repaint();
  }
  public void mouseDragged(MouseEvent evt){}
}
```

Rysunek 8.16. *Przykładowy efekt działania apletu z listingu 8.21*



Dźwięki

Pisane przez nas aplety możemy wyposażyć w możliwość odtwarzania dźwięków. Java obsługuje standardowo kilka formatów plików dźwiękowych, są to *AU*, *AIFF*, *WAVE* oraz *MIDI*. W klasie `JApplet` została zdefiniowana metoda `play`, która pobiera i odtwarza klip dźwiękowy. Występuje ona w dwóch przeciążonych wersjach:

```
play(URL url)
play(URL url, String name).
```

Pierwsza z nich przyjmuje jako argument obiekt klasy `URL` bezpośrednio wskazujący na plik dźwiękowy, druga wymaga podania argumentu klasy `URL` wskazującego na umiejscowienie pliku (np. *http://host.domena/java/sound/*) i drugiego określającego nazwę pliku. Jeśli plik znajduje się w strukturze katalogów naszego serwera, wygodniejsze może być użycie drugiej postaci konstruktora, podobnie jak w przypadku metody `getImage` omawianej w lekcji 39. Przykład apletu, który podczas uruchamiania odtwarza plik dźwiękowy, jest widoczny na listingu 8.23.

Listing 8.23.

```
import javax.swing.JApplet;

public class Aplet extends JApplet {
  public void start() {
    play (getDocumentBase(), "ding.au");
  }
}
```

Drugim sposobem na odtwarzanie dźwięku jest wykorzystanie interfejsu `AudioClip`. Interfejs ten definiuje trzy metody: `start`, `stop` i `loop`. Metoda `start` rozpoczyna odtwarzanie dźwięku, `stop` kończy odtwarzanie, natomiast `loop` rozpoczyna odtwarzanie

dźwięku w pętli. Ponieważ AudioClip został zdefiniowany w pakiecie java.applet, tym razem jako klasę apletu wykorzystamy Applet (zamiast JApplet). Obiekt implementujący interfejs AudioClip otrzymamy, wywołując metodę getAudioClip klasy Applet. Metoda ta występuje w dwóch przeciążonych wersjach:

```
getAudioClip(URL url)
getAudioClip(URL url, String name)
```

Znaczenie argumentów jest takie samo, jak w przypadku opisanej wyżej metody play. Proste wykorzystanie interfejsu AudioClip ilustruje przykład z listingu 8.24. Po uruchomieniu apletu rozpoczyna się odtwarzanie dźwięku, a przy kończeniu jego pracy odtwarzanie jest zatrzymywane.

Listing 8.24.

```
import java.applet.*;

public class Aplet extends Applet {
  AudioClip audioClip;
  public void init() {
    audioClip = getAudioClip(getDocumentBase(), "ding.au");
  }
  public void start() {
    audioClip.loop();
  }
  public void stop() {
    audioClip.stop();
  }
}
```

Znacznie ciekawsze byłoby jednak połączenie możliwości odtwarzania dźwięków oraz reakcji na zdarzenia związane z obsługą myszy. Wykorzystanie możliwości, jakie dają interfejsy MouseListener oraz AudioClip, pozwala na napisanie apletu, który będzie np. odtwarzał plik dźwiękowy, kiedy użytkownik kliknie myszą. Aplet realizujący takie zadanie jest przedstawiony na listingu 8.25.

Listing 8.25.

```
import java.applet.*;
import java.awt.event.*;

public class Aplet extends Applet implements MouseListener {
  AudioClip audioClip;
  public void init() {
    addMouseListener(this);
    audioClip = getAudioClip(getDocumentBase(), "ding.au");
  }
  public void mouseClicked(MouseEvent evt) {
    audioClip.play();
  }
  public void mouseEntered(MouseEvent evt){}
  public void mouseExited(MouseEvent evt){}
  public void mousePressed(MouseEvent evt){}
  public void mouseReleased(MouseEvent evt){}
}
```

Klasa `Aplet` implementuje interfejs `MouseListener`, a zatem zawiera definicje wszystkich jego metod. Wykorzystujemy jednak jedynie metodę `mouseClicked`, która będzie wywoływana po każdym kliknięciu myszą. Rozpoczynamy w niej odtwarzanie dźwięku poprzez wywołanie metody `play` obiektu `audioClip`. Obiekt wskazywany przez pole `audioClip` uzyskujemy w metodzie `init` przez wywołanie metody `getAudioClip` klasy `Applet`, dokładnie w taki sam sposób, jak w poprzednim przykładzie. Nie zapominamy również o wywołaniu metody `addMouseListener`, bez której aplet nie będzie reagował na kliknięcia.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 40.1.

Zmień kod apletu z listingu 8.19 w taki sposób, aby reagował nie na kliknięcia, ale na samo naciśnięcie przycisku myszy.

Ćwiczenie 40.2.

Napisz aplet, w którym obsługa ruchów myszy będzie realizowana przez oddzielną klasę `MyMouseMotionListener`. Przy każdym ruchu myszy wyświetl współrzędne kursora myszy na konsoli.

Ćwiczenie 40.3.

Napisz aplet, który będzie odtwarzał plik dźwiękowy, kiedy użytkownik zbliży kursor myszy na mniej niż 10 pikseli od brzegów powierzchni apletu. Wysokość oraz szerokość obszaru apletu można uzyskać, wywołując metody `getWidth` oraz `getHeight` klasy `Applet`.

Ćwiczenie 40.4.

Napisz aplet, który przy kliknięciu będzie odtwarzał dźwięki. Odtwarzanie powinno być realizowane przez pakietową klasę `MyAudioClip` implementującą interfejs `AudioClip`.

Aplikacje

Lekcja 41. Tworzenie aplikacji

Na początku rozdziału 8. poznaliśmy różnicę między aplikacją i apletem, wiemy, że aplikacja potrzebuje do uruchomienia jedynie maszyny wirtualnej, a aplet jest programem wbudowanym, zagnieżdżonym w innym programie, najczęściej w przeglądarce internetowej. Wszystkie programy, które powstawały w rozdziałach 1. – 7., były właśnie aplikacjami, pracującymi jednak w trybie tekstowym. W tej lekcji zobaczymy, w jaki sposób tworzy się aplikacje pracujące w trybie graficznym, czyli popularne aplikacje okienkowe.

Pierwsze okno

W lekcji 37., na listingu 8.1 powstał nasz pierwszy aplet. Zobaczmy teraz, jak napisać aplikację, która będzie wykonywała to samo zadanie, czyli wyświetli napis na ekranie. Wymagać to będzie napisania klasy np. o nazwie `PierwszaAplikacja`, która będzie dziedziczyć z klasy `JFrame`. Jest to klasa zawarta w pakiecie `javax.swing`. Alternatywnie można użyć również klasy `Frame` z pakietu `java.awt`, jednak jest ona uznawana za przestarzałą. Kod pierwszej aplikacji został przedstawiony na listingu 8.26.

Listing 8.26.

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

public class PierwszaAplikacja extends JFrame {
  public PierwszaAplikacja() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(320, 200);
    setVisible(true);
  }
  public void paint(Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString ("Pierwsza aplikacja", 100, 100);
  }
  public static void main(String args[]) {
    new PierwszaAplikacja();
  }
}
```

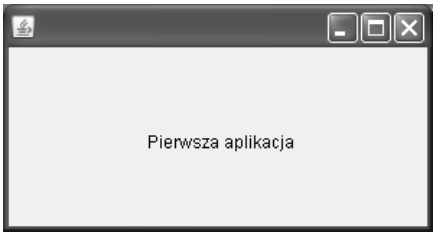
Za utworzenie okna odpowiada klasa `PierwszaAplikacja`, która dziedziczy z klasy `JFrame`. W konstruktorze za pomocą metody `setSize` ustalamy rozmiary okna, natomiast za pomocą metody `setVisible` powodujemy, że zostanie ono wyświetlone na ekranie. Za wyświetlenie na ekranie napisu odpowiada metoda `drawString` klasy `Graphics`, odbywa się to dokładnie w taki sam sposób, jak w przypadku omawianych w poprzednich lekcjach apletów. Należy również zwrócić uwagę na instrukcję:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

która powoduje, że domyślnym działaniem wykonywanym podczas zamykania okna (np. gdy użytkownik kliknie przycisk zamykający lub wybierze taką akcję z menu systemowego) będzie zakończenie działania całego programu (mówi o tym stała `EXIT_ON_CLOSE`). Jeśli ta instrukcja zostania pominięta, nie będzie można w standardowy sposób zakończyć działania aplikacji.

Obiekt klasy `PierwszaAplikacja` jest tworzony w metodzie `main`, od której rozpoczyna się wykonywanie kodu. Po uruchomieniu zobaczymy widok przedstawiony na rysunku 8.17.

