WYKORZYSTAJ POTENCJAŁ LIDERA
NA RYNKU JĘZYKÓW PROGRAMOWANIA!

PRENTICE HALL

JAVA™

Podstawy

WYDANIE IX

CAY S. HORSTMANN · GARY CORNELL

Helion

Tytuł oryginału: Core Java Volume I--Fundamentals (9th Edition)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-7758-0

Authorized translation from the English language edition, entitled CORE JAVA VOLUME I – FUNDAMENTALS, 9TH EDITION; ISBN 0137081898; by Cay S. Horstmann; and Gary Cornell; published by Pearson Education, Inc, publishing as Prentice Hall.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javpd9.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javpd9
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

7

Grafika

W tym rozdziale:

- Wprowadzenie do biblioteki Swing
- Tworzenie ramek
- Pozycjonowanie ramek
- Wyświetlanie informacji w komponencie
- Figury 2D
- Kolory
- Kroje czcionek
- Wyświetlanie obrazów

Do tej pory tworzyliśmy tylko programy, które pobierały dane z klawiatury, przetwarzały je i wyświetlały wyniki w konsoli. Większość użytkowników oczekuje jednak nieco więcej. Ani nowoczesne programy, ani strony internetowe nie działają w ten sposób. W tym rozdziale zaczynamy naukę pisania programów z graficznym interfejsem użytkownika (GUI). Nauczymy się przede wszystkim ustawiać rozmiar i położenie okien na ekranie, wyświetlać tekst pisany różnymi krojami czcionki, wyświetlać obrazy itd. Cały zdobyty tu warsztat przyda nam się w kolejnych rozdziałach, w których będziemy tworzyć ciekawe projekty.

Dwa następne rozdziały opisują przetwarzanie zdarzeń, takie jak wciśnięcie klawisza na klawiaturze lub kliknięcie przyciskiem myszy, oraz dodawanie do aplikacji takich elementów interfejsu jak menu i przyciski. Po zapoznaniu się z tymi trzema rozdziałami będziesz dysponować wiedzą, która jest niezbędna do pisania aplikacji graficznych. Bardziej zaawansowane techniki związane z programowaniem grafiki zostały opisane w drugim tomie.

Osoby zainteresowane wyłącznie programowaniem po stronie serwera, które nie mają w planach pisania GUI, mogą bezpiecznie pominąć te rozdziały.

7.1. Wprowadzenie do pakietu Swing

W Java 1.0 udostępniono bibliotekę klas o nazwie Abstract Window Toolkit (AWT), która dostarczała podstawowe narzędzia do programowania GUI. Podstawowa biblioteka AWT przerzuca zadania dotyczące tworzenia elementów interfejsu oraz obsługi ich zachowań na natywne narzędzia GUI danej platformy (Windows, Solaris, Mac OS X itd.). Jeśli na przykład przy użyciu oryginalnej biblioteki AWT umieszczono w oknie pole tekstowe, dane wprowadzane do niego były w rzeczywistości obsługiwane przez odpowiednik tego pola w systemie. Dzięki temu program napisany w Javie mógł teoretycznie działać na dowolnej platformie, a jego styl był taki sam jak innych programów w danym systemie. Stąd wziął się slogan firmy Sun: „Napisz raz, uruchamiaj wszędzie".

Metoda programowania oparta na odpowiednikach sprawdzała się w przypadku prostych aplikacji. Natomiast szybko wyszło na jaw, że napisanie wysokiej jakości przenośnej biblioteki graficznej opartej na natywnych elementach interfejsu użytkownika jest niezwykle trudnym zadaniem. Elementy interfejsu, jak menu, paski przewijania i pola tekstowe, mogą się nieco różnić na różnych platformach. Przez to trudno było stworzyć program działający identycznie w różnych systemach. Ponadto niektóre środowiska graficzne (jak np. X11/Motif) nie dysponują tak szeroką gamą elementów interfejsu jak systemy Windows czy Mac OS X. Ten fakt ograniczał zasobność przenośnej biblioteki do tych elementów, które można znaleźć na wszystkich platformach. W wyniku tego aplikacje GUI budowane na bazie AWT ustępowały wyglądem i funkcjonalnością natywnym aplikacjom takich systemów jak Windows czy Mac OS X. Na domiar złego na różnych platformach biblioteka AWT zawierała różne błędy. Programiści narzekali, że muszą testować swoje aplikacje na wszystkich platformach, co złośliwie nazywano „napisz raz, testuj wszędzie".

W 1996 roku firma Netscape stworzyła bibliotekę GUI o nazwie IFC (ang. *Internet Foundation Classes*), w której zastosowano zupełnie inne podejście. Elementy interfejsu użytkownika, jak przyciski, menu itd., **były rysowane** w pustym oknie. Rola systemu polegała tylko na wyświetlaniu okien i rysowaniu w nich. Dzięki temu widgety firmy Netscape wyglądały i działały zawsze tak samo, bez względu na platformę. Firma Sun podjęła współpracę z Netscape w celu udoskonalenia tej technologii, czego owocem była biblioteka o nazwie kodowej Swing. Biblioteka ta została udostępniona w postaci dodatku w Java 1.1, a do biblioteki standardowej wcielono ją w Java SE 1.2.

Zgodnie z myślą Duke'a Ellingtona, że „Nic nie ma znaczenia, jeśli jest pozbawione swingu", Swing stał się oficjalną nazwą zestawu narzędzi do tworzenia GUI, niewykorzystującego systemowych odpowiedników elementów. Swing wchodzi w skład Java Foundation Classes (JFC). Biblioteka JFC jest bardzo duża i zawiera wiele innych narzędzi poza Swingiem. Zaliczają się do nich interfejsy API dostępności, grafiki dwuwymiarowej oraz obsługi operacji przeciągania i upuszczania.

Oczywiście elementy interfejsu oparte na Swingu pojawiają się z pewnym opóźnieniem w stosunku do komponentów używanych przez AWT. Z naszego doświadczenia wynika, że różnica ta nie powinna stanowić problemu na żadnym w miarę niezbyt starym sprzęcie. Z drugiej strony argumenty przemawiające za użyciem Swinga są przytłaczające:

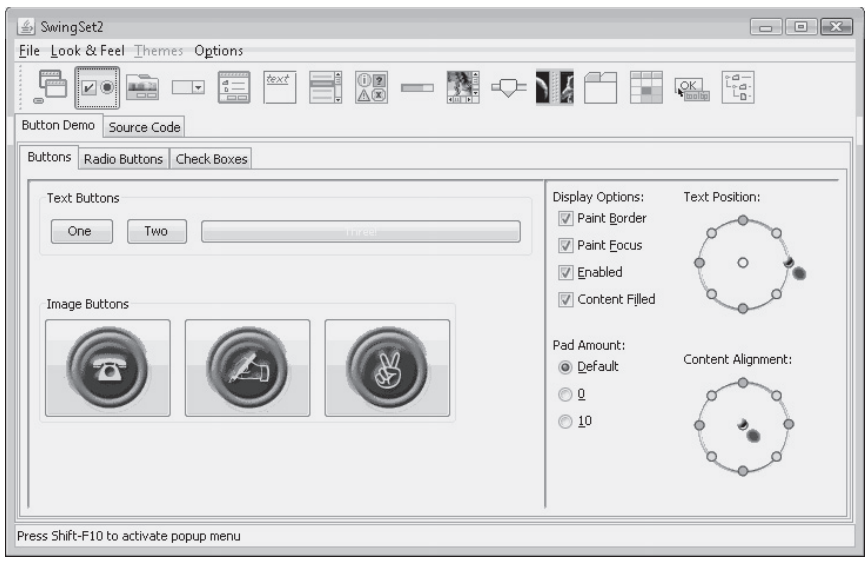
- Swing udostępnia bogaty i wygodny w użyciu zestaw elementów interfejsu użytkownika.

> Biblioteka Swing nie zastąpiła AWT, tylko została zbudowana w oparciu o architekturę swojej poprzedniczki. Komponenty Swing oferują znacznie większe możliwości. Używając biblioteki Swing, zawsze korzysta się z podstawowej biblioteki AWT, zwłaszcza przy obsłudze zdarzeń. Od tej pory pisząc „Swing", mamy na myśli klasy rysujące interfejs użytkownika, a pisząc „AWT", myślimy o podstawowych mechanizmach okien, takich jak obsługa zdarzeń.

- Swing w niewielkim stopniu zależy od platformy, dzięki czemu jest mniej podatny na błędy związane z danym systemem.
- Sposób działania i wygląd elementów Swinga jest taki sam na różnych platformach.

Niemniej trzecia z wymienionych zalet może być uważana za wadę — jeśli elementy interfejsu użytkownika wyglądają tak samo na wszystkich platformach, to znaczy, że wyglądają **inaczej** niż elementy natywne, co z kolei oznacza, że będą słabiej znane użytkownikom.

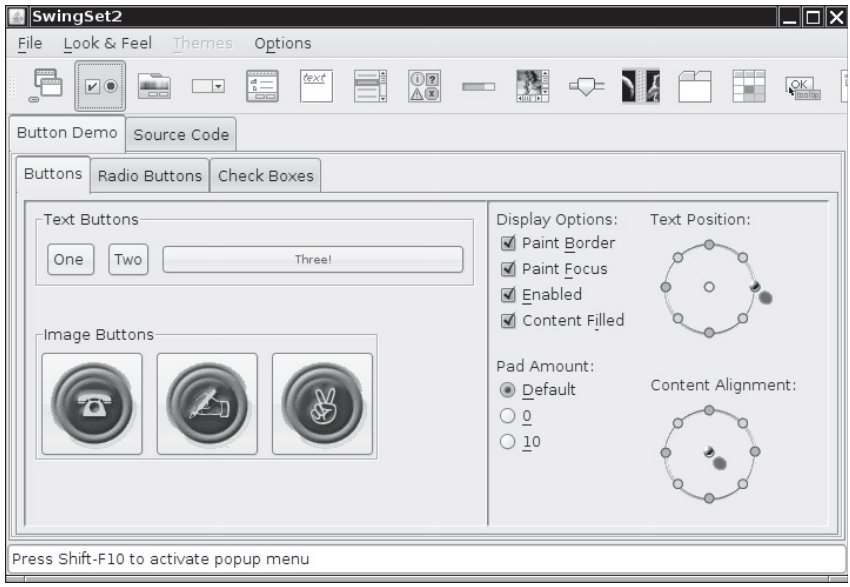
W pakiecie Swing problem ten został rozwiązany w bardzo elegancki sposób. Programista piszący program przy użyciu klas Swing może nadać mu specyficzny charakter. Rysunki 7.1 i 7.2 przedstawiają ten sam program w stylu systemu Windows i GTK.



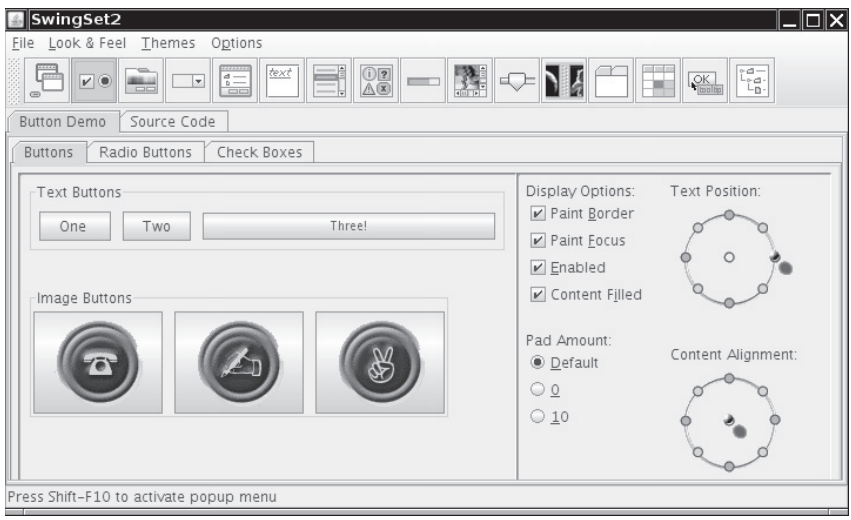
Rysunek 7.1. Styl systemu Windows

Dodatkowo firma Sun opracowała niezależny od platformy styl o nazwie **Metal**, który później przez specjalistów od marketingu przechrzczono na **Java look and feel**. Jednak większość programistów nadal używa określenia „Metal" i my również trzymamy się tej konwencji w tej książce.