Wraz z platformą Java2 SE5 pojawił się jednak nowy model obsługi zdarzeń dla biblioteki *Swing*, w którym operacje związane z komponentami (a okno aplikacji jest komponentem) nie powinny być obsługiwane bezpośrednio, ale trafiać do kolejki zdarzeń. Dotyczy to również samego uruchamiania aplikacji. Należy użyć metody `invokeLater` klasy



Rysunek 8.17. *Wygląd prostej aplikacji*

`SwingUtilities`, która umieści nasze wywołanie w kolejce zdarzeń. Argumentem tej metody musi być obiekt implementujący interfejs `Runnable`, a operacja, którą chcemy wykonać, powinna się znaleźć w metodzie `run` tego interfejsu. Zgodnie z tym standardem metoda `main` powinna mieć postać:

```
public static void main(String args[]) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      new PierwszaAplikacja();
    }
  });
}
```

Ten też sposób będzie stosowany na listingach w dalszej części rozdziału.

Zdarzenia związane z oknem

Z każdym oknem związany jest zestaw zdarzeń reprezentowanych przez interfejs `WindowListener`. Interfejs ten definiuje metody, które zostały zebrane w tabeli 8.6. Oczywiście sama implementacja interfejsu to nie wszystko, musimy jeszcze poinformować system, że to właśnie nasze okno ma odbierać wysyłane komunikaty, co robimy, wywołując metodę `addWindowListener` (por. metody `addMouseListener` i `addMouseMotionListener` z lekcji 40.). Zatem zamiast stosować metodę `setDefaultCloseOperation`, można również w nieco inny sposób obsłużyć zdarzenie polegające na zamknięciu okna. Przykładowy kod klasy `Aplikacja` tworzącej okno reagujące na próby zamknięcia przy użyciu interfejsu `WindowListener` jest widoczny na listingu 8.27.

Tabela 8.6. *Metody interfejsu WindowListener*

Deklaracja metody	Opis	Od JDK
`void windowActivated(WindowEvent e)`	Metoda wywoływana po aktywacji okna.	1.1
`void windowClosed(WindowEvent e)`	Metoda wykonywana, kiedy okno zostanie zamknięte poprzez wywołanie metody `dispose`.	1.1
`void windowClosing(WindowEvent e)`	Metoda wywoływana, kiedy następuje próba zamknięcia okna przez użytkownika.	1.1
`void windowDeactivated(WindowEvent e)`	Metoda wywoływana po dezaktywacji okna.	1.1
`void windowDeiconified(WindowEvent e)`	Metoda wywoływana, kiedy okno zmieni stan ze zminimalizowanego na normalny.	1.1
`void windowIconified(WindowEvent e)`	Metoda wywoływana po minimalizacji okna.	1.1
`void windowOpened(WindowEvent e)`	Metoda wywoływana po otwarciu okna.	1.1

Listing 8.27.

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

public class Aplikacja extends JFrame implements WindowListener {
  public Aplikacja() {
    addWindowListener(this);
    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
  public void windowClosing(WindowEvent e){
    dispose();
  }
  public void windowClosed(WindowEvent e){}
  public void windowOpened(WindowEvent e){}
  public void windowIconified(WindowEvent e){}
  public void windowDeiconified(WindowEvent e){}
  public void windowActivated(WindowEvent e){}
  public void windowDeactivated(WindowEvent e){}
}
```

Klasa `Aplikacja` dziedziczy z klasy `JFrame` i implementuje interfejs `WindowListener`. W konstruktorze poprzez wywołanie metody `addWindowListener` (z parametrem `this` wskazującym na obiekt aplikacji), powodujemy, że informacje o zdarzeniach będą przekazywane właśnie obiektowi aplikacji, czyli że będą wywoływane metody `windowClosing`, `windowClosed`, `windowOpened`, `windowIconified`, `windowDeiconified`, `windowActivated`, `windowDeactivated` z klasy `Aplikacja`. Ponieważ interesuje nas jedynie obsługa zdarzenia polegającego na zamknięciu okna, oprogramowujemy jedynie metodę `windowClosing`. Jest ona wywoływana, kiedy użytkownik próbuje zamknąć okno poprzez wybranie odpowiedniej pozycji z menu systemowego bądź też poprzez kliknięcie odpowiedniej ikony paska tytułowego okna. W takiej sytuacji wywołujemy metodę `dispose`, która powoduje zwolnienie zasobów związanych z oknem, zamknięcie okna i, jeżeli jest to ostatnie okno aplikacji, zakończenie pracy aplikacji.

Obsługa zdarzeń przez klasy anonimowe

Przykład z listingu 8.27 przedstawiał aplikację reagującą na zdarzenia związane z jej oknem. Konkretnie była to aplikacja, która kończyła swoje działanie po wybraniu przez użytkownika odpowiedniej pozycji z menu systemowego lub też kliknięciu właściwej ikony paska tytułowego. Możliwe to było dzięki implementacji interfejsu `WindowListener` bezpośrednio przez klasę okna. W takim jednak przypadku konieczna była deklaracja wszystkich metod klasy `WindowListener`, nawet tych, które nie były wykorzystywane. Zamiast tego wygodniej jest więc skorzystać z klasy adaptera, czyli specjalnej klasy zawierającej puste implementacje metod danego interfejsu. W przypadku interfejsu

WindowListener jest to klasa WindowAdapter. Jeśli więc z WindowAdapter wyprowadzimy naszą własną klasę i przesłonimy w niej wybraną metodę, nie będzie konieczności definiowania pozostałych. Taka sytuacja została zobrazowana na listingu 8.28.

Listing 8.28.

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

public class Aplikacja extends JFrame {
  class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent e){
      dispose();
    }
  }
  public Aplikacja() {
    addWindowListener(new MyWindowAdapter());
    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Tym razem w klasie Aplikacja została zdefiniowana klasa wewnętrzna MyWindowAdapter, pochodna od WindowAdapter, a w niej metoda windowClosing. W metodzie tej wywoływana jest natomiast metoda dispose klasy Aplikacja. Jest to możliwe, jako że klasa wewnętrzna ma dostęp do metod klasy zewnętrznej (por. lekcja 28). W konstruktorze klasy Aplikacja została wywołana metoda addWindowListener i został jej przekazany w postaci argumentu obiekt klasy MyWindowAdapter (addWindowListener(new MyWindowAdapter());). To nic innego jak informacja, że zdarzeniami związanymi z oknem będzie się zajmował właśnie ten obiekt. Tak więc całą obsługą zdarzenia zajmować się będzie teraz klasa MyWindowAdapter.

Zauważmy jednak, że ta klasa mogłaby być z powodzeniem klasą anonimową (por. lekcja 30.), jej nazwy w przedstawionej sytuacji tak naprawdę do niczego nie potrzebujemy. Program mógłby więc przyjąć postać widoczną na listingu 8.29.

Listing 8.29.

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

public class Aplikacja extends JFrame {
  public Aplikacja() {
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          dispose();
```

```
        }
      }
    );
    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Spójrzmy: w konstruktorze jest wywoływana metoda `addWindowListener` oznajmiająca, że zdarzenia związane z obsługą okna będą przekazywane obiektowi będącemu argumentem tej metody. Tym argumentem jest z kolei obiekt klasy anonimowej pochodnej od `WindowAdapter`. Ponieważ klasa anonimowa jest z natury rzeczy klasą wewnętrzną, ma ona dostęp do składowych klasy zewnętrznej — `Aplikacja` — i może wywołać metodę `dispose` zwalniającą zasoby i zamykającą okno aplikacji.

Podobnie możemy postąpić przy obsłudze zdarzeń związanych z myszą. Jeśli potrzebujemy implementacji interfejsu `MouseListener`, należy skorzystać z klasy `MouseAdapter`, jeśli natomiast niezbędny jest interfejs `MouseMotionListener`, należy skorzystać z klasy `MouseMotionAdapter`. Oba te adaptery zdefiniowane są w pakiecie `java.awt`. Pakiet `javax.swing` udostępnia natomiast dodatkowy adapter zbiorczy implementujący wszystkie interfejsy związane z myszą. Jest to `MouseInputAdapter`.

Powróćmy więc do kodu apletu z listingu 8.21 z lekcji 40., który pokazywał aktualne współrzędne kursora myszy, i przeróbmy go w taki sposób, aby do obsługi zdarzeń był wykorzystywany obiekt anonimowej klasy dziedziczącej z `MouseInputAdapter`. Kod takiego apletu jest widoczny na listingu 8.30 (usunięta zastała jedynie istniejąca na listingu 8.21 obsługa metody `mouseDragged`).

Listing 8.30.

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

public class Aplet extends JApplet {
  String tekst = "";
  public void init() {
    addMouseMotionListener(
      new MouseInputAdapter(){
        public void mouseMoved(MouseEvent evt) {
          tekst = "zdarzenie mouseMoved, ";
          tekst += "współrzędne: x = " + evt.getX() + ", ";
          tekst += "y = " + evt.getY();
          repaint();
        }
```

```
      }
    );
  }
  public void paint (Graphics gDC) {
    gDC.clearRect(0, 0, getSize().width, getSize().height);
    gDC.drawString(tekst, 20, 20);
  }
}
```

Menu

Rzadko która aplikacja okienkowa może obyć się bez menu. W Javie w celu dodania menu musimy skorzystać z kilku klas: `JMenuBar`, `JMenu` i `JMenuItem`. Pierwsza z nich opisuje pasek menu, druga menu znajdujące się na tym pasku, a trzecia poszczególne elementy menu. Pasek menu dodajemy do okna aplikacji za pomocą metody `setJMenuBar`, natomiast menu do paska dodajemy za pomocą metody `add`. Jak to wygląda w praktyce, obrazuje listing 8.31.

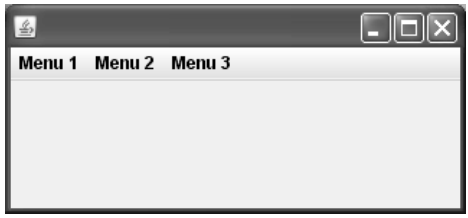
Listing 8.31.

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

public class JAplikacja extends JFrame {
  public Aplikacja() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JMenuBar mb = new JMenuBar();

    JMenu menu1 = new JMenu("Menu 1");
    JMenu menu2 = new JMenu("Menu 2");
    JMenu menu3 = new JMenu("Menu 3");

    mb.add(menu1);
    mb.add(menu2);
    mb.add(menu3);

    setJMenuBar(mb);

    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

W konstruktorze tworzymy nowy obiekt klasy `JMenuBar` i przypisujemy go zmiennej `mb`. Następnie tworzymy trzy obiekty klasy `JMenu` i dodajemy je do paska memu (czyli obiektu `mb`) za pomocą metody `add`. W konstruktorze klasy `JMenu` przekazujemy nazwy

menu, czyli tekst, który będzie przez nie wyświetlany. Pasek menu dodajemy do okna przez wywołanie metody `setJMenuBar`. Ostatecznie po skompilowaniu i uruchomieniu aplikacji na ekranie zobaczymy widok zaprezentowany na rysunku 8.18.

Rysunek 8.18. *Aplikacja z trzema pozycjami menu*



Do tak stworzonego menu należy dodać poszczególne pozycje. Służy do tego klasa `JMenuItem`. Obiekt tej klasy dodajemy, stosując metodę `add`. Jeśli zatem każde menu utworzone w aplikacji z listingu 8.31 miałoby mieć po dwie pozycje, konstruktor należałoby zmodyfikować w sposób widoczny na listingu 8.32.

Listing 8.32.

```
public Aplikacja() {
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JMenuBar mb = new JMenuBar();

  JMenu menu1 = new JMenu("Menu 1");
  JMenu menu2 = new JMenu("Menu 2");
  JMenu menu3 = new JMenu("Menu 3");

  JMenuItem menuItem11 = new JMenuItem("Pozycja 1");
  JMenuItem menuItem12 = new JMenuItem("Pozycja 2");

  JMenuItem menuItem21 = new JMenuItem("Pozycja 1");
  JMenuItem menuItem22 = new JMenuItem("Pozycja 2");

  JMenuItem menuItem31 = new JMenuItem("Pozycja 1");
  JMenuItem menuItem32 = new JMenuItem("Pozycja 2");

  menu1.add(menuItem11);
  menu1.add(menuItem12);

  menu2.add(menuItem21);
  menu2.add(menuItem22);

  menu3.add(menuItem31);
  menu3.add(menuItem32);

  mb.add(menu1);
  mb.add(menu2);
  mb.add(menu3);

  setJMenuBar(mb);

  setSize(320, 200);
  setVisible(true);
}
```

Tworzymy sześć różnych obiektów klasy `JMenuItem` odpowiadających poszczególnym pozycjom menu. Obiekty te dołączamy do kolejnych menu, wywołując metody `add` klasy `JMenu`. Czyli, przykładowo, instrukcja `menu1.add(menuItem11);` powoduje dodanie pierwszej pozycji do pierwszego menu. Po wykonaniu wszystkich instrukcji powyższego kodu każde menu będzie miało po dwie pozycje o nazwach *Pozycja 1* i *Pozycja 2*. Wygląd rozwiniętego menu przedstawia rysunek 8.19.

Rysunek 8.19.
Menu zawierające trzy pozycje

Wiemy już, jak tworzyć menu, warto więc teraz zadać pytanie, w jaki sposób spowodować, aby program reagował na wybranie jednej z pozycji. Łatwo się zapewne domyślić, że trzeba będzie skorzystać z jakiegoś interfejsu typu *MenuListener* i przekazać zdarzenia do jakiegoś obiektu, być może obiektu aplikacji, być może obiektu dodatkowej klasy.

Faktycznie tak należy postąpić. Co prawda, nie istnieje interfejs o nazwie *MenuListener*, skorzystać należy zatem z interfejsu `ActionListener`. Jest w nim zdefiniowana tylko jedna metoda o nazwie `actionPerformed`. Do reagowania na wybranie danej pozycji menu można więc zastosować technikę przedstawioną na listingu 8.33.

Listing 8.33.

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

public class Aplikacja extends JFrame {
  private JMenuItem miZamknij;
  private ActionListener al = new ActionListener(){
    public void actionPerformed(ActionEvent e) {
      if(e.getSource() == miZamknij)
        dispose();
    }
  };

  public Aplikacja() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JMenuBar mb = new JMenuBar();
    JMenu menu = new JMenu("Plik");

    miZamknij = new JMenuItem("Zamknij");
    menu.add(miZamknij);
    mb.add(menu);

    setJMenuBar(mb);
    miZamknij.addActionListener(al);

    setSize(320, 200);
    setVisible(true);
```

```
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Aplikacja ma w tej chwili jedynie menu *Plik* z jedną pozycją o nazwie *Zamknij*, tak jak jest to widoczne na rysunku 8.20. Sposób utworzenia menu jest analogiczny jak w poprzednich przykładach, z tą różnicą, że zmienna określająca pozycję menu *Zamknij* jest prywatnym polem klasy `Aplikacja`. Chodzi o to, abyśmy mieli do tej pozycji dostęp nie tylko w obrębie konstruktora, ale w całej klasie. Po utworzeniu paska menu (obiekt `mb`), samego menu (obiekt `menu`) oraz pozycji (obiekt `miZamknij`) dodajemy do tej pozycji obsługę zdarzeń, wywołując instrukcję:

```
miZamknij.addActionListener(al);
```

Rysunek 8.20. *Menu aplikacji z listingu 8.33*

Tym samym zdarzenia związane z obsługą tego menu będą przekazywane do metody `actionPerformed` obiektu `al`. Obiekt ten jest obiektem klasy anonimowej implementującej interfejs `ActionListener` i został zdefiniowany na początku klasy `Aplikacja`.

W metodzie `actionPerformed` sprawdzamy, czy obiektem, który wywołał zdarzenie, jest `miZamknij`, czyli pozycja menu o nazwie *Zamknij*. Jeśli tak, zamykamy okno, a tym samym całą aplikację. Aby uzyskać referencję do obiektu, który wywołał zdarzenie, wywołujemy metodę `getSource` obiektu klasy `ActionEvent` otrzymanego jako argument metody `actionPerformed`.

Kaskadowe menu

Pozycje menu można łączyć kaskadowo, uzyskując rozbudowane struktury podmenu. Utworzenie tego typu konstrukcji jest możliwe poprzez dodanie do menu innego menu, czyli przekazanie metodzie `add` klasy `JMenu` (w postaci argumentu) innego obiektu tej klasy. Przykład tego typu konstrukcji jest widoczny na listingu 8.34.

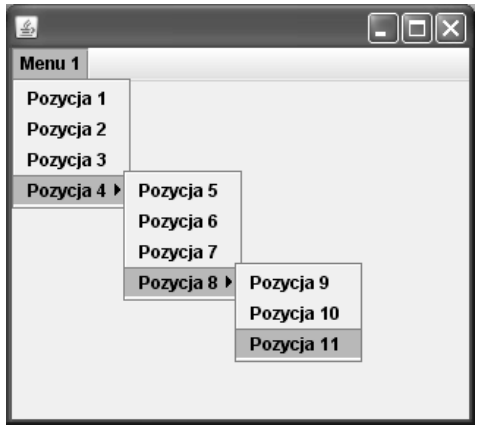
Listing 8.34.