Ze względu na krytyczne głosy kierowane pod adresem stylu Metal, który według niektórych wydawał się zbyt ciężki, w Java SE 5.0 nieco go odświeżono (zobacz rysunek 7.3). Obecnie styl Metal obsługuje wiele motywów — różnych wersji kolorystycznych i zestawów czcionek. Motyw domyślny nazywa się Ocean.



Rysunek 7.2. Styl GTK

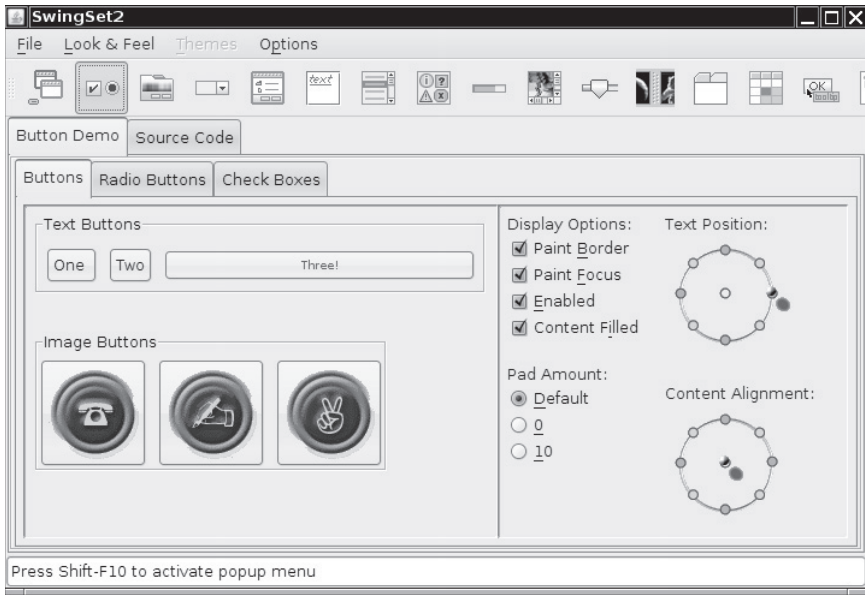


Rysunek 7.3. Motyw Ocean stylu Metal

W Java SE 6 poprawiono obsługę natywnego stylu systemu Windows i GTK. Aktualnie aplikacje Swing obsługują schematy kolorów i pulsujące przyciski oraz paski przewijania, które stały się ostatnio modne.

W Java 7 dodano nowy styl o nazwie Nimbus (rysunek 7.4), ale nie jest on domyślnie dostępny. W stylu tym wykorzystywana jest grafika wektorowa zamiast bitmapowej, dzięki czemu interfejsy tworzone przy jego użyciu są niezależne od rozmiaru ekranu.

Niektórzy użytkownicy wolą, aby aplikacje w Javie wyglądały tak jak inne programy na danej platformie, inni wolą styl Metal, a jeszcze inni preferują styl całkiem innego producenta. Jak przekonamy się w rozdziale 8., umożliwienie użytkownikom wyboru dowolnego stylu jest bardzo łatwe.



Rysunek 7.4. Styl Nimbus

W Javie możliwe jest rozszerzenie istniejącego stylu, a nawet utworzenie całkiem nowego. W tej książce nie mamy wystarczająco dużo miejsca, aby opisać ten żmudny proces polegający na określeniu sposobu rysowania każdego komponentu. Jednak niektórzy programiści pokusili się o to, zwłaszcza ci, którzy przenosili programy w Javie na nietypowe platformy, jak terminale sklepowe czy urządzenia kieszonkowe. Zbiór ciekawych stylów można znaleźć pod adresem *http://www.javootoo.com/*.

W Java SE 5.0 wprowadzono styl o nazwie Synth, który upraszcza cały ten proces. W Synth styl można definiować poprzez dostarczenie obrazów i deskryptorów XML. Nie trzeba nic programować.

Styl Napkin (*http://napkinlaf.sourceforge.net*) nadaje elementom interfejsu użytkownika wygląd przypominający odręczne rysowanie. Wysyłając do klienta utworzony w nim prototyp, dajemy wyraźny sygnał, że nie jest to jeszcze ukończony produkt.

Programowanie interfejsu użytkownika w Javie opiera się obecnie w większości na pakiecie Swing. Jest tylko jeden godny uwagi wyjątek. Środowisko zintegrowane Eclipse używa zestawu narzędzi graficznych o nazwie SWT, który podobnie jak AWT odwzorowuje natywne komponenty na różnych platformach. Artykuły na temat SWT można znaleźć na stronie *http://www.eclipse.org/articles/*.

Firma Oracle pracuje nad alternatywną technologią o nazwie JavaFX, która może w przyszłości zastąpić Swing. Jeśli chcesz dowiedzieć się o niej więcej, zajrzyj na stronę *http://www.oracle.com/technetwork/java/javafx/overview*.

Każdy, kto pisał programy dla systemu Microsoft Windows w językach Visual Basic lub C#, wie, jak dużym ułatwieniem są graficzne narzędzia do projektowania układu i edytory zasobów. Narzędzia te umożliwiają zaprojektowanie całej wizualnej strony aplikacji oraz automatycznie generują większość (często całość) kodu GUI. Dla Javy również dostępne są narzędzia

wspomagające budowanie GUI, ale naszym zdaniem, aby się nimi sprawnie posługiwać, trzeba najpierw nauczyć się robić to samodzielnie. Pozostała część tego rozdziału została poświęcona opisowi technik wyświetlania okien i rysowania w nich różnych elementów.

7.2. Tworzenie ramki

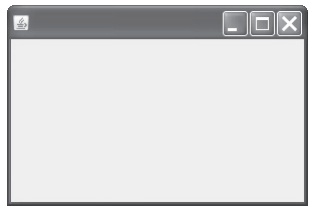
Okno najwyższego poziomu, tzn. takie, które nie jest zawarte w żadnym innym oknie, nazywa się w Javie **ramką** (ang. *frame*). W bibliotece AWT dla tego typu okien utworzono klasę o nazwie `Frame`. Wersja Swing tej klasy nosi nazwę `JFrame` i ją rozszerza. Ramka `JFrame` jest jednym z niewielu komponentów Swinga, które nie są rysowane w obszarze roboczym. W związku z tym elementy dodatkowe (przyciski, pasek tytułu, ikony itd.) są rysowane przez system operacyjny, a nie klasy Swing.

> Nazwy większości klas komponentów Swing zaczynają się od litery `J`, np. `JButton`, `JFrame` itd. Istnieją także klasy `Button` i `Frame`, ale są to komponenty AWT. Jeśli litera `J` zostanie przypadkowo pominięta, program może przejść kompilację bez problemu, ale mieszanina komponentów AWT i Swing może spowodować nietypowe zachowania lub wygląd aplikacji.

W tym podrozdziale przejrzymy najpopularniejsze metody pracy z komponentem Swing o nazwie `JFrame`. Listing 7.1 przedstawia prosty program wyświetlający na ekranie pustą ramkę widoczną na rysunku 7.5.

Listing 7.1. simpleFrame/SimpleFrameTest.java

```
package simpleFrame;

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

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



Rysunek 7.5. Najprostsza widoczna ramka

```
class SimpleFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public SimpleFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

Przeanalizujemy powyższy program wiersz po wierszu.

Klasy Swing znajdują się w pakiecie `javax.swing`. Nazwa pakietu `javax` oznacza, że jest to pakiet rozszerzający Javy, a nie podstawowy. Swing jest uznawany za rozszerzenie ze względów historycznych. Jest jednak dostępny w każdej wersji Java SE od 1.2.

Domyślny rozmiar ramki 0×0 jest raczej mało atrakcyjny. Zdefiniowaliśmy podklasę o nazwie `SimpleFrame`, której konstruktor ustawia rozmiar na 300×200 pikseli. Jest to jedyna różnica pomiędzy klasami `SimpleFrame` i `JFrame`.

W metodzie `main` klasy `SimpleFrameTest` tworzony jest obiekt klasy `SimpleFrame`, który następnie został uwidoczniony.

W każdym programie opartym na Swingu trzeba poradzić sobie z dwoma zagadnieniami technicznymi.

Po pierwsze, konfiguracja każdego komponentu Swing musi się odbywać w **wątku dystrybucji zdarzeń** (ang. *event dispatch thread*), który steruje wysyłaniem zdarzeń, jak kliknięcia przyciskiem myszy lub wciśnięcie klawisza na klawiaturze, do elementów interfejsu użytkownika. Poniższy fragment programu wykonuje instrukcje w wątku dystrybucji zdarzeń:

```
EventQueue.invokeLater(new Runnable()
   {
      public void run()
      {
         instrukcje
      }
   });
```

Szczegółowy opis tego zagadnienia znajduje się w rozdziale 14. Na razie potraktujmy to jako magiczny fragment kodu potrzebny do uruchomienia programu Swing.

> Wiele programów nie inicjuje interfejsu użytkownika w wątku dystrybucji zdarzeń. Nie ma żadnych przeszkód, aby operację tę przeprowadzać w wątku głównym. Niestety ze względu na to, że komponenty Swing stawały się coraz bardziej złożone, programiści w firmie Sun nie mogli zagwarantować bezpieczeństwa tej metody. Prawdopodobieństwo wystąpienia błędu jest niezwykle niskie, ale nikt nie chciałby być tym pechowcem, któremu się to przydarzy. Lepiej pisać właściwy kod, nawet jeśli wygląda nieco tajemniczo.

Po drugie, określamy, co ma się stać, kiedy użytkownik zamknie ramkę aplikacji. W tym przypadku chcemy, aby program został zamknięty. Odpowiedzialna jest za to poniższa instrukcja:

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

W programach składających się z wielu ramek program nie powinien kończyć działania w wyniku zamknięcia jednej z nich. Przy standardowych ustawieniach, jeśli użytkownik zamknie ramkę, zostanie ona ukryta, a program nie zakończy działania (dobrze by było, gdyby program był wyłączany w chwili zamknięcia jego **ostatniej** ramki, ale Swing działa inaczej).

Samo utworzenie ramki nie oznacza, że zostanie ona wyświetlona. Ramki na początku swojego istnienia są niewidoczne. Dzięki temu programista może dodać do nich wszystkie komponenty, zanim ukażą się po raz pierwszy. Aby wyświetlić ramkę, metoda `main` wywołuje na jej rzecz metodę `setVisible`.

> Przed Java SE 5.0 można było używać metody `show` dziedziczonej przez klasę `JFrame` po nadklasie `Window`. Nadklasą klasy `Window` jest `Component`, która także zawiera metodę `show`. Stosowanie metody `Component.show` zaczęto odradzać w Java SE 1.2. W zamian, aby wyświetlić komponent, należy użyć metody `setVisible(true)`. Natomiast metoda `Window.show` **nie była** odradzana aż do Java SE 1.4. Możliwość uwidocznienia okna i przeniesienia go na przód była nawet przydatna. Niestety metoda `show` dla okien również jest odradzana od Java SE 5.0.

Po rozplanowaniu instrukcji inicjujących metoda `main` kończy działanie. Zauważmy, że zakończenie metody `main` nie oznacza zamknięcia programu, a jedynie głównego wątku. Wątek dystrybucji zdarzeń podtrzymuje działanie programu aż do jego zakończenia poprzez zamknięcie ramki lub wywołanie metody `System.exit`.

Uruchomiony program przedstawia rysunek 7.5 — jest to zwykłe szare okno najwyższego poziomu. Jak widać, pasek tytułu i pozostałe dodatki, jak zaokrąglone rogi służące do zmiany rozmiaru okna, zostały narysowane przez system operacyjny, a nie klasy Swing. Elementy te będą wyglądały inaczej, jeśli uruchomimy ten program w systemach Windows, GTK czy Mac OS X. Biblioteka Swing rysuje wszystko, co znajduje się wewnątrz ramki. W tym przypadku jej zadanie sprowadza się jedynie do wypełnienia domyślnym kolorem tła.

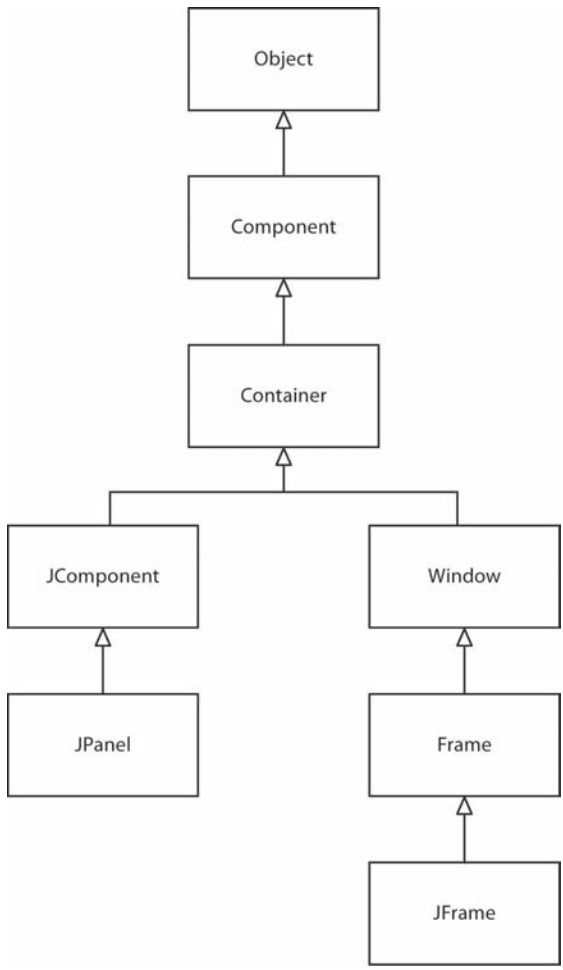
> Od Java SE 1.4 istnieje możliwość wyłączenia wszelkich dodatków za pomocą wywołania metody `frame.setUndecorated(true)`.

7.3. Pozycjonowanie ramki

Klasa `JFrame` udostępnia tylko kilka metod zmieniających wygląd ramek. Oczywiście większość metod służących do zmiany wymiarów i położenia ramki jest w klasie `JFrame` dziedziczona po różnych nadklasach. Poniżej znajduje się lista najważniejszych z tych metod:

- `setLocation` i `setBounds` — ustawiają położenie ramki.
- `setIconImage` — określa ikonę wyświetlaną w pasku tytułu, w zasobniku systemowym itd.
- `setTitle` — ustawia tekst w pasku tytułu.
- `setResizable` — pobiera wartość logiczną określającą, czy użytkownik może zmieniać rozmiar ramki.

Rysunek 7.6 przedstawia hierarchię dziedziczenia klasy `JFrame`.



Rysunek 7.6. Hierarchia dziedziczenia klas ramek i komponentów w pakietach AWT i Swing

W wyciągach z API do tego podrozdziału przedstawiamy te metody, które naszym zdaniem mają największe znaczenie przy nadawaniu ramkom odpowiedniego stylu. Niektóre z nich są udostępnione w klasie `JFrame`. Inne z kolei pochodzą od różnych nadklas klasy `JFrame`. Czasami konieczne może być przeszukanie dokumentacji API w celu znalezienia metod przeznaczonych do określonego celu. Niestety jest to dość żmudna praca, jeśli chodzi o metody dziedziczone. Na przykład metoda `toFront` ma zastosowanie do obiektów typu `JFrame`, ale ponieważ jest dziedziczona po klasie `Window`, w dokumentacji `JFrame` nie ma jej opisu. Jeśli uważasz, że powinna istnieć metoda wykonująca określone działanie, ale nie ma jej w dokumentacji danej klasy, przeszukaj dokumentację metod nadklas tej klasy. Na górze każdej strony w dokumentacji API znajdują się odnośniki do nadklas, a lista metod dziedziczonych znajduje się pod zestawieniem nowych i przesłoniętych metod.

Według dokumentacji API metody służące do zmieniania rozmiaru i kształtu ramek znajdują się w klasach `Component` (będącej przodkiem wszystkich obiektów GUI) i `Window` (będącej nadklasą klasy `Frame`). Na przykład do zmiany położenia komponentu można użyć metody `setLocation` z klasy `Component`. Wywołanie:

```
setLocation(x, y)
```

umieści lewy górny róg komponentu w odległości `x` pikseli od lewej krawędzi ekranu i `y` pikseli od góry. Wartości `(0, 0)` oznaczają lewy górny róg ekranu. Podobnie metoda `setBounds` w klasie `Component` umożliwia zmianę rozmiaru i położenia komponentu (zwłaszcza ramki `JFrame`) w jednym wywołaniu:

```
setBounds(x, y, width, height)
```

Istnieje też możliwość pozostawienia decyzji o położeniu okna systemowi. Wywołanie:

```
setLocationByPlatform(true);
```

przed wyświetleniem okna spowoduje, że system we własnym zakresie określi jego położenie (ale nie rozmiar). Zazwyczaj nowe okno jest wyświetlane z nieznacznym przesunięciem względem poprzedniego.

W przypadku ramki współrzędne metod `setLocation` i `setBounds` są względne do całego ekranu. W rozdziale 9. dowiemy się, że współrzędne komponentów znajdujących się w kontenerze odnoszą się do tego kontenera.

7.3.1. Własności ramek

Wiele metod klas komponentów występuje w parach `get-set`, jak poniższe metody klasy `Frame`:

```
public String getTitle()
public void setTitle(String title)
```

Taka para metod typu `get-set` nazywa się **własnością** (ang. *property*). Własność ma nazwę i typ. Nazwa jest taka sama jak słowo uzyskane w wyniku opuszczenia członu `get` i zamienienia pierwszej litery powstałego słowa na małą. Na przykład klasa `Frame` ma własność o nazwie `title` i typie `String`.

Z założenia `title` jest własnością ramki. Ustawienie (`set`) niniejszej własności powoduje zmianę tytułu na ekranie użytkownika. Pobranie jej (`get`) zwraca wartość, która została wcześniej ustawiona.

Nie wiemy (i nie obchodzi nas to), jak klasa `Frame` implementuje tę własność. Prawdopodobnie wykorzystuje swój odpowiednik ramki do przechowywania tytułu. Możliwe, że ma następujące pole:

```
private String title;    // nie jest wymagane dla własności
```

Jeśli klasa zawiera pasujące pole, nie wiemy (lub nie obchodzi nas to), jak metody dostępowe i ustawiające są zaimplementowane. Prawdopodobnie po prostu odczytują i ustawiają dane pole. Możliwe, że robią jeszcze coś, np. powiadamiają system o każdej zmianie tytułu.

Jest tylko jeden wyjątek od konwencji `get-set`: metody własności typu logicznego zaczynają się od przedrostka `is`. Na przykład dwie przedstawione poniżej metody definiują własność `locationByPlatform`:

```
public boolean isLocationByPlatform()
public void setLocationByPlatform(boolean b)
```

Znacznie więcej na temat własności piszemy w rozdziale 8. drugiego tomu.

> Wiele języków programowania, zwłaszcza Visual Basic i C#, standardowo obsługuje własności. Niewykluczone, że w przyszłości w Javie również pojawi się podobna konstrukcja językowa.

7.3.2. Określanie rozmiaru ramki

Pamiętajmy, że jeśli nie ustawimy własnego rozmiaru ramek, wszystkie będą miały wymiary 0×0 pikseli. Aby nie komplikować naszych przykładowych programów, ustawiamy ramki na rozmiar do przyjęcia na większości ekranów. Jednak w profesjonalnej aplikacji należy sprawdzić rozdzielczość ekranu użytkownika i napisać podprogram zmieniający rozmiar ramek w zależności od potrzeby. Na przykład okno, które wygląda znakomicie na ekranie laptopa, na ekranie o wysokiej rozdzielczości skurczy się do rozmiarów znaczka pocztowego.

Aby sprawdzić rozmiar ekranu, należy wykonać następujące działania: wywołaj statyczną metodę `getDefaultToolkit` klasy `Toolkit` w celu utworzenia obiektu typu `Toolkit` (klasa `Toolkit` jest zbiornikiem rozmaitych metod, które współpracują z systemem). Następnie wywołaj metodę `getScreenSize`, która zwraca rozmiar ekranu w postaci obiektu `Dimension`. Obiekt tego typu przechowuje wysokość i szerokość w zmiennych publicznych (!) o nazwach `width` i `height`. Poniżej przedstawiamy opisywany fragment programu:

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

Rozmiar ramki ustawiamy na połowę ekranu i pozwalamy systemowi na ustalenie położenia ramki:

```
setSize(screenWidth / 2, screenHeight / 2);
setLocationByPlatform(true);
```

Dodatkowo dostarczymy ikonę. Do wygodnego ładowania obrazów można wykorzystać klasę `ImageIcon`. Poniżej przedstawiony jest sposób jej użycia:

```
Image img = new ImageIcon("icon.gif").getImage();
setIconImage(img);
```

Ikona może się pojawić w różnych miejscach, w zależności od systemu operacyjnego. W systemie Windows pojawi się na przykład w lewym górnym rogu okna oraz będzie widoczna wśród aktywnych zadań pojawiających się w wyniku naciśnięcia kombinacji klawiszy *Alt+Tab*.

Listing 7.2 przedstawia pełny kod omawianego programu. Po jego uruchomieniu zwróć uwagę na ikonę *Core Java*.

Listing 7.2. sizedFrame/SizedFrameTest.java

```
package sizedFrame;

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

/**
 * @version 1.32 2007-04-14
 * @author Cay Horstmann
 */
public class SizedFrameTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new SizedFrame();
               frame.setTitle("SizedFrame");
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true););
            }
         });
   }
}

class SizedFrame extends JFrame
{
   public SizedFrame()
   {
      // Sprawdzenie wymiarów ekranu.

      Toolkit kit = Toolkit.getDefaultToolkit();
      Dimension screenSize = kit.getScreenSize();
      int screenHeight = screenSize.height;
      int screenWidth = screenSize.width;

      // Ustawienie szerokości i wysokości ramki oraz polecenie systemowi, aby ustalił jej położenie.
```

```
        setSize(screenWidth / 2, screenHeight / 2);
        setLocationByPlatform(true);

        // Ustawienie ikony i tytułu.

        Image img = new ImageIcon("icon.gif").getImage();
        setIconImage(img);
    }
}
```

Jeszcze kilka dodatkowych wskazówek dotyczących obsługi ramek:

- Jeśli ramka zawiera tylko standardowe komponenty, jak przyciski i pola tekstowe, jej rozmiar można ustawić za pomocą metody `pack`. Rozmiar zostanie ustawiony na najmniejszy, w którym zmieszczą się wszystkie komponenty. Często rozmiar głównej ramki jest ustawiany na największą wartość. Od Java SE 1.4 ramkę można zmaksymalizować za pomocą następującego wywołania:

  ```
  frame.setExtendedState(Frame.MAXIMIZED_BOTH);
  ```

- Dobrym pomysłem jest zapisanie położenia i rozmiaru ramki ustawionych przez użytkownika i zastosowanie tych ustawień przy ponownym uruchomieniu aplikacji. Jak to zrobić, dowiemy się w rozdziale 10., przy okazji omawiania API preferencji.
- Jeśli aplikacja wykorzystuje kilka ekranów, ich rozmiary można sprawdzić za pomocą metod `GraphicsEnvironment` i `GraphicsDevice`.
- Ponadto klasa `GraphicsDevice` umożliwia uruchomienie programu w trybie pełnoekranowym.

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

- `boolean isVisible()`
- `void setVisible(boolean b)`

  Sprawdza lub ustawia własność widoczności. Komponenty są początkowo widoczne z wyjątkiem komponentów najwyższego poziomu, jak `JFrame`.

- `void setSize(int width, int height)` **1.1**

  Ustawia szerokość i wysokość komponentu.

- `void setLocation(int x, int y)` **1.1**

  Przenosi komponent w inne miejsce. Współrzędne `x` i `y` są względne do kontenera lub ekranu, jeśli komponent jest komponentem najwyższego poziomu (np. `JFrame`).