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

public class Aplikacja extends JFrame {
  private ActionListener al = new ActionListener(){
    public void actionPerformed(ActionEvent e) {
```

```
        String text = ((JMenuItem)e.getSource()).getText();
        System.out.println(text);
      }
    };

    public Aplikacja() {
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JMenuBar mb = new JMenuBar();
      JMenu menu = new JMenu("Menu 1");
      JMenu submenu1 = new JMenu("Pozycja 4");
      JMenu submenu2 = new JMenu("Pozycja 8");

      menu.add(new JMenuItem("Pozycja 1"));
      menu.getItem(0).addActionListener(al);
      menu.add(new JMenuItem("Pozycja 2"));
      menu.getItem(1).addActionListener(al);
      menu.add(new JMenuItem("Pozycja 3"));
      menu.getItem(2).addActionListener(al);

      submenu1.add(new JMenuItem("Pozycja 5"));
      submenu1.getItem(0).addActionListener(al);
      submenu1.add(new JMenuItem("Pozycja 6"));
      submenu1.getItem(1).addActionListener(al);
      submenu1.add(new JMenuItem("Pozycja 7"));
      submenu1.getItem(2).addActionListener(al);

      submenu2.add(new JMenuItem("Pozycja 9"));
      submenu2.getItem(0).addActionListener(al);
      submenu2.add(new JMenuItem("Pozycja 10"));
      submenu2.getItem(1).addActionListener(al);
      submenu2.add(new JMenuItem("Pozycja 11"));
      submenu2.getItem(2).addActionListener(al);

      menu.add(submenu1);
      submenu1.add(submenu2);

      mb.add(menu);

      setJMenuBar(mb);
      setSize(320, 200);
      setVisible(true);
    }
    public static void main(String args[]) {
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          new Aplikacja();
        }
      });
    }
  }
```

Tworzymy trzy różne menu (obiekty klasy `Menu`) o nazwach `menu`, `submenu1` i `submenu2`. Pierwsze menu otrzymuje etykietę `Menu 1`, drugie — `Pozycja 4`, a trzecie — `Pozycja 8`. Dlaczego akurat takie nazwy etykiet? Otóż każde z menu otrzymuje po trzy pozycje, a następnie do pierwszego dodajemy `submenu1`, a do niego dodajemy `submenu2`. Etykieta

menu dodawanego staje się ostatnią pozycją menu, do którego zostało ono dodane. Jeśli więc etykietą `submenu1` jest `Pozycja 4`, to po dodaniu `submenu1` do `menu` etykieta `Pozycja 4` staje się ostatnią pozycją `menu`. Jeśli opis nadal nie jest do końca jasny, spójrzmy na rysunek 8.21, który powinien rozwiać wszelkie wątpliwości. Jest to ilustracja struktury menu z listingu 8.34.



Rysunek 8.21. *Kaskadowe menu z listingu 8.35*

Obsługą zdarzeń zajmuje się, podobnie jak we wcześniejszych przykładach, obiekt klasy anonimowej implementującej interfejs `ActionListener` reprezentowany przez prywatne pole `al` klasy `Aplikacja`. W związku z tym w stosunku do każdego obiektu reprezentującego daną pozycję menu jest wywoływana metoda `addActionListener` w postaci `addActionListener(al)`. Wykorzystywany jest tu jednak inny sposób dostępu do tych obiektów. Ponieważ są one tworzone bezpośrednio w metodzie `add` klasy `JMenu`, np.:

```
menu.add(new JMenuItem("Pozycja 1"));
```

to aby otrzymać odpowiednią referencję, wywołujemy metodę `getItem`, podając jako argument indeks wybranego menu. Indeks pierwszego menu ma wartość 0, drugiego — 1 itd.

Nowe instrukcje pojawiły się również w metodzie `actionPerformed`. Jej zadaniem jest wyświetlenie na konsoli nazwy menu, które zostało wybrane przez użytkownika. W celu pobrania tej nazwy jest wykonywana złożona instrukcja:

```
String text = ((JMenuItem)e.getSource()).getText();
```

Obiekt `e` to obiekt klasy `ActionEvent` zawierający wszystkie informacje o zdarzeniu. Metoda `getSource` pozwala na pobranie obiektu, który zapoczątkował dane zdarzenie, a więc obiektu menu wybranego przez użytkownika. Ponieważ typem wartości zwracanej przez `getSource` jest `Object`, dokonujemy rzutowania na typ `JMenuItem`, a potem wywołujemy metodę `getText`, która zwraca nazwę wybranego menu. Uzyskany tekst jest wyświetlany na konsoli za pomocą instrukcji:

```
System.out.println(text);
```

Zwróćmy w tym miejscu uwagę, że taki sposób obsługi był możliwy, jako że jedynymi źródłami zdarzeń były obiekty związane z pozycjami menu, czyli klasy `JMenuItem`.

Jako ćwiczenie do samodzielnego przemyślenia można zaproponować wykonanie takiego samego zadania w sytuacji, kiedy źródłami zdarzeń są również inne komponenty niż `JMenuItem`.

CheckBoxMenu

Oprócz zwykłych menu zaprezentowanych na wcześniejszych stronach pakiet `javax.swing` oferuje też menu, które umożliwiają zaznaczanie poszczególnych pozycji (ang. *check-box menu*). Za ich reprezentację odpowiada klasa o nazwie `JCheckBoxMenuItem`. Tworzenie tego typu menu odbywa się na takiej samej zasadzie jak zwykłych, z tą różnicą, że zamiast klasy `JMenuItem` korzystamy z `JCheckBoxMenuItem`. Zmienić niestety musimy jednak również sposób obsługi zdarzeń. Wykorzystać należy dodatkowy interfejs o nazwie `ItemListener`. Definiuje on tylko jedną metodę o nazwie `itemStateChanged`, która jest wywoływana za każdym razem, kiedy zmieni się stan komponentu (w naszym przypadku stan pozycji menu). Kod aplikacji zawierającej przykładowe menu posiadające możliwość zaznaczania poszczególnych pozycji został przedstawiony na listingu 8.35.

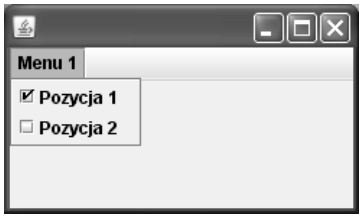
Listing 8.35.

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

public class Aplikacja extends JFrame {

  private JCheckBoxMenuItem menuItem1, menuItem2;

  private ItemListener il = new ItemListener(){
    public void itemStateChanged(ItemEvent e) {
      if(e.getSource() == menuItem1){
        if(menuItem1.getState())
          System.out.println("Pozycja 1 jest zaznaczona.");
         else
           System.out.println("Pozycja 1 nie jest zaznaczona.");
      }
      else if(e.getSource() == menuItem2){
        if(menuItem2.getState())
          System.out.println("Pozycja 2 jest zaznaczona.");
        else
          System.out.println("Pozycja 2 nie jest zaznaczona.");
      }
    }
  };

  public Aplikacja() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JMenuBar mb = new JMenuBar();
    JMenu menu = new JMenu("Menu 1");

    menuItem1 = new JCheckBoxMenuItem("Pozycja 1", true);
    menuItem1.addItemListener(il);
    menu.add(menuItem1);

    menuItem2 = new JCheckBoxMenuItem("Pozycja 2", false);
    menuItem2.addItemListener(il);
```

```
        menu.add(menuItem2);

        mb.add(menu);
        setJMenuBar(mb);

        setSize(320, 200);
        setVisible(true);
      }
      public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {
          public void run() {
            new Aplikacja();
          }
        });
      }
    }
```

Struktura menu jest tworzona w taki sam sposób, jak w przypadku elementów typu `MenuItem`. Wykorzystujemy jedynie dodatkowy argument konstruktora klasy `CheckBoxMenuItem`, który określa, czy dana pozycja ma być zaznaczona (`true`), czy nie (`false`). Wygląd menu jest widoczny na rysunku 8.22.

Rysunek 8.22. *Menu umożliwiające zaznaczanie poszczególnych pozycji*



Obsługa zdarzeń przychodzących z obiektów klasy `JCheckBoxMenuItem` wymaga implementacji interfejsu `ItemListener`, a tym samym metody `itemStateChanged`. Metoda ta będzie wywoływana za każdym razem, kiedy nastąpi zmiana stanu danej pozycji menu, czyli kiedy zostanie ona zaznaczona lub jej zaznaczenie zostanie usunięte. Sprawdzamy wtedy, która pozycja spowodowała wygenerowanie zdarzenia: `menuItem1` czy `menuItem2`. Referencję uzyskujemy przez wywołanie metody `getSource`. Jeśli chcemy sprawdzić, czy dana pozycja jest zaznaczona, czy nie, korzystamy z kolei z metody `getState`. Jeżeli zwróci ona wartość `true`, pozycja jest zaznaczona, jeśli `false` — nie jest.