- `void setBounds(int x, int y, int width, int height)` **1.1**

  Przesuwa komponent i zmienia jego rozmiar.

- `Dimension getSize()` **1.1**
- `void setSize(Dimension d)` **1.1**

  Pobiera lub ustawia własność `size` komponentu.

`java.awt.Window` **1.0**

- `void toFront()`

  Przenosi okno przed wszystkie pozostałe okna.

- `void toBack()`

  Przenosi okno na sam dół stosu okien i odpowiednio przestawia pozostałe widoczne okna.

- `boolean isLocationByPlatform()` **5.0**
- `void setLocationByPlatform(boolean b)` **5.0**

  Pobiera lub ustawia własność `locationByPlatform`. Jeśli zostanie ona ustawiona przed wyświetleniem okna, platforma wybierze odpowiednią lokalizację.

`java.awt.Frame` **1.0**

- `boolean isResizable()`
- `void setResizable(boolean b)`

  Pobiera lub ustawia własność `resizable`. Jeśli jest ona ustawiona, użytkownik może zmieniać rozmiar ramki.

- `String getTitle()`
- `void setTitle(String s)`

  Pobiera lub ustawia własność `title` określającą tekst na pasku tytułu.

- `Image getIconImage()`
- `void setIconImage(Image image)`

  Pobiera lub ustawia własność `iconImage`, która określa ikonę ramki. System może wyświetlić ikonę jako dodatek w ramce lub w innym miejscu.

- `boolean isUndecorated()` **1.4**
- `void setUndecorated(boolean b)` **1.4**

  Pobiera lub ustawia własność `undecorated`. Jeśli ta własność jest ustawiona, ramka nie zawiera żadnych dodatków, jak pasek tytułu czy przycisk zamykający. Ta metoda musi być wywołana przed wyświetleniem ramki.

- `int getExtendedState()` **1.4**
- `void setExtendedState(int state)` **1.4**

  Pobiera lub ustawia stan rozszerzonego okna. Możliwe stany to:

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

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

- `static Toolkit getDefaultToolkit()`

  Zwraca standardowy zestaw narzędzi.

- `Dimension getScreenSize()`

  Pobiera rozmiar ekranu.

`javax.swing.ImageIcon` **1.2**

- `ImageIcon(String filename)`

  Tworzy ikonę, której obraz jest przechowywany w pliku.

- `Image getImage()`

  Pobiera obraz ikony.

7.4. Wyświetlanie informacji w komponencie

W tym podrozdziale nauczymy się wyświetlać informacje w ramce. Zamiast przykładowego programu wyświetlającego tekst w konsoli, który widzieliśmy w rozdziale 3., zaprezentujemy program wyświetlający tekst w ramce, jak na rysunku 7.7.

Rysunek 7.7. Ramka wyświetlająca tekst

Łańcuch tekstowy można wydrukować bezpośrednio na ramce, ale jest to uznawane za zły zwyczaj programistyczny. Ramki w Javie są z założenia kontenerami do przechowywania komponentów, takich jak pasek menu i inne elementy interfejsu użytkownika. Z reguły rysowanie odbywa się na innym dodanym do ramki komponencie.

Struktura ramki `JFrame` jest zadziwiająco skomplikowana. Rysunek 7.8 przedstawia schemat budowy takiej ramki. Jak widać, ramka `JFrame` składa się z czterech warstw. Trzy z nich — podstawowa (ang. *root pane*), warstwowa (ang. *layered pane*) i przezroczysta (ang. *glass pane*) — nie są dla nas interesujące. Ich przeznaczeniem jest organizacja paska menu i warstwy z treścią oraz implementacja stylu. Część interesująca programistów Swing to **warstwa treści** (ang. *content pane*). Podczas projektowania ramki komponenty dodaje się do warstwy treści, stosując kod podobny do poniższego:

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```

Do Java SE 1.4 metoda `add` klasy `JFrame` powodowała wyjątek wyświetlający komunikat „Do not use JFrame.add(). Use JFrame.getContentPane().add() instead". W Java SE 5.0 zrezygnowano z takiej edukacji programistów, czego wyrazem jest zezwolenie na wywoływanie metody `JFrame.add` na rzecz warstwy z treścią.

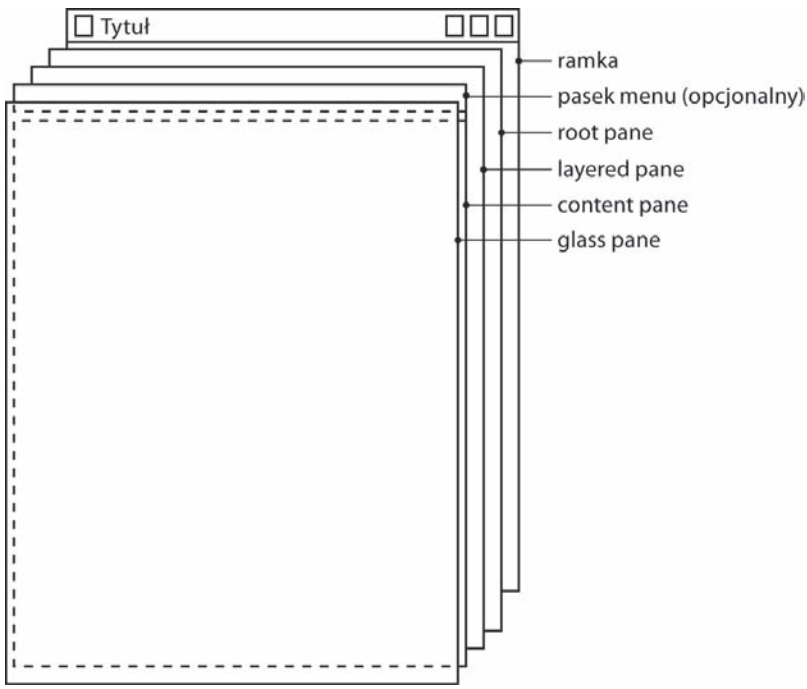
Dzięki temu od Java SE 5.0 można stosować krótki zapis:

```
frame.add(c);
```

Tym razem chcemy dodać do ramki jeden komponent, na którym narysujemy nasz komunikat. Aby móc rysować na komponencie, należy zdefiniować klasę rozszerzającą klasę `JComponent` i przesłonić w niej metodę `paintComponent` klasy nadrzędnej.

Metoda `paintComponent` przyjmuje jeden parametr typu `Graphics`. Obiekt typu `Graphics` zawiera ustawienia dotyczące rysowania obrazów i tekstu, jak czcionka czy aktualny kolor. Rysowanie w Javie zawsze odbywa się za pośrednictwem obiektu `Graphics`. Udostępnia on metody rysujące wzory, obrazy i tekst.

Rysunek 7.8. Wewnętrzna struktura ramki JFrame



> Parametr `Graphics` jest podobny do kontekstu urządzeń (ang. *device context*) w systemie Windows i kontekstu graficznego (ang. *graphics context*) w programowaniu dla systemu X11.

Poniższy fragment programu tworzy komponent, na którym można rysować:

```
class MyComponent extends JComponent
{
   public void paintComponent(Graphics g)
```

```
    {
       kod rysujący
    }
}
```

Za każdym razem, kiedy okno musi być ponownie narysowane, metoda obsługi zdarzeń informuje o tym komponent. Powoduje to uruchomienie metod `paintComponent` wszystkich komponentów.

Nigdy nie należy wywoływać metody `paintComponent` samodzielnie. Jest ona wywoływana automatycznie, gdy trzeba ponownie narysować jakąś część aplikacji, i nie należy zaburzać tego automatycznego procesu.

Jakiego rodzaju czynności uruchamiają tę automatyczną reakcję? Rysowanie jest konieczne, na przykład kiedy użytkownik zwiększy rozmiar okna albo je zminimalizuje, a następnie zmaksymalizuje. Jeśli zostanie otwarte nowe okno, które częściowo przykryje stare, to po zamknięciu tego nowego okna przykryta część starego zostanie zniszczona i trzeba ją będzie ponownie narysować (system graficzny nie zapisuje pikseli, które znajdują się pod spodem). Oczywiście przy pierwszym uruchomieniu okna musi ono przetworzyć kod określający sposób i miejsce rysowania początkowych elementów.

> Aby wymusić ponowne rysowanie ekranu, należy użyć metody `repaint`. Wywoła ona metody `paintComponent` wszystkich komponentów, które mają prawidłowo skonfigurowany obiekt `Graphics`.

Z przedstawionego powyżej fragmentu kodu wnioskujemy, że metoda `paintComponent` przyjmuje tylko jeden parametr typu `Graphics`. Jednostką miary obiektów `Graphics` wyświetlanych na ekranie są piksele. Para współrzędnych (0, 0) określa lewy górny róg komponentu, na którego powierzchni odbywa się rysowanie.

Wyświetlanie tekstu jest specjalnym rodzajem rysowania. Klasa `Graphics` udostępnia metodę `drawString` o następującej składni:

```
g.drawString(text, x, y)
```

Chcemy narysować łańcuch: *To nie jest program „Witaj, świecie”* w oryginalnym oknie w odległości około jednej czwartej szerokości od lewej krawędzi i połowy wysokości od krawędzi górnej. Mimo że nie potrafimy jeszcze mierzyć długości łańcuchów, zaczniemy rysowanie w punkcie o współrzędnych (75, 100). Oznacza to, że pierwsza litera łańcucha jest przesunięta w prawo o 75 pikseli i w dół o 100 pikseli (w rzeczywistości o 100 pikseli w dół przesunięta jest podstawowa linia pisma — więcej na ten temat w dalszej części rozdziału). Kod opisanej metody `paintComponent` znajduje się poniżej:

```
class NotHelloWorldComponent extends JComponent
{
   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;

   public void paintComponent(Graphics g)
   {
      g.drawString("To nie jest program „Witaj, świecie”", MESSAGE_X, MESSAGE_Y);
```

```
   }
   . . .
}
```

W końcu komponent powinien informować użytkownika o tym, jaki ma być jego rozmiar. Przesłonimy metodę `getPreferredSize`, aby zwracała obiekt klasy `Dimension` z preferowaną szerokością i wysokością:

```
class NotHelloWorldComponent extends JComponent
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
   . . .
   public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
DEFAULT_HEIGHT); }
}
```

Gdy umieścimy w ramce jakieś komponenty i będziemy chcieli użyć ich preferowanych rozmiarów, to zamiast `setSize` wywołamy metodę `pack`:

```
class NotHelloWorldFrame extends JFrame
{
   public NotHelloWorldFrame()
   {
      add(new NotHelloWorldComponent());
      pack();
   }
}
```

Listing 7.3 zawiera pełny kod programu.

Niektórzy programiści wolą rozszerzać klasę `JPanel` zamiast `JComponent`. Obiekt `JPanel` jest z założenia **kontenerem** na inne komponenty, ale można także na nim rysować. Jest jednak między nimi jedna różnica — panel jest **nieprzezroczysty**, czyli rysuje wszystkie piksele w swoim obrębie. Najprostszym sposobem na zrobienie tego jest narysowanie panelu z kolorowym tłem za pomocą wywołania metody `super.paintCompo➥nent` w metodzie `paintComponent` każdej podklasy panelu:

```
class NotHelloWorldPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      procedury rysujące
   }
}
```

Listing 7.3. notHelloWorld/NotHelloWorld.java

```
package notHelloWorld;

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

/**
 * @version 1.32 2007-06-12
```

```
 * @author Cay Horstmann
 */
public class NotHelloWorld
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new NotHelloWorldFrame();
               frame.setTitle("NotHelloWorld");
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * Ramka zawierająca panel z komunikatem.
 */
class NotHelloWorldFrame extends JFrame
{
   public NotHelloWorldFrame()
   {
      add(new NotHelloWorldComponent());
      pack();
   }
}

/**
 * Panel wyświetlający komunikat.
 */
class NotHelloWorldPanel extends JComponent
{
   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public void paintComponent(Graphics g)
   {
      g.drawString("To nie jest program „Witaj, świecie”.", MESSAGE_X, MESSAGE_Y);
   }
    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
DEFAULT_HEIGHT); }
}
```

`javax.swing.JFrame` **1.2**

- `Container getContentPane()`

  Zwraca obiekt `ContentPane` dla ramki `JFrame`.

- `Component add(Component c)`

  Dodaje i zwraca dany komponent do warstwy treści ramki (przed Java SE 5.0 ta metoda powodowała wyjątek).

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

- `void repaint()`

  Powoduje ponowne jak najszybsze narysowanie komponentu.

- `Dimension getPreferredSize()`

  Metoda, którą należy przesłonić, aby zwracała preferowany rozmiar komponentu.

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

- `void paintComponent(Graphics g)`

  Metoda, którą należy przesłonić w celu zdefiniowania sposobu rysowania określonego komponentu.

`java.awt.Window` **1.0**

- `void pack()`

  Zmienia rozmiar okna, biorąc pod uwagę preferowane rozmiary znajdujących się w nim komponentów.

7.5. Figury 2D

Od Java 1.0 klasa `Graphics` udostępnia metody rysujące linie, prostokąty, elipsy itd. Ich możliwości są jednak bardzo ograniczone. Nie ma na przykład możliwości ustawienia grubości linii ani obracania figur.

W Java 1.2 wprowadzono bibliotekę `Java2D` udostępniającą szeroki wachlarz metod graficznych. W tym rozdziale opisujemy tylko podstawy tej biblioteki — więcej bardziej zaawansowanych informacji na ten temat znajduje się w rozdziale 7. w drugim tomie.

Aby narysować figurę biblioteki `Java2D`, trzeba utworzyć obiekt klasy `Graphics2D`. Klasa ta jest podklasą klasy `Graphics`. Od Java SE 2 metody takie jak `paintComponent` automatycznie odbierają obiekty klasy `Graphics2D`. Wystarczy zastosować rzutowanie:

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

Figury geometryczne w bibliotece Java2D są obiektami. Istnieją klasy reprezentujące linie, prostokąty i elipsy:

```
Line2D
Rectangle2D
Ellipse2D
```

Wszystkie te klasy implementują interfejs Shape.

> Biblioteka Java2D obsługuje także bardziej skomplikowane figury, jak łuki, krzywe drugiego i trzeciego stopnia oraz trajektorie. Więcej informacji na ten temat znajduje się w rozdziale 7. drugiego tomu.

Aby narysować figurę, należy najpierw utworzyć obiekt klasy implementującej interfejs Shape, a następnie wywołać metodę draw klasy Graphics2D. Na przykład:

```
Rectangle2D rect = . . .;
g2.draw(rect);
```

> Przed pojawieniem się biblioteki Java2D do rysowania figur używano metod klasy Graphics, np. drawRectangle. Na pierwszy rzut oka te stare wywołania wydają się prostsze, ale używając biblioteki Java2D, pozostawiamy sobie różne możliwości do wyboru — można później ulepszyć rysunki za pomocą rozmaitych narzędzi dostępnych w bibliotece Java2D.

Biblioteka Java2D nieco komplikuje programowanie. W przeciwieństwie do metod rysujących z wersji 1.0, w których współrzędne były liczbami całkowitymi, figury Java2D używają współrzędnych zmiennoprzecinkowych. W wielu przypadkach stanowi to duże ułatwienie dla programisty, ponieważ może on określać figury przy użyciu lepiej znanych mu jednostek (jak milimetry lub cale), które później są konwertowane na piksele. Biblioteka Java2D wykonuje obliczenia o pojedynczej precyzji na liczbach typu float w większości wykonywanych wewnętrznie działań. Pojedyncza precyzja w zupełności wystarcza — celem obliczeń geometrycznych jest przecież ustawienie pikseli na ekranie lub w drukarce. Dopóki błędy zaokrąglania mieszczą się w zakresie jednego piksela, rezultat wizualny nie cierpi. Ponadto obliczenia na liczbach typu float są na niektórych platformach szybsze, a dodatkowo wartości tego typu zajmują o połowę mniej miejsca niż wartości typu double.

Czasami jednak obliczenia na liczbach typu float bywają niewygodne, ponieważ Java niewzruszenie wymaga rzutowania, jeśli niezbędna jest konwersja wartości typu double na typ float. Przeanalizujmy na przykład poniższą instrukcję:

```
float f = 1.2;     // błąd
```

Ta instrukcja spowoduje błąd kompilacji, ponieważ stała 1.2 jest typu double i kompilator obawia się utraty danych. Rozwiązaniem jest dodanie przyrostka F do stałej zmiennoprzecinkowej:

```
float f = 1.2F;    // OK
```

Teraz przyjrzyjmy się poniższej instrukcji:

```
Rectangle2D r = . . .
float f = r.getWidth();          // błąd
```

Instrukcja ta spowoduje błąd kompilacji z tego samego powodu co poprzednia. Metoda `getWidth` zwraca wartość typu `double`. W tym przypadku rozwiązaniem jest rzutowanie:

```
float f = (float) r.getWidth();   // OK
```

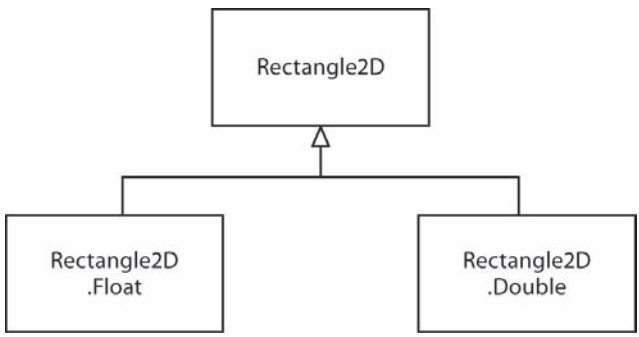
Ponieważ stosowanie przyrostków i rzutowania nie jest wygodne, projektanci biblioteki `Java2D` postanowili utworzyć **dwie wersje** każdej klasy reprezentującej figurę: jedną ze współrzędnymi typu `float` dla oszczędnych programistów i jedną ze współrzędnymi typu `double` dla leniwych (w tej książce zaliczamy się do tych drugich, czyli stosujemy współrzędne typu `double`, gdzie tylko możemy).

Projektanci biblioteki zastosowali ciekawą i początkowo wprowadzającą w błąd metodę pakowania obu wersji klas. Przyjrzyjmy się klasie `Rectangle2D`. Jest to abstrakcyjna klasa mająca dwie konkretne podklasy, które są dodatkowo statycznymi klasami wewnętrznymi:

```
Rectangle2D.Float
Rectangle2D.Double
```

Rysunek 7.9 przedstawia diagram dziedziczenia.

Rysunek 7.9. Klasy prostokątów 2D



Najlepiej ignorować fakt, że obie te konkretne klasy są statyczne i wewnętrzne — to tylko taki chwyt, który pozwala uniknąć nazw `FloatRectangle2D` i `DoubleRectangle2D` (więcej informacji na temat statycznych klas wewnętrznych znajduje się w rozdziale 6.).

Tworząc obiekt klasy `Rectangle2D.Float`, wartości określające współrzędne należy podawać jako typu `float`. W przypadku klasy `Rectangle2D.Double` współrzędne muszą być typu `double`.

```
Rectangle2D.Float floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Ponieważ zarówno klasa `Rectangle2D.Float`, jak i `Rectangle2D.Double` rozszerzają wspólną klasę `Rectangle2D`, a metody w tych podklasach przesłaniają metody nadklasy, zapamiętywanie typu figury nie przynosi właściwie żadnych korzyści. Referencje do prostokątów można przechowywać w zmiennych typu `Rectangle2D`.

```
Rectangle2D floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Oznacza to, że użycie klas wewnętrznych jest konieczne tylko przy tworzeniu obiektów figur.

Parametry konstruktora określają lewy górny róg, szerokość i wysokość prostokąta.

> Klasa `Rectangle2D.Float` zawiera jedną metodę, której nie dziedziczy po klasie `Rectangle2D`. Jest to metoda `setRect(float x, float y, float h, float w)`. Metody tej nie można użyć, jeśli referencja do obiektu typu `Rectangle2D.Float` jest przechowywana w zmiennej typu `Rectangle2D`. Nie jest to jednak duża strata — klasa `Rectangle2D` zawiera metodę `setRect` z parametrami typu `double`.

Metody klasy `Rectangle2D` przyjmują parametry i zwracają wartości typu `double`. Na przykład metoda `getWidth` zwraca wartość typu `double`, nawet jeśli szerokość jest zapisana w postaci liczby typu `float` w obiekcie typu `Rectangle2D.Float`.

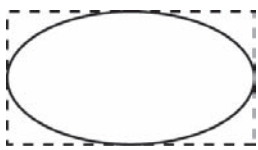
> Aby całkowicie pozbyć się wartości typu `float`, należy używać klas typu `Double`. Jednak w programach tworzących wiele tysięcy figur warto rozważyć użycie klas `Float` ze względu na oszczędność pamięci.

Wszystko, co napisaliśmy do tej pory na temat klas `Rectangle2D`, dotyczy również pozostałych klas reprezentujących figury. Dodatkowo istnieje klasa o nazwie `Point2D`, której podklasy to `Point2D.Float` i `Point2D.Double`. Poniższy fragment programu tworzy obiekt takiej klasy:

```
Point2D p = new Point2D.Double(10, 20);
```

> Klasa `Point2D` jest niezwykle przydatna — zastosowanie obiektów `Point2D` jest znacznie bliższe idei programowania obiektowego niż używanie oddzielnych wartości `x` i `y`. Wiele konstruktorów przyjmuje parametry typu `Point2D`. Zalecamy stosowanie obiektów tej klasy, gdzie się da — dzięki nim obliczenia geometryczne są często dużo prostsze.

Klasy `Rectangle2D` i `Ellipse2D` dziedziczą po wspólnej nadklasie `RectangularShape`. Wprawdzie elipsa nie jest prostokątna, ale można na niej **opisać prostokąt** (zobacz rysunek 7.10).

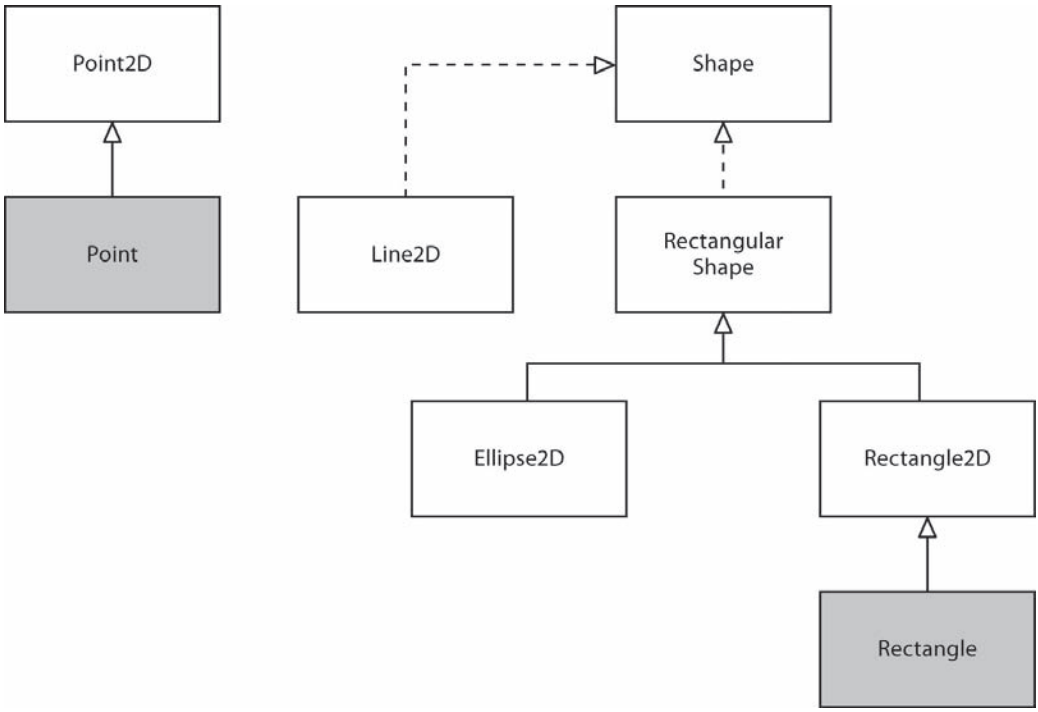


Rysunek 7.10. Prostokąt opisany na elipsie

Klasa `RectangularShape` definiuje ponad 20 metod wspólnych dla tych figur. Zaliczają się do nich metody `getWidth`, `getHeight`, `getCenterX` i `getCenterY` (niestety w czasie pisania tej książki nie było metody `getCenter` zwracającej obiekt typu `Point2D`).