Menu kontekstowe

Jeśli zachodzi potrzeba wyposażenia aplikacji w menu kontekstowe, istnieje oczywiście taka możliwość. Należy wtedy skorzystać z klasy `JPopupMenu`. Konstrukcja taka jest bardzo podobna do zwyczajnego menu, z tą różnicą, że menu kontekstowego nie dodaje się do paska menu. Przykładowa aplikacja zawierająca menu kontekstowe została przedstawiona na listingu 8.36.

Listing 8.36.

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

```
public class Aplikacja extends JFrame {
  private JPopupMenu popupMenu;
  private JMenuItem miPozycja1, miPozycja2, miZamknij;
  private ActionListener al = new ActionListener(){
    public void actionPerformed(ActionEvent e) {
      if(e.getSource() == miZamknij)
        dispose();
    }
  };

  private MouseAdapter ma = new MouseAdapter(){
    public void mousePressed(MouseEvent e) {
      if(e.isPopupTrigger())
        popupMenu.show(e.getComponent(), e.getX(), e.getY());
    }
    public void mouseReleased(MouseEvent e) {
      if(e.isPopupTrigger())
        popupMenu.show(e.getComponent(), e.getX(), e.getY());
    }
  };

  public Aplikacja() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    popupMenu = new JPopupMenu();
    miPozycja1 = new JMenuItem("Pozycja 1");
    miPozycja2 = new JMenuItem("Pozycja 2");
    miZamknij = new JMenuItem("Zamknij");

    miPozycja1.addActionListener(al);
    miPozycja2.addActionListener(al);
    miZamknij.addActionListener(al);

    popupMenu.add(miPozycja1);
    popupMenu.add(miPozycja2);
    popupMenu.addSeparator();
    popupMenu.add(miZamknij);

    addMouseListener(ma);

    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Obiekt menu kontekstowego tworzymy, wywołując konstruktor klasy `JPopupMenu`. Konstruktor ten może być bezargumentowy, tak jak na powyższym listingu, lub też przyjmować jeden argument klasy `String`. W tym drugim przypadku menu otrzyma identyfikującą je nazwę. Struktura menu jest tworzona w identyczny sposób, jak w przypadku wcześniej omawianych zwyczajnych menu — dodajemy po prostu kolejne obiekty

klasy `JMenuItem`, które staną się pozycjami menu. Dodatkowo za pomocą wywołania metody `addSeparator` dodany został również separator pozycji menu. Otrzymana struktura jest widoczna na rysunku 8.23. Również obsługa zdarzeń jest analogiczna, jak w poprzednich przykładach. Obiektem obsługującym zdarzenia powiązanym z każdym z obiektów `JMenuItem` jest obiekt klasy anonimowej pochodnej od `ActionListener`. W metodzie `actionPerformed` sprawdzane jest, czy wybrana pozycja menu to `Zamknij` (czyli czy obiektem źródłowym zdarzenia jest `miZamknij`). Jeśli tak, wywoływana jest metoda `dispose` obiektu aplikacji, która zwalnia zasoby związane z oknem i zamyka okno. Ponieważ jest to jedyne okno aplikacji, czynność ta jest równoznaczna z zakończeniem pracy całego programu.

Rysunek 8.23. *Aplikacja wyposażona w menu kontekstowe*

Aby jednak menu kontekstowe pojawiło się na ekranie, użytkownik aplikacji musi nacisnąć prawy klawisz myszy. To oznacza, że aplikacja musi reagować na zdarzenia związane z obsługą myszy, a jak wiemy z lekcji 38., wymaga to implementacji interfejsu `MouseListener`. Osiągamy to przez zastosowanie obiektu klasy anonimowej pochodnej od `MouseAdapter`. Obsługujemy metody `mousePressed` oraz `mouseReleased`. Jest to niezbędne, gdyż w różnych systemach w różny sposób wywoływane jest menu kontekstowe. W obu przypadkach za pomocą metody `isPopupTrigger` sprawdzamy, czy zdarzenie jest wynikiem wywołania przez użytkownika menu kontekstowego. Jeśli tak (wywołanie `isPopupTrigger` zwróciło wartość `true`), wyświetlamy menu w miejscu wskazywanym przez kursor myszy.

Wyświetlenie menu odbywa się za pomocą metody `show` obiektu `popupMenu`. Pierwszym parametrem jest obiekt, na którego powierzchni ma się pojawić menu i względem którego będą obliczane współrzędne wyświetlania, przekazywane jako drugi i trzeci argument. W naszym przypadku pierwszym argumentem jest po prostu obiekt aplikacji uzyskiwany przez wywołanie `e.getComponent()`. Metoda `getComponent` klasy `MouseEvent` zwraca bowiem obiekt, który zapoczątkował zdarzenie.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 41.1.

Popraw kod z listingu 8.26 w taki sposób, aby wyświetlany tekst znajdował się w centrum okna aplikacji.

Ćwiczenie 41.2.

Zmień kod z listingu 8.27 tak, aby obsługa zdarzeń odbywała się poprzez obiekt klasy wewnętrznej implementującej interfejs `WindowListener`.

Ćwiczenie 41.3.

Zmodyfikuj kod z listingu 8.27 tak, aby obsługa zdarzeń odbywała się poprzez obiekt niezależnej klasy pakietowej implementującej interfejs `WindowListener`.

Ćwiczenie 41.4.

Napisz aplikację zawierającą wielopoziomowe menu kontekstowe.

Ćwiczenie 41.5.

Napisz aplikację, która będzie wyświetlała w swoim oknie aktualne współrzędne kursora myszy.

Ćwiczenie 41.6.

Napisz aplikację zawierającą menu. Po wybraniu z niego dowolnej pozycji powinno pojawić się nowe okno (w tym celu utwórz nowy obiekt klasy `JFrame`). Okno to musi dać się zamknąć przez kliknięcie odpowiedniej ikony z paska tytułu.

Lekcja 42. Komponenty

Każda aplikacja okienkowa, oprócz menu, których różne rodzaje poznaliśmy w lekcji 41., jest także wyposażona w wiele innych elementów graficznych, takich jak przyciski, etykiety, pola tekstowe czy listy rozwijane[8]. Pakiet `javax.swing` zawiera oczywiście odpowiednią porcję klas, które pozwalają na zastosowanie tego rodzaju komponentów. Jest ich bardzo wiele, część z nich poznamy w ostatniej, 42. lekcji.

Klasa komponent

Praktycznie wszystkie graficzne komponenty, które pozwalają na zastosowanie typowych elementów środowiska okienkowego, dziedziczą, pośrednio lub bezpośrednio, z klasy `JComponent`, a zatem są wyposażone w jej metody[9]. Metod tych jest dosyć dużo, ich pełną listę można znaleźć w dokumentacji JDK. W tabeli 8.7 zostały natomiast zebrane niektóre z nich, przydatne przy wykonywaniu typowych operacji. Większość została odziedziczona z klasy `Component` biblioteki AWT. W kolumnie *Od JDK* została zawarta informacja na temat tego, kiedy po raz pierwszy dana metoda pojawiła się w JDK. Wartości w nawiasach oznaczają natomiast, kiedy dana metoda została redefiniowana w klasie `JComponent`.

[8] Często spotyka się też termin „lista rozwijalna”.

[9] A także klas, z których dziedziczy `JComponent`.

Tabela 8.7. *Wybrane metody klasy Component*

Deklaracja metody	Opis	Od JDK
`void addKeyListener(KeyListener l)`	Ustala obiekt, który będzie obsługiwał zdarzenia związane z klawiaturą.	1.1
`void addMouseListener(MouseListener l)`	Ustala obiekt, który będzie obsługiwał zdarzenia związane z klikaniem przyciskami myszy.	1.1
`void addMouseMotionListener (MouseMotionListener l)`	Ustala obiekt, który będzie obsługiwał zdarzenia związane z ruchami myszy.	1.1
`boolean contains(int x, int y)`	Sprawdza, czy komponent zawiera punkt o współrzędnych `x`, `y`.	1.1
`boolean contains(Point p)`	Sprawdza, czy komponent zawiera punkt wskazywany przez argument `p`.	1.1
`Color getBackground()`	Pobiera kolor tła komponentu.	1.0
`Rectangle getBounds()`	Zwraca rozmiar komponentu w postaci obiektu klasy `Rectangle`.	1.0
`Cursor getCursor()`	Zwraca kursor związany z komponentem.	1.1
`Font getFont()`	Zwraca czcionkę związaną z komponentem.	1.0
`FontMetrics getFontMetrics(Font font)`	Zwraca obiekt opisujący właściwości czcionki związanej z komponentem.	1.0 (1.5)
`Color getForeground()`	Zwraca pierwszoplanowy kolor komponentu.	1.0
`Graphics getGraphics()`	Zwraca tzw. graficzny kontekst urządzenia, obiekt pozwalający na wykonywanie operacji graficznych na komponencie.	1.0 (1.2)
`int getHeight()`	Zwraca wysokość komponentu.	1.2
`String getName()`	Zwraca nazwę komponentu.	1.1
`Container getParent()`	Zwraca obiekt nadrzędny komponentu.	1.0
`Dimension getSize()`	Zwraca rozmiary komponentu w postaci obiektu klasy `Dimension`.	1.1
`int getWidth()`	Zwraca szerokość komponentu.	1.2
`int getX()`	Zwraca współrzędną `x` położenia komponentu.	1.2
`int getY()`	Zwraca współrzędną `y` położenia komponentu.	1.2
`void repaint()`	Odrysowuje cały obszar komponentu.	1.0
`void repaint(int x, int y, int width, int height)`	Odrysowuje wskazany obszar komponentu.	1.0
`void setBackground(Color c)`	Ustala kolor tła komponentu.	1.0 (1.2)
`void setBounds(int x, int y, int width, int height)`	Ustala rozmiar i położenie komponentu.	1.1
`void setBounds(Rectangle r)`	Ustala rozmiar i położenie komponentu.	1.1
`void setCursor(Cursor cursor)`	Ustawia rodzaj kursora przypisany komponentowi.	1.1
`void setFont(Font f)`	Ustawia czcionkę przypisaną komponentowi.	1.0 (1.2)
`void setLocation(int x, int y)`	Zmienia położenie komponentu.	1.1

Tabela 8.7. *Wybrane metody klasy Component — ciąg dalszy*

Deklaracja metody	Opis	Od JDK
`void setLocation(Point p)`	Zmienia położenie komponentu.	1.1
`void setName(String name)`	Ustala nazwę komponentu.	1.1
`void setSize(Dimension d)`	Ustala rozmiary komponentu.	1.1
`void setSize(int width, int height)`	Ustala rozmiary komponentu.	1.1
`void setVisible(boolean b)`	Wyświetla lub ukrywa komponent.	1.0 (1.2)

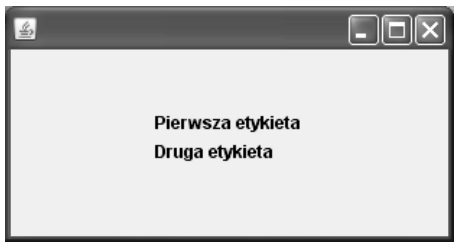
Etykiety

Etykiety tekstowe to jedne z najprostszych komponentów graficznych. Umożliwiają one wyświetlanie tekstu. Aby utworzyć etykietę, należy skorzystać z klasy `JLabel`. Konstruktor klasy `JLabel` może być bezargumentowy, tworzy wtedy pustą etykietę, lub przyjmować jako argument tekst, który ma być na niej wyświetlany. Po utworzeniu etykiety znajdujący się na niej tekst można pobrać za pomocą metody `getText`, jeśli natomiast chcemy go zmienić, korzystamy z metody `setText`. Etykietę umieszczamy w oknie lub na innym komponencie, wywołując metodę `add`. Prosty przykład obrazujący wykorzystanie klasy `JLabel` jest widoczny na listingu 8.37, natomiast efekt jego działania na rysunku 8.24.

Listing 8.37.