Dodatkowo do hierarchii klas reprezentujących figury dodano kilka starszych klas z Java 1.0. Klasy `Rectangle` i `Point`, które przechowują prostokąt i punkt przy użyciu współrzędnych całkowitych, rozszerzają klasy `Rectangle2D` i `Point2D`.

Rysunek 7.11 przedstawia relacje pomiędzy klasami figur. Klasy `Double` i `Float` zostały pominięte, a klasy spadkowe wyróżniono szarym tłem.



Rysunek 7.11. Relacje między klasami figur

Tworzenie obiektów typu `Rectangle2D` i `Ellipse2D` jest prostym zadaniem. Należy podać:

- współrzędne `x` i `y` lewego górnego rogu,
- wysokość i szerokość.

W przypadku elipsy te wartości dotyczą opisanego na niej prostokąta. Na przykład instrukcja:

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

utworzy elipsę wpisaną w prostokąt, którego lewy górny róg znajduje się w punkcie o współrzędnych (150, 200) o szerokości 100 i wysokości 50.

Czasami jednak współrzędne lewego górnego rogu nie są od razu dostępne. Często zdarza się, że dostępne są dwa punkty leżące naprzeciw siebie, ale nie są to rogi górny lewy i prawy dolny. Nie można utworzyć prostokąta w poniższy sposób:

```
Rectangle2D rect = new Rectangle2D.Double(px, py, qx - px, qy - py);    // błąd
```

Jeśli `p` nie jest lewym górnym rogiem, jedna lub obie współrzędne będą miały wartości ujemne i prostokąt się nie pojawi. W takim przypadku należy najpierw utworzyć pusty prostokąt i użyć metody `setFrameFromDiagonal`:

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

Jeszcze lepiej, jeśli p i q są punktami rogów reprezentowanymi przez obiekty typu Point2D:

```
rect.setFrameFromDiagonal(p, q);
```

Przy tworzeniu elipsy zazwyczaj znane są środek, szerokość i wysokość opisanego na niej prostokąta, a nie jego rogi (które nawet nie leżą na elipsie). Metoda setFrameFromCenter przyjmuje punkt środkowy, ale wymaga także jednego z czterech rogów. W związku z tym elipsę zazwyczaj tworzy się następująco:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2, centerY - height / 2,
width, height);
```

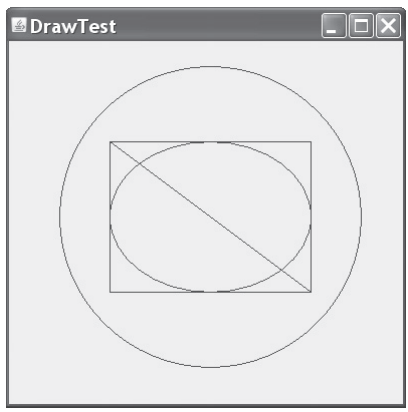
Aby utworzyć linię, należy podać jej punkt początkowy i końcowy w postaci obiektów Point2D lub par liczb:

```
Line2D line = new Line2D.Double(start, end);
```

lub

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

Program przedstawiony na listingu 7.4 rysuje prostokąt, elipsę znajdującą się wewnątrz tego prostokąta, przekątną prostokąta oraz koło o takim samym środku jak prostokąt. Rysunek 7.12 przedstawia wynik działania tego programu.



Rysunek 7.12. Rysowanie figur geometrycznych

Listing 7.4. draw/DrawTest.java

```
package draw;

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

/**
 * @version 1.32 2007-04-14
 * @author Cay Horstmann
 */
public class DrawTest
{
   public static void main(String[] args)
   {
```

```
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new DrawFrame();
               frame.setTitle("DrawTest");
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * Ramka zawierająca panel z rysunkami.
 */
class DrawFrame extends JFrame
{
   public DrawFrame()
   {
      add(new DrawComponent());
      pack();
   }
}

/**
 * Komponent wyświetlający prostokąty i elipsy.
 */
class DrawComponent extends JComponent
{
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;

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

      // Rysowanie prostokąta.

      double leftX = 100;
      double topY = 100;
      double width = 200;
      double height = 150;

      Rectangle2D rect = new Rectangle2D.Double(leftX, topY, width, height);
      g2.draw(rect);

      // Rysowanie elipsy.

      Ellipse2D ellipse = new Ellipse2D.Double();
      ellipse.setFrame(rect);
      g2.draw(ellipse);

      // Rysowanie przekątnej.

      g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));
```

```
        // Rysowanie koła z takim samym środkiem.

        double centerX = rect.getCenterX();
        double centerY = rect.getCenterY();
        double radius = 150;

        Ellipse2D circle = new Ellipse2D.Double();
        circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + radius);
        g2.draw(circle);
    }
    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
DEFAULT_HEIGHT); }
}
```

`java.awt.geom.RectangularShape` **1.2**

- `double getCenterX()`
- `double getCenterY()`
- `double getMinX()`
- `double getMinY()`
- `double getMaxX()`
- `double getMaxY()`

  Zwraca współrzędną `x` lub `y` punktu środkowego, punktu o najmniejszych lub największych współrzędnych prostokąta.

- `double getWidth()`
- `double getHeight()`

  Zwraca szerokość lub wysokość prostokąta.

- `double getX()`
- `double getY()`

  Zwraca współrzędną `x` lub `y` lewego górnego rogu prostokąta.

`java.awt.geom.Rectangle2D.Double` **1.2**

- `Rectangle2D.Double(double x, double y, double w, double h)`

  Tworzy prostokąt z lewym górnym rogiem w podanym miejscu i o podanej szerokości i długości.

`java.awt.geom.Rectangle2D.Float` **1.2**

- `Rectangle2D.Float(float x, float y, float w, float h)`

  Tworzy prostokąt z lewym górnym rogiem w podanym miejscu i o podanej szerokości i długości.

`java.awt.geom.Ellipse2D.Double` **1.2**

- `Ellipse2D.Double(double x, double y, double w, double h)`

  Rysuje elipsę wpisaną w prostokąt, którego lewy górny róg znajduje się w podanym miejscu i który ma określone wysokość oraz szerokość.

`java.awt.geom.Point2D.Double` **1.2**

- `Point2D.Double(double x, double y)`

  Rysuje punkt o podanych współrzędnych.

`java.awt.geom.Line2D.Double` **1.2**

- `Line2D.Double(Point2D start, Point2D end)`
- `Line2D.Double(double startX, double startY, double endX, double endY)`

  Rysuje linię między dwoma podanymi punktami.

7.6. Kolory

Metoda `setPaint` z klasy `Graphics2D` ustawia kolor, który jest stosowany we wszystkich kolejnych rysunkach graficznych. Na przykład:

```
g2.setPaint(Color.RED);
g2.drawString("Uwaga!", 100, 100);
```

Figury zamknięte (np. prostokąt czy elipsa) można w takiej sytuacji wypełnić za pomocą metody `fill` (zamiast `draw`):

```
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect);    // Wypełnienie prostokąta rect kolorem czerwonym.
```

Aby zastosować kilka kolorów, należy wybrać kolor, zastosować metodę `draw` lub `fill`, a następnie wybrać inny kolor i ponownie zastosować metodę `draw` lub `fill`.

> Metoda `fill` rysuje o jeden piksel mniej po prawej i na dole. Jeśli na przykład narysujemy prostokąt `new Rectangle2D.Double(0, 0, 10, 20)`, to rysunek będzie obejmował piksele o współrzędnych `x` = 10 i `y` = 20. Jeśli wypełnimy ten prostokąt kolorem, piksele te nie zostaną pokolorowane.

Do definiowania kolorów służy klasa `java.awt.Color`. W klasie tej dostępnych jest 13 następujących predefiniowanych stałych reprezentujących kolory:

```
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED,
↪WHITE, YELLOW
```

> Przed Java SE 1.4 stałe określające kolory były pisane małymi literami, np. `Color.red`. Było to sprzeczne z przyjętą konwencją pisania stałych wielkimi literami. Obecnie nazwy tych stałych można pisać wielkimi lub, ze względu na zgodność wsteczną, małymi literami.

Niestandardowy kolor można zdefiniować, tworząc obiekt klasy `Color` i podając wartości trzech składowych: czerwonego, zielonego i niebieskiego. Wartość każdego ze składników (zajmujących po jednym bajcie) musi należeć do zbioru 0 – 255. Poniższy kod przedstawia sposób wywołania konstruktora klasy `Color` z parametrami określającymi stopień czerwieni, niebieskiego i zieleni:

```
Color(int redness, int greenness, int blueness)
```

Poniżej znajduje się przykładowa procedura tworząca niestandardowy kolor:

```
g2.setPaint(new Color(0, 128, 128));    // niebieskozielony
g2.drawString("Witaj!", 75, 125);
```

> Poza jednolitymi kolorami można także stosować bardziej skomplikowane ustawienia, jak różne odcienie czy obrazy. Więcej informacji na ten temat znajduje się w drugim tomie w rozdziale o zaawansowanych technikach AWT. Jeśli zamiast obiektu typu `Graphics2D` zostanie użyty obiekt `Graphics`, kolory należy ustawiać za pomocą metody `setColor`.

Do ustawiania **koloru tła** służy metoda `setBackground` z klasy `Component`, będącej nadklasą klasy `JComponent`.

```
MyComponent p = new MyComponent();
p.setBackground(Color.PINK);
```

Istnieje też metoda `setForeground`, która określa kolor elementów rysowanych na komponencie.

> Metody `brighter()` i `darker()` — jak sama nazwa wskazuje — sprawiają, że aktualnie używany kolor staje się jaśniejszy bądź ciemniejszy. Ponadto metoda `brighter` jest dobrym sposobem na wyróżnienie wybranego elementu. W rzeczywistości metoda ta nieznacznie rozjaśnia kolor. Aby kolor był dużo jaśniejszy, można tę metodę zastosować trzy razy: `c.brighter().brighter().brighter()`.

Znacznie więcej predefiniowanych nazw kolorów znajduje się w klasie `SystemColor`. Stałe tej klasy określają kolory stosowane do rozmaitych elementów systemu użytkownika. Na przykład instrukcja:

```
p.setBackground(SystemColor.window)
```

ustawia kolor tła komponentu na domyślny dla wszystkich okien w systemie użytkownika (tło jest wstawiane przy każdym rysowaniu okna). Kolory zdefiniowane w klasie `SystemColor` są szczególnie przydatne, kiedy chcemy narysować elementy interfejsu użytkownika nieodbiegające kolorystyką od standardowych elementów w systemie. Tabela 7.1 zawiera nazwy kolorów systemowych oraz ich opisy.

Tabela 7.1. System kolorów

Nazwa	Zastosowanie
desktop	Kolor tła pulpitu
activeCaption	Kolor belki tytułowej aktywnego okna
activeCaptionText	Kolor tekstu na belce tytułowej
activeCaptionBorder	Kolor obramowania aktywnej belki
inactiveCaption	Kolor nieaktywnej belki
inactiveCaptionText	Kolor tekstu nieaktywnej belki
inactiveCaptionBorder	Kolor obramowania nieaktywnej belki
window	Tło okna
windowBorder	Kolor obramowania okna
windowText	Kolor tekstu w oknie
menu	Tło menu
menuText	Kolor tekstu w menu
text	Kolor tła tekstu
textText	Kolor tekstu
textInactiveText	Kolor tekstu nieaktywnych elementów sterujących
textHighlight	Kolor tła wyróżnionego tekstu
textHighlightText	Kolor wyróżnionego tekstu
control	Kolor tła elementów sterujących
controlText	Kolor tekstu w elementach sterujących
controlLtHighlight	Słabe wyróżnienie elementów sterujących
controlHighlight	Silne wyróżnienie elementów sterujących
controlShadow	Kolor cienia elementów sterujących
controlDkShadow	Ciemniejszy kolor cienia elementów sterujących
scrollbar	Kolor tła dla suwaków
info	Kolor tła dla tekstu pomocy
infoText	Kolor tekstu pomocy

`java.awt.Color` **1.0**

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

  Tworzy obiekt reprezentujący kolor.

  Parametry: `r` Wartość barwy czerwonej

  `g` Wartość barwy zielonej

  `b` Wartość barwy niebieskiej

`java.awt.Graphics` **1.0**

- `Color getColor()`
- `void setColor(Color c)`

  Pobiera lub ustawia kolor. Wszystkie następne rysunki będą miały ten kolor.

  Parametry: `c` Nowy kolor

`java.awt.Graphics2D` **1.2**

- `Paint getPaint()`
- `void setPaint(Paint p)`

  Pobiera lub ustawia własność `paint` danego kontekstu graficznego. Klasa `Color` implementuje interfejs `Paint`. W związku z tym za pomocą tej metody można ustawić atrybut `paint` na jednolity kolor.

- `void fill(shape s)`

  Wypełnia figurę aktualnym kolorem.

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

- `Color getBackground()`
- `void setBackground(Color c)`

  Pobiera lub ustawia kolor tła.

  Parametry: `c` Nowy kolor tła

- `Color getForeground()`
- `void setForeground(Color c)`

  Pobiera lub ustawia kolor frontu.

  Parametry: `c` Nowy kolor frontu

7.7. Czcionki

Program przedstawiony na początku tego rozdziału wyświetlał łańcuch tekstu pisany domyślną czcionką. Często jednak zdarza się, że tekst musi być napisany inną czcionką. Identyfikatorem czcionki jest jej **nazwa**. Nazwa czcionki składa się z **nazwy rodziny czcionek**, np. Helvetica, i opcjonalnego przyrostka, np. Bold. Na przykład nazwy Helvetica i Helvetica Bold należą do rodziny czcionek o nazwie Helvetica.

Aby sprawdzić, jakie czcionki są dostępne w danym komputerze, należy wywołać metodę `getAvailableFontFamilyNames` z klasy `GraphicsEnvironment`. Ta metoda zwraca tablicę nazw wszystkich dostępnych czcionek w postaci łańcuchów. Egzemplarz klasy `GraphicsEnvi`
`↪ronment` reprezentujący środowisko graficzne systemu użytkownika można utworzyć za

pomocą statycznej metody `getLocalGraphicsEnvironment`. Poniższy program drukuje nazwy wszystkich czcionek znajdujących się w systemie:

```
import java.awt.*;

public class ListFonts
{
   public static void main(String[] args)
   {
      String[] fontNames = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();
      for (String fontName : fontNames)
         System.out.println(fontName);
   }
}
```

W jednym z systemów początek tej listy wygląda następująco:

```
Abadi MT Condensed Light
Arial
Arial Black
Arial Narrow
Arioso
Baskerville
Binner Gothic
. . .
```

Lista ta zawiera ponad 70 pozycji.

Nazwy czcionek mogą być znakami towarowymi, a ich projekty mogą w niektórych jurysdykcjach podlegać prawom autorskim. W związku z tym dystrybucja czcionek często wiąże się z uiszczaniem opłat licencyjnych ich właścicielom. Oczywiście, podobnie jak są tanie podróbki drogich perfum, istnieją też podróbki czcionek imitujące oryginały. Na przykład imitacja czcionki Helvetica w systemie Windows nosi nazwę Arial.

Jako wspólny punkt odniesienia w bibliotece AWT zdefiniowano pięć **logicznych** nazw czcionek:

```
SansSerif
Serif
Monospaced
Dialog
DialogInput
```

Czcionki te są zawsze zamieniane na czcionki, które znajdują się w danym urządzeniu. Na przykład w systemie Windows czcionka SansSerif jest zastępowana czcionką Arial.

Dodatkowo pakiet SDK firmy Sun zawsze zawiera trzy rodziny czcionek: Lucida Sans, Lucida Bright i Lucida Sans Typewriter.

Aby narysować znak daną czcionką, najpierw trzeba utworzyć obiekt klasy `Font`. Konieczne jest podanie nazwy i stylu czcionki oraz rozmiaru w punktach drukarskich. Poniższa instrukcja tworzy obiekt klasy `Font`:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

Trzeci argument określa rozmiar w punktach. Jednostka ta jest powszechnie stosowana w typografii do określania rozmiaru czcionek. Jeden punkt jest równy 1/72 cala, czyli około 0,35 mm.

W konstruktorze klasy `Font` można użyć logicznej nazwy czcionki zamiast nazwy fizycznej. Styl (zwykły, **pogrubiony**, *kursywa* lub ***pogrubiona kursywa***) określa drugi argument konstruktora `Font`, który może mieć jedną z poniższych wartości:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```

> Sposób odwzorowania logicznych nazw czcionek na fizyczne jest określony w pliku *fontconfig.properties* w katalogu *jre/lib* znajdującym się w folderze instalacji Javy. Informacje na temat tego pliku znajdują się pod adresem *http://docs.oracle.com/javase/7/docs/technotes/guides/intl/fontconfig.html*.

Pliki czcionek można wczytywać w formatach TrueType lub PostScript type 1. Potrzebny jest do tego strumień wejściowy dla danej czcionki — zazwyczaj z pliku lub adresu URL (więcej informacji na temat strumieni znajduje się w rozdziale 1. drugiego tomu). Następnie należy wywołać statyczną metodę `Font.createFont`:

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f1 = Font.createFont(Font.TRUETYPE_FONT, in);
```

Zastosowana została zwykła czcionka o rozmiarze 1 punktu. Do określenia żądanego rozmiaru czcionki należy użyć metody `deriveFont`:

```
Font f = f1.deriveFont(14.0F);
```

> Istnieją dwie przeciążone wersje metody `deriveFont`. Jedna z nich (przyjmująca parametr typu `float`) ustawia rozmiar czcionki, a druga (przyjmująca parametr typu `int`) ustawia styl czcionki. W związku z tym instrukcja `f1.deriveFont(14)` ustawia styl, a nie rozmiar czcionki! Styl czcionki będzie w tym przypadku kursywą, ponieważ binarna reprezentacja liczby 14 ustawia bit `ITALIC`.

Fonty Javy zawierają symbole i znaki ASCII. Na przykład znak *\u2297* fontu Dialog to znak ⊗. Dostępne są tylko te symbole, które zdefiniowano w zestawie znaków Unicode.

Poniższy fragment programu wyświetla napis *Witaj, świecie!* standardową czcionką bezszeryfową systemu z zastosowaniem pogrubienia i o rozmiarze 14 punktów:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Witaj, świecie!";
g2.drawString(message, 75, 100);
```

Teraz **wypośrodkujemy** nasz napis w zawierającym go komponencie. Do tego celu potrzebne są informacje o szerokości i wysokości łańcucha w pikselach. O wymiarach tych decydują trzy czynniki:

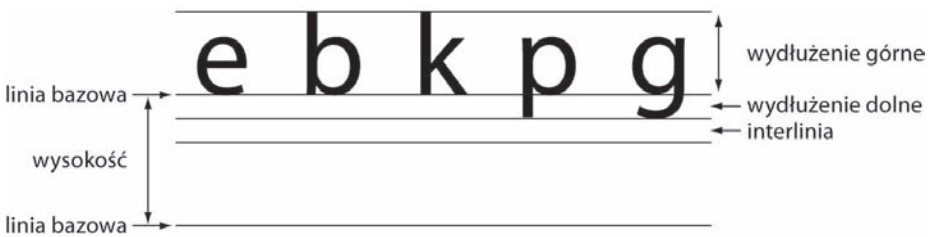
- czcionka (w tym przypadku jest to pogrubiona czcionka bezszeryfowa o rozmiarze 14 punktów),
- łańcuch (w tym przypadku *Witaj, świecie!*),
- urządzenie, na którym łańcuch będzie wyświetlany (w tym przypadku ekran monitora).

Obiekt reprezentujący własności czcionki urządzenia z ekranem tworzymy za pomocą metody `getFontRenderContext` z klasy `Graphics2D`. Zwraca ona obiekt klasy `FontRenderContext`. Obiekt ten należy przekazać metodzie `getStringBounds` z klasy `Font`:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

Metoda `getStringBounds` zwraca prostokąt, w którym mieści się łańcuch.

Do interpretacji wymiarów tego prostokąta potrzebna jest znajomość podstawowych pojęć z zakresu składu tekstów (zobacz rysunek 7.13). **Linia bazowa** (ang. *baseline*) to teoretyczna linia, na której opiera się dolna część litery, np. *e*. **Wydłużenie górne** (ang. *ascent*) to odstęp dzielący linię bazową i **linię górną pisma** (ang. *ascender*), która określa górną granicę liter, takich jak *b* lub *k* czy też wielkich liter. **Wydłużenie dolne** (ang. *descent*) to odległość pomiędzy linią bazową a **linią dolną pisma** (ang. *descender*), która stanowi granicę dolnej części takich liter jak *p* lub *g*.



Rysunek 7.13. Pojęcia z zakresu składu tekstów

Interlinia (ang. *leading*) to odstęp pomiędzy wydłużeniem dolnym jednej linii a wydłużeniem górnym następnej linii (termin pochodzi od pasków ołowiu używanych przez zecerów do oddzielania linii). **Wysokość** (ang. *height*) czcionki to odległość pomiędzy następującymi po sobie liniami bazowymi i jest równa sumie wydłużenia dolnego, leadingu i wydłużenia górnego.

Szerokość prostokąta zwracanego przez metodę `getStringBounds` określa szerokość tekstu. Wysokość natomiast jest równa sumie wydłużenia dolnego, leadingu i wydłużenia górnego. Prostokąt ma swój początek na linii bazowej łańcucha. Górna współrzędna `y` prostokąta ma wartość ujemną. W związku z tym szerokość, wysokość i wydłużenie górne łańcucha można sprawdzić następująco:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

Aby sprawdzić wydłużenie dolne lub leading, należy użyć metody `getLineMetrics` klasy `Font`. Zwraca ona obiekt klasy `LineMetrics`, dysponujący metodami do sprawdzania wydłużenia dolnego i leadingu:

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

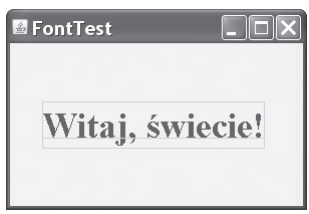
W poniższym fragmencie programu wykorzystano wszystkie opisane powyżej informacje do umieszczenia łańcucha na środku zawierającego go komponentu:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);

// (x, y) = lewy górny róg tekstu
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;

// Dodanie wydłużenia górnego do y w celu sięgnięcia do linii bazowej.
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

Aby ułatwić sobie zrozumienie techniki wyśrodkowywania tekstu, warto sobie uzmysłowić, że metoda `getWidth()` zwraca szerokość komponentu. Pewna część tej przestrzeni, `bounds.getWidth()`, jest zajmowana przez tekst. Reszta powinna być podzielona na dwie równe części, rozmieszczone po obu stronach tekstu. Ten sam sposób rozumowania dotyczy wysokości.

> Kiedy konieczne jest obliczenie wymiarów układu bez użycia metody `paintComponent`, nie można uzyskać obiektu obrazowania czcionki typu `Graphics2D`. W zamian należy wywołać metodę `getFontMetrics` klasy `JComponent`, a następnie metodę `getFontRenderContext`.
>
> ```
> FontRenderContext context = getFontMetrics(f).getFontRenderContext();
> ```

Przykładowy program przedstawiony poniżej nie tylko drukuje napis, ale także linię bazową i prostokąt otaczający napis. Rysunek 7.14 przedstawia wynik działania tego programu. Listing 7.5 zawiera jego kod.



Rysunek 7.14. Linia bazowa i prostokąt otaczający łańcuch

Listing 7.5. font/FontTest.java

```
package font;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * @version 1.33 2007-04-14
```

```
 * @author Cay Horstmann
 */
public class FontTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new FontFrame();
               frame.setTitle("FontTest");
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * Ramka z komponentem zawierającym tekst.
 */
class FontFrame extends JFrame
{
   public FontFrame()
   {
      add(new FontComponent());
      pack();
   }
}

/**
 * Komponent z tekstem w ramce na środku.
 */
class FontComponent extends JComponent
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

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

      String message = "Witaj, świecie!";

      Font f = new Font("Serif", Font.BOLD, 36);
      g2.setFont(f);

      // Sprawdzenie rozmiaru tekstu.