```
import javax.swing.*;

public class Aplikacja extends JFrame {
  public Aplikacja() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(null);

    JLabel label1 = new JLabel("Pierwsza etykieta");
    label1.setBounds(100, 40, 120, 20);

    JLabel label2 = new JLabel();
    label2.setText("Druga etykieta");
    label2.setBounds(100, 60, 120, 20);

    add(label1);
    add(label2);

    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Rysunek 8.24.
Wygląd przykładowych etykiet tekstowych



Aplikacja wykorzystuje oba wymienione wyżej sposoby tworzenia etykiet. W pierwszym przypadku (obiekt `label1`) już w konstruktorze przekazywany jest tekst, jaki ma być wyświetlany, w przypadku drugim (obiekt `label2`) tworzona etykieta jest pusta, a tekst jest jej przypisywany za pomocą metody `setText`. Rozmiary oraz położenie etykiet są ustalane dzięki metodzie `setBounds`. Pierwsze dwa argumenty tej metody określają współrzędne `x` i `y`, natomiast dwa kolejne szerokość oraz wysokość. Etykiety są dodawane do okna aplikacji za pomocą instrukcji:

```
add(label1);
add(label2);
```

Metoda `setLayout` ustala sposób rozkładu elementów w oknie. Przekazany jej argument `null` oznacza, że rezygnujemy z automatycznego rozmieszczania elementów i będziemy je pozycjonować ręcznie[10]. W tym przypadku oznacza to, że argumenty metody `setBounds` będą traktowane jako bezwzględne współrzędne okna aplikacji.

Przyciski

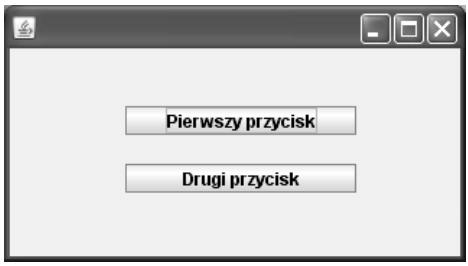
Obsługą i wyświetlaniem przycisków zajmuje się klasa `JButton`. Podobnie jak w przypadku klasy `JLabel`, konstruktor może być bezargumentowy, powstaje wtedy przycisk bez napisu na jego powierzchni, jak również może przyjmować argument klasy `String`. W tym drugim przypadku przekazany napis pojawi się na przycisku. Jeśli zastosujemy konstruktor bezargumentowy, będzie możliwe późniejsze przypisanie tekstu przyciskowi za pomocą metody `setText`. W odróżnieniu od etykiet, przyciski powinny jednak reagować na kliknięcia myszą, przy ich stosowaniu niezbędne będzie zatem zaimplementowanie interfejsu `ActionListener`. Przykładowa aplikacja zawierająca dwa przyciski, taka że po kliknięciu drugiego z nich nastąpi jej zamknięcie, jest widoczna na listingu 8.38, natomiast ich wygląd obrazuje rysunek 8.25.

Listing 8.38.

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

public class Aplikacja extends JFrame {
  private JButton button1, button2;
  public Aplikacja() {
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
```

[10] Czytelnicy zainteresowani rozkładami automatycznymi powinni zapoznać się z opisem klasy `LayoutManager` oraz klas pochodnych, a także z opisem metody `setLayout` klasy `Container`.

```
            if(e.getSource() == button2)
              dispose();
          }
        };

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);

        button1 = new JButton("Pierwszy przycisk");
        button1.setBounds(100, 40, 160, 20);
        button1.addActionListener(al);

        button2 = new JButton();
        button2.setText("Drugi przycisk");
        button2.setBounds(100, 80, 160, 20);
        button2.addActionListener(al);

        add(button1);
        add(button2);