      FontRenderContext context = g2.getFontRenderContext();
      Rectangle2D bounds = f.getStringBounds(message, context);

      // set (x, y) = lewy górny róg tekstu

      double x = (getWidth() - bounds.getWidth()) / 2;
      double y = (getHeight() - bounds.getHeight()) / 2;
```

```
        // Dodanie wydłużenia górnego do y w celu sięgnięcia do linii bazowej.

        double ascent = -bounds.getY();
        double baseY = y + ascent;

        // Rysowanie komunikatu.

        g2.drawString(message, (int) x, (int) baseY);

        g2.setPaint(Color.LIGHT_GRAY);

        // Rysowanie linii bazowej.

        g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));

        // Rysowanie otaczającego tekst prostokąta.

        Rectangle2D rect = new Rectangle2D.Double(x, y, bounds.getWidth(),
        ↪bounds.getHeight());
        g2.draw(rect);
    }
    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
DEFAULT_HEIGHT); }
}
```

`java.awt.Font` **1.0**

- `Font(String name, int style, int size)`

  Tworzy obiekt reprezentujący czcionkę.

  Parametry:

`name`	Nazwa czcionki — może być nazwa typu Helvetica Bold lub logiczna nazwa typu Serif lub SansSerif
`style`	Styl: `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` lub `Font.BOLD + Font.ITALIC`
`size`	Rozmiar w punktach (na przykład 12)

- `String getFontName()`

  Pobiera nazwę czcionki (typu Helvetica Bold).

- `String getFamily()`

  Pobiera nazwę rodziny czcionek (np. Helvetica).

- `String getName()`

  Pobiera nazwę logiczną (np. SansSerif), jeśli czcionka została utworzona z nazwy logicznej. W przeciwnym przypadku zwraca nazwę czcionki.

- `Rectangle 2D getStringBounds(String s, FontRenderContext context)` **1.2**

  Zwraca prostokąt otaczający łańcuch. Prostokąt ma swój początek na linii bazowej łańcucha. Górna współrzędna `y` prostokąta ma wartość równą odwrotności wydłużenia górnego. Wysokość prostokąta jest równa sumie wydłużenia górnego, dolnego i leadingu. Szerokość jest równa szerokości tekstu.

- `LineMetrics getLineMetrics(String s, FontRenderContext context)` **1.2**

  Zwraca ona obiekt klasy `LineMetrics` dysponujący metodami do sprawdzania wydłużenia dolnego i leadingu.

- `Font deriveFont(int style)` **1.2**
- `Font deriveFont(float size)` **1.2**
- `Font deriveFont(int style, float size)` **1.2**

  Zwraca nową czcionkę różniącą się od aktualnej tylko rozmiarem podanym jako argument.

`java.awt.font.LineMetrics` **1.2**

- `float getAscent()`

  Pobiera wydłużenie górne czcionki — odległość linii bazowej od wierzchołków wielkich liter.

- `float getDescent()`

  Pobiera wydłużenie dolne czcionki — odległość linii bazowej od podstaw liter sięgających dolnej linii pisma.

- `float getLeading()`

  Pobiera leading czcionki — odstęp pomiędzy spodem jednej linii tekstu a wierzchołkiem następnej.

- `float getHeight()`

  Pobiera całkowitą wysokość czcionki — odległość pomiędzy dwiema liniami bazowymi tekstu (wydłużenie dolne + leading + wydłużenie górne).

`java.awt.Graphics` **1.0**

- `Font getFont()`
- `void setFont(Font font)`

  Pobiera lub ustawia czcionkę. Czcionka ta będzie stosowana w kolejnych operacjach rysowania tekstu.

  Parametry: `font` Czcionka

- `void drawString(String str, int x, int y)`

  Rysuje łańcuch przy użyciu aktualnej czcionki i koloru.

  Parametry: `str` Łańcuch

  `x` Współrzędna `x` początku łańcucha

  `y` Współrzędna `y` linii bazowej łańcucha

`java.awt.Graphics2D` **1.2**

- `FontRenderContext getFontRenderContext()`

  Pobiera kontekst wizualizacji czcionki, który określa cechy czcionki w kontekście graficznym.

- `void drawString(String str, float x, float y)`

  Rysuje łańcuch przy zastosowaniu aktualnej czcionki i koloru.

  Parametry:

`str`	Łańcuch
`x`	Współrzędna `x` początku łańcucha
`y`	Współrzędna `y` linii bazowej łańcucha

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

- `FontMetrics getFontMetrics(Font f)` **5.0**

  Pobiera cechy czcionki. Klasa `FontMetrics` jest prekursorem klasy `LineMetrics`.

`java.awt.FontMetrics` **1.0**

- `FontRenderContext getFontRenderContext()` **1.2**

  Pobiera kontekst wizualizacji czcionki.

7.8. Wyświetlanie obrazów

Poznaliśmy techniki tworzenia prostych rysunków składających się z linii i figur geometrycznych. Bardziej złożone obrazy, jak zdjęcia, mają zazwyczaj inne pochodzenie, np. przenosi się je do komputera za pomocą skanera lub wytwarza w wyspecjalizowanym do tego celu oprogramowaniu. W drugim tomie nauczymy się tworzyć obrazy złożone z pojedynczych pikseli zapisanych w tablicy — technika ta jest często używana na przykład podczas tworzenia obrazów fraktalnych.

Obrazy zapisane w postaci plików na dysku lub w internecie można wczytać do aplikacji w Javie i wyświetlić na obiektach `Graphics`. Obrazy można wczytywać na wiele sposobów. W poniższym przykładzie użyta jest znana nam już klasa `ImageIcon`:

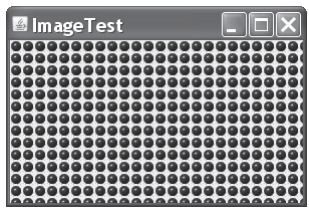
```
Image image = new ImageIcon(filename).getImage();
```

Zmienna `image` zawiera referencję do obiektu opakowującego obraz. Można go wyświetlić za pomocą metody `drawImage` z klasy `Graphics`:

```
public void paintComponent(Graphics g)
{
   . . .
   g.drawImage(image, x, y, null);
}
```

Program z listingu 7.6 robi nawet więcej, ponieważ wypełnia całe okno wieloma obrazami. Rezultat tego widać na rysunku 7.15. Za to kaskadowe wypełnienie odpowiedzialna jest metoda `paintComponent`. Najpierw rysujemy jeden obraz w lewym górnym rogu, a następnie zapełniamy całe okno za pomocą metody `copyArea`:

```
for (int i = 0; i * imageWidth <= getWidth(); i++)
   for (int j = 0; j * imageHeight <= getHeight(); j++)
      if (i + j > 0)
         g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j * imageHeight);
```



Rysunek 7.15. Okno wypełnione kopiami jednego obrazu

Listing 7.6 przedstawia pełny kod źródłowy opisywanego program.

Listing 7.6. image/ImageTest.java

```
package image;

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

/**
 * @version 1.33 2007-04-14
 * @author Cay Horstmann
 */
public class ImageTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ImageFrame();
               frame.setTitle("ImageTest");
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * Ramka zawierająca komponent obrazu.
 */
class ImageFrame extends JFrame
{
   public ImageFrame()
   {
      add(new ImageComponent());
```

```
      pack();
   }
}

/**
 * Komponent wyświetlający powielony obraz.
 */
class ImageComponent extends JComponent
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   private Image image;

   public ImageComponent()
   {
      image = new ImageIcon("blue-ball.gif").getImage();
   }

   public void paintComponent(Graphics g)
   {
      if (image == null) return;

      int imageWidth = image.getWidth(this);
      int imageHeight = image.getHeight(this);

      // Rysowanie obrazu w lewym górnym rogu.

      g.drawImage(image, 0, 0, null);
      // Powielenie obrazu w obrębie komponentu.

      for (int i = 0; i * imageWidth <= getWidth(); i++)
         for (int j = 0; j * imageHeight <= getHeight(); j++)
            if (i + j > 0) g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j
                  * imageHeight);
   }
   public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
   ↪DEFAULT_HEIGHT); }
}
```

`java.awt.Graphics` **1.0**

- `boolean drawImage(Image img, int x, int y, ImageObserver observer)`

  Rysuje obraz w naturalnym rozmiarze. Uwaga: to wywołanie może zwrócić wartość przed narysowaniem obrazu.

  Parametry:

`img`	Obraz do narysowania
`x`	Współrzędna `x` lewego górnego rogu
`y`	Współrzędna `y` lewego górnego rogu
`observer`	Obiekt powiadamiający o postępie procesu wizualizacji (może być wartość `null`)

- `boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)`

  Rysuje obraz o zmienionych wymiarach. System dopasowuje rozmiar obrazu do obszaru o podanej szerokości i wysokości. Uwaga: to wywołanie może zwrócić wartość przed narysowaniem obrazu.

  Parametry:

`img`	Obraz do narysowania
`x`	Współrzędna `x` lewego górnego rogu
`y`	Współrzędna `y` lewego górnego rogu
`width`	Szerokość obrazu
`height`	Wysokość obrazu
`observer`	Obiekt powiadamiający o postępie procesu wizualizacji (może być wartość `null`)

- `void copyArea(int x, int y, int width, int height, int dx, int dy)`

  Kopiuje obszar ekranu.

  Parametry:

`x`	Współrzędna `x` lewego górnego rogu obszaru źródłowego
`y`	Współrzędna `y` lewego górnego rogu obszaru źródłowego
`width`	Szerokość obszaru źródłowego
`height`	Wysokość obszaru źródłowego
`dx`	Odległość w poziomie od obszaru źródłowego do obszaru docelowego
`dy`	Odległość w pionie od obszaru źródłowego do obszaru docelowego

Na tym zakończymy wprowadzenie do grafiki w Javie. Bardziej zaawansowane techniki, takie jak grafika 2D i obróbka obrazów, zostały opisane w drugim tomie. W kolejnym rozdziale dowiemy się, jak programy reagują na dane wprowadzane przez użytkownika.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ź

Ż

WYKORZYSTAJ SIŁĘ OBIEKTÓW. ZACZNIJ PROGRAMOWAĆ OBIEKTOWO W JĘZYKU JAVA!

Pomimo zaawansowanego wieku Java wciąż jest na topie. Ten język programowania oraz narzędzia z nim powiązane są najczęściej wybierane do tworzenia rozbudowanych systemów informatycznych. Skąd ta popularność? Przejrzysta składnia, obsługa nowoczesnych technik przesyłania informacji, automatyczne czyszczenie pamięci to tylko niektóre z atutów Javy. Jeżeli dołożymy do tego ogromną rzeszę użytkowników chętnych do pomocy, wszystko staje się jasne. Java jeszcze długo będzie na świeczniku!

Kolejne wydanie tej cenionej książki zostało zaktualizowane o nowości, które pojawiły się w wersji 7 platformy Java Standard Edition. W trakcie lektury poznasz składnię języka oraz wszystkie istotne kwestie związane z programowaniem w Javie. Zrozumiesz założenia programowania obiektowego, nauczysz się korzystać z interfejsów oraz obsługiwać wyjątki. Przekonasz się również, jakie ułatwienia w tym zakresie oferuje Java 7 — obsługa wielu wyjątków w ramach jednego bloku catch to tylko jedno z wielu udogodnień. Książka ta jest idealną pozycją dla wszystkich osób chcących poznać język Java. Sprawdzi się ona również w rękach doświadczonych programistów — jako źródło informacji na temat nowości w Java Standard Edition 7.

POZNAJ:

- podstawy języka Java
- zasady programowania obiektowego
- zastosowanie interfejsów
- nowości wprowadzone w ostatniej wersji Javy

PRENTICE HALL
PEARSON EDUCATION

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

cena: 99,00 zł
ISBN 978-83-246-7758-0
9 788324 677580