        setSize(320, 200);
        setVisible(true);
      }
      public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {
          public void run() {
            new Aplikacja();
          }
        });
      }
    }
```



Rysunek 8.25. *Aplikacja zawierająca dwa przyciski*

Pierwszy przycisk jest tworzony za pomocą konstruktora przyjmującego w argumencie obiekt klasy `String`, czyli po prostu ciąg znaków. W drugim przypadku do ustawiania napisu wyświetlanego na przycisku wykorzystujemy metodę `setText`. Do ustalenia położenia i rozmiarów przycisków wykorzystujemy natomiast metodę `setBounds`, której działanie jest identyczne, jak w przypadku przedstawionych wcześniej etykiet. Ponieważ przyciski muszą reagować na kliknięcia, tworzymy nowy obiekt (`al`) anonimowej klasy implementującej interfejs `ActionListener`. W metodzie `actionPerformed` sprawdzamy, czy źródłem zdarzenia był drugi przycisk (`if(e.getSource() == button2)`). Jeśli tak, zamykamy okno i kończymy działanie aplikacji (wywołanie metody `dispose`).

Pola tekstowe

Pola tekstowe występują w kilku rodzajach: `JTextField`, `JPasswordField`, `JFormattedTextField`, `JTextArea`, `JEditorPane` i `JTextPane`. Wszystkie te klasy dziedziczą bezpośrednio bądź pośrednio z `JTextComponent`, która z kolei dziedziczy z `JComponent`. Nie mamy niestety miejsca na dokładne omówienie wszystkich tych komponentów, przedstawione zostaną więc jedynie przykłady użycia dwóch podstawowych: `JTextField` oraz `JTextArea`. Pierwszy z nich pozwala na wprowadzenie tekstu w jednej linii, drugi — tekstu wielowierszowego. Ich obsługa, jak zobaczymy za chwilę, jest podobna.

Klasa JTextField

Klasa `JTextField` tworzy jednowierszowe pole tekstowe, takie jak zaprezentowane na rysunku 8.26. Oferuje nam ona pięć konstruktorów przedstawionych w tabeli 8.8. Przykład wykorzystujący pole tekstowe jest widoczny na listingu 8.39.

Tabela 8.8. *Konstruktory klasy JTextField*

Konstruktor	Opis
`JTextField()`	Tworzy nowe, puste pole tekstowe.
`JTextField(Document doc, String text, int columns)`	Tworzy nowe pole tekstowe o zadanej liczbie kolumn i zawartości, zgodne z modelem dokumentu wskazanym przez argument `doc`.
`JTextField(int columns)`	Tworzy nowe pole tekstowe o zadanej liczbie kolumn.
`JTextField(String text)`	Tworzy nowe pole tekstowe zawierające wskazany tekst.
`JTextField(String text, int columns)`	Tworzy nowe pole tekstowe o zadanej liczbie kolumn zawierające wskazany tekst.

Listing 8.39.

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

public class Aplikacja extends JFrame {
  private JTextField textField;
  private JButton button1;
  public Aplikacja() {
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(e.getSource() == button1)
          setTitle(textField.getText());
      }
    };

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(null);

    textField = new JTextField();
    textField.setBounds(100, 50, 100, 20);

    button1 = new JButton("Kliknij!");
    button1.setBounds(100, 80, 100, 20);
```

```
    button1.addActionListener(al);

    add(button1);
    add(textField);

    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Rysunek 8.26. *Tekst z pola tekstowego staje się tytułem okna aplikacji*

Pole tekstowe tworzymy za pomocą instrukcji, których znaczenie nie powinno budzić żadnych wątpliwości. Jego szerokość ustawiamy na `100` pikseli, a wysokość na `20`. Do dyspozycji mamy również przycisk, którego kliknięcie będzie powodowało, że tekst znajdujący się w polu stanie się tytułem okna aplikacji. Do obsługi zdarzeń wykorzystujemy interfejs `ActionListener` i metodę `actionPerformed`, podobnie jak miało to miejsce we wcześniejszych przykładach. Tekst zapisany w polu tekstowym odczytujemy za pomocą metody `getText`, natomiast tytuł okna aplikacji zmieniamy za pomocą metody `setTitle`.

Jeśli chcemy mieć możliwość reagowania na każdą zmianę tekstu, jaka zachodzi w polu tekstowym `JTextField`, sytuacja nieco się komplikuje. Otóż należy monitorować zmiany dokumentu (obiektu klasy `Document`) powiązanego z komponentem `JTextField`. Trzeba zatem skorzystać z interfejsu `DocumentListener`, tworząc nowy obiekt implementujący ten interfejs, i przekazać go jako argument metody `setDocumentListener` wywołanej na rzecz obiektu zwróconego przez wywołanie `getDocument` klasy `JTextField`. Zakładając więc, że obiektem implementującym interfejs `DocumentListener` jest `dl`, a pole tekstowe reprezentuje obiekt `textField`, wywołanie powinno mieć postać:

```
textField.getDocument().addDocumentListener(dl)
```

W interfejsie `DocumentListener` zdefiniowane zostały natomiast trzy metody:

- `changedUpdate` — wywoływana po zmianie atrybutu lub atrybutów;
- `insertUpdate` — wywoływana przy wstawianiu treści do dokumentu;
- `removeUpdate` — wywoływana przy usuwaniu treści z dokumentu.

Jeśli zatem mielibyśmy napisać aplikację zawierającą pole tekstowe, taką, że każda zmiana w tym polu powodowałaby zmianę napisu znajdującego się na pasku tytułowym okna, możemy wykorzystać kod przedstawiony na listingu 8.40.

Listing 8.40.

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

public class Aplikacja extends JFrame {
  private JTextField textField;
  public Aplikacja() {
    DocumentListener dl = new DocumentListener() {
      public void changedUpdate(DocumentEvent e) {
        setTitle(textField.getText());
      }
      public void insertUpdate(DocumentEvent e) {
        setTitle(textField.getText());
      }
      public void removeUpdate(DocumentEvent e) {
        setTitle(textField.getText());
      }
    };

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(null);

    textField = new JTextField();
    textField.setBounds(100, 50, 100, 20);
    textField.getDocument().addDocumentListener(dl);

    add(textField);

    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Klasa JTextArea

Klasa `JTextArea` pozwala na tworzenie komponentów umożliwiających wprowadzenie większej ilości tekstu. Oferuje ona pięć konstruktorów przedstawionych w tabeli 8.9. Przykład wykorzystania obiektu tej klasy jest natomiast widoczny na listingu 8.41.

Tabela 8.9. *Konstruktory klasy JTextArea*

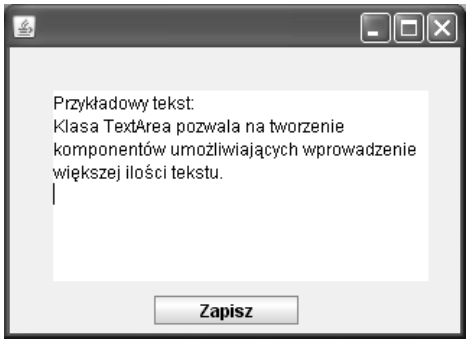
Konstruktor	Opis
`JTextArea()`	Tworzy pusty komponent `JTextArea`.
`JTextArea(Document doc)`	Tworzy nowe pole tekstowe zgodne z modelem dokumentu wskazanym przez argument `doc`.
`JTextArea(Document doc, String text, int rows, int columns)`	Tworzy nowe pole tekstowe zgodne z modelem dokumentu wskazanym przez argument `doc`, o zadanej zawartości oraz liczbie wierszy i kolumn.
`JTextArea(int rows, int columns)`	Tworzy pusty komponent `JTextArea` o określonej liczbie wierszy i kolumn.
`JTextArea(String text)`	Tworzy komponent `JTextArea` zawierający tekst określony przez argument `text`.
`JTextArea(String text, int rows, int columns)`	Tworzy komponent `JTextArea` zawierający tekst określony przez argument `text`, o liczbie wierszy i kolumn określonej argumentami `rows` i `columns`.

Listing 8.41.

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

public class Aplikacja extends JFrame {
  private JTextArea textArea;
  private JButton button1;
  public Aplikacja() {
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(e.getSource() == button1)
          save();
      }
    };
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(null);

    textArea = new JTextArea();
    textArea.setBounds(30, 30, 260, 200);
    this.add(textArea);

    button1 = new JButton("Zapisz");
    button1.setBounds(100, 240, 100, 20);
    button1.addActionListener(al);

    add(textArea);
    add(button1);

    setSize(320, 300);
    setVisible(true);
  }
  public void save() {
    FileWriter fileWriter = null;
    String text = textArea.getText();
```

```
        try{
          fileWriter = new FileWriter("test.txt", true);
          fileWriter.write(text, 0, text.length());
          fileWriter.close();
        }
        catch(IOException e){
          //System.out.println("Błąd podczas zapisu pliku.");
        }
      }
      public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {
          public void run() {
            new Aplikacja();
          }
        });
      }
    }
```

Tworzymy jeden komponent klasy `JTextArea` (obiekt `textArea`) oraz jeden przycisk, tak jak jest to widoczne na rysunku 8.27. Aplikacja działa w taki sposób, że po kliknięciu przycisku cały tekst z pola tekstowego zostanie zapisany w pliku o nazwie *test.txt*. Obsługę zdarzeń związanych z klikaniem przycisku zapewnia nam obiekt klasy anonimowej implementującej interfejs `ActionListener`, dokładnie tak samo jak w przypadku wcześniej prezentowanych przykładów. Po kliknięciu przycisku jest zatem wywoływana metoda `actionPerformed`, która sprawdza, czy sprawcą zdarzenia faktycznie jest przycisk `button1`. Jeśli tak, wywoływana jest metoda `save` klasy `TextArea`. Metoda `save` korzysta z obiektu klasy `FileWriter` do zapisania zawartości komponentu `textArea` w pliku (por. lekcja 34.).



Rysunek 8.27. *Aplikacja wykorzystująca pole tekstowe klasy TextArea*

Pola JCheckBox

Klasa `JCheckBox` tworzy komponenty będące polami wyboru umożliwiającymi zaznaczanie opcji. Konstruktory klasy zostały przedstawione w tabeli 8.10, a przykład wyświetlający sześć pól typu `JCheckBox` na listingu 8.42. Wygląd okna takiej aplikacji jest natomiast widoczny na rysunku 8.28.

Tabela 8.10. *Konstruktory klasy JCheckBox*

Konstruktor	Opis
`JCheckBox()`	Tworzy niezaznaczone pole wyboru, bez ikony i przypisanego tekstu.
`JCheckBox(Action a)`	Tworzy nowe pole wyboru o właściwościach wskazanych przez argument `a`.
`JCheckBox(Icon icon)`	Tworzy niezaznaczone pole wyboru z przypisaną ikoną.
`JCheckBox(Icon icon, boolean selected)`	Tworzy pole wyboru z ikoną, jego stan jest określony przez argument `selected`.
`JCheckBox(String text)`	Tworzy niezaznaczone pole wyboru z przypisanym tekstem.
`JCheckBox(String text, boolean selected)`	Tworzy pole wyboru z przypisanym tekstem, którego stan jest określony przez argument `selected`.
`JCheckBox(String text, Icon icon)`	Tworzy niezaznaczone pole wyboru z przypisaną ikoną oraz tekstem.
`JCheckBox(String text, Icon icon, boolean selected)`	Tworzy pole wyboru z przypisaną ikoną i tekstem, którego stan jest określony przez argument `selected`.

Listing 8.42.

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

public class Aplikacja extends JFrame {
  public Aplikacja() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new GridLayout(3, 2));

    JCheckBox cb1 = new JCheckBox("Opcja 1");
    JCheckBox cb2 = new JCheckBox("Opcja 2");
    JCheckBox cb3 = new JCheckBox("Opcja 3");
    JCheckBox cb4 = new JCheckBox("Opcja 4");
    JCheckBox cb5 = new JCheckBox("Opcja 5");
    JCheckBox cb6 = new JCheckBox("Opcja 6");

    add(cb1);
    add(cb2);
    add(cb3);
    add(cb4);
    add(cb5);
    add(cb6);

    setSize(320, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
      }
    });
  }
}
```

Rysunek 8.28. *Pola JCheckBox umieszczone w oknie aplikacji korzystającej z rozkładu siatkowego*

Zauważymy zapewne od razu pewną różnicę w stosunku do poprzednich przykładów: tym razem nie podajemy bowiem żadnych współrzędnych, w których mają znaleźć się poszczególne elementy okna. Zamiast tego zastosowaliśmy jeden z rozkładów automatycznych, tzw. rozkład tabelaryczny lub siatkowy (ang. *grid layout*). Za jego ustawienie odpowiada linia:

```
setLayout(new GridLayout(3, 2));
```

Parametrem metody `setLayout` jest nowy obiekt klasy `GridLayout`. Dzięki temu powierzchnia okna aplikacji zostanie podzielona na sześć części, tabelę o dwóch wierszach i trzech kolumnach. Komponenty, które będą dodawane do okna aplikacji, będą od tej chwili automatycznie umieszczane w kolejnych komórkach takiej tabeli, dzięki czemu uzyskamy ich równomierny rozkład.

Listy rozwijane

Tworzenie list rozwijalnych umożliwia klasa o nazwie `JComboBox`. Jej konstruktory zostały przedstawione w tabeli 8.11, a wybrane metody w tabeli 8.12. Oczywiście w rzeczywistości metod jest o wiele więcej, jednak zaprezentowane poniżej umożliwiają wykonywanie podstawowych operacji. Przykład wykorzystania listy obrazuje kod z listingu 8.43.

Tabela 8.11. *Konstruktory klasy JComboBox*

Konstruktor	Opis
`JComboBox()`	Tworzy pustą listę.
`JComboBox(ComboBoxModel aModel)`	Tworzy nową listę z modelem danych wskazanym przez argument `aModel`.
`JComboBox(Object[] items)`	Tworzy listę zawierającą dane znajdujące się w tablicy `items`.
`JComboBox(Vector<?> items)`	Tworzy listę zawierającą dane znajdujące się w wektorze `items`.

Listing 8.43.

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

public class Aplikacja extends JFrame {
  JComboBox cmb;
  JLabel label;
  public Aplikacja() {
```

Tabela 8.12. *Wybrane metody klasy JComboBox*

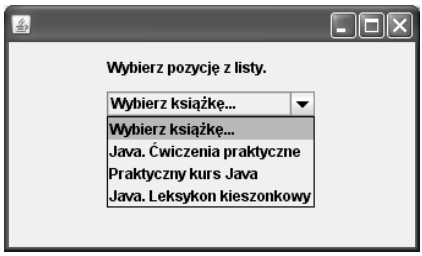
Deklaracja metody	Opis
`void addItem(Object anObject)`	Dodaje nową pozycję do listy.
`Object getItemAt(int index)`	Pobiera element listy znajdujący się pod wskazanym indeksem.
`int getItemCount()`	Pobiera liczbę elementów listy.
`int getSelectedIndex()`	Pobiera indeks zaznaczonego elementu.
`Object getSelectedItem()`	Pobiera zaznaczony element.
`void insertItemAt(Object anObject, int index)`	Wstawia nowy element we wskazanej pozycji listy.
`boolean isEditable()`	Zwraca `true`, jeśli istnieje możliwość edycji listy.
`void removeAllItems()`	Usuwa wszystkie elementy listy.
`void removeItem(Object anObject)`	Usuwa wskazany element listy.
`void removeItemAt(int anIndex)`	Usuwa element listy znajdujący się pod wskazanym indeksem.
`void setEditable(boolean aFlag)`	Ustala, czy lista ma mieć możliwość edycji.
`void setSelectedIndex(int anIndex)`	Zaznacza element listy znajdujący się pod wskazanym indeksem.
`void setSelectedItem(Object anObject)`	Zaznacza wskazany element listy.

```
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int itemIndex = cmb.getSelectedIndex();
        if(itemIndex < 1) return;
        String itemText = cmb.getSelectedItem().toString();
        label.setText(itemText);
      }
    };

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(null);

    label = new JLabel("Wybierz pozycję z listy.");
    label.setBounds(80, 10, 170, 20);

    cmb = new JComboBox();
    cmb.setBounds (80, 40, 170, 20);
    cmb.addItem("Wybierz książkę...");
    cmb.addItem("Java. Ćwiczenia praktyczne");
    cmb.addItem("Praktyczny kurs Java");
    cmb.addItem("Java. Leksykon kieszonkowy");
    cmb.addActionListener(al);

    add(cmb);
    add(label);

    setSize(340, 200);
    setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new Aplikacja();
```

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

Tworzymy tu aplikację zawierającą jedną listę rozwijalną oraz jedną etykietę, tak jak jest to widoczne na rysunku 8.29. Działanie programu jest następujące: po wybraniu nowej pozycji z listy przypisany jej tekst pojawi się na etykiecie.

Rysunek 8.29. *Wygląd aplikacji z listingu*



Sposób utworzenia listy oraz etykiety nie powinien budzić żadnych wątpliwości, odbywa się to analogicznie jak we wcześniejszych przykładach z bieżącej lekcji. Elementy listy są natomiast dodawane za pomocą metody `addItem` klasy `JComboBox`. Zdarzenia nadchodzące z listy obsługuje obiekt klasy anonimowej implementującej interfejs `ActionListener` o nazwie `al`. Jest w nim oczywiście zdefiniowana tylko jedna metoda — `actionPerformed`. Za pomocą wywołania `cmb.getSelectedIndex()` pobiera ona najpierw indeks zaznaczonego elementu listy. Jeśli okaże się, że indeks ten jest mniejszy od 1 (co by oznaczało, że albo nie został zaznaczony żaden z elementów, albo też zaznaczony jest element o indeksie 0), metoda kończy działanie, wywołując instrukcję `return`. Jeśli indeks jest większy od 0, pobierany jest tekst przypisany zaznaczonemu elementowi listy:

```
String itemText = cmb.getSelectedItem().toString();
```

który następnie jest wykorzystywany jako argument metody `setText` obiektu etykiety:

```
label.setText(itemText);
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 42.1.

Napisz aplikację zawierającą pole tekstowe, etykietę i przycisk. Po kliknięciu przycisku zawartość pola tekstowego powinna się znaleźć na etykiecie.

Ćwiczenie 42.2.

Zmodyfikuj kod z listingu 8.38 tak, aby zamknięcie aplikacji następowało jedynie po kliknięciu przycisków w kolejności: *Pierwszy przycisk*, *Drugi przycisk*.

Ćwiczenie 42.3.

Do aplikacji z listingu 8.43 dodaj przycisk. Po jego kliknięciu powinny zostać usunięte wszystkie elementy listy.