Wykład IX

Java - środowisko GUI
Biblioteka Swing

Swing to framework obsługujących komponenty graficznego interfejsu użytkownika (GUI), które oferują większe możliwości i elastyczność w porównaniu z tradycyjnymi komponentami zestawu narzędzi AWT. W efekcie jest on pakietem do tworzenia graficznego interfejsu użytkownika.
Zestaw AWT (*Abstract Window Toolkit*). definiuje podstawowy zbiór kontrolek, okien i okien dialogowych niezbędnych do obsługi praktycznego, ale też dość ograniczonego interfejsu użytkownika. Jednym ze źródeł tych ograniczeń jest zdolność tłumaczenia oryginalnych kontrolek AWT na wiele różnych komponentów wizualnych, tzw. **odpowiedników**, stosowanych w odmiennych środowiskach graficznych róznych systemów operacyjnych. Oznacza to, że za wygląd i sposób obsługi poszczególnych komponentów odpowiada docelowa platforma, nie Java. Ponieważ komponenty zestawu narzędzi AWT stosują rdzenne zasoby kodu, określa się je mianem komponentów **ciężkich**.
Stosowanie rdzennych odpowiedników kontrolek rodzi wiele problemów:

1. Po pierwsze, różnice dzielące poszczególne systemy operacyjne powodują, że komponenty mogą wyglądać, a nawet działać nieco inaczej na różnych platformach. Ta potencjalna odmienność zagraża naczelnej zasadzie przyświecającej twórcom Javy: napisz raz, uruchamiaj gdziekolwiek.
2. Po drugie, wygląd i sposób obsługi poszczególnych komponentów zostały trwale zakodowane (są definiowane przez platformę) i jako takie nie mogą być (łatwo) zmieniane. Po trzecie, stosowanie komponentów ciężkich wiąże się z wieloma ograniczeniami. Przykładem takiego ograniczenia jest brak możliwości stosowania efektu przezroczystości.

Bibliotekę Swing wydano w 1997 roku w ramach klas JFC (*Java Foundation Classes*) zbudowano na bazie zestawu narzędzi AWT
Właśnie dlatego AWT wciąż stanowi jeden z najważniejszych elementów Javy.
Biblioteka Swing stosuje ten sam mechanizm obsługi zdarzeń co zestaw narzędzi AWT. Oznacza to, że stosowanie biblioteki Swing nie jest możliwe bez znajomości zestawu narzędzi AWT i zasad obsługi zdarzeń.
Komponenty biblioteki Swing są lekkie
Poza kilkoma wyjątkami komponenty biblioteki Swing są **lekkie**. Oznacza to, że są pisane w całości w Javie i jako takie nie są bezpośrednio odwzorowywane na swoje odpowiedniki stosowane na poszczególnych platformach. Komponenty lekkie są dzięki temu bardziej efektywne i elastyczne. Co więcej, ponieważ komponenty lekkie nie są tłumaczone na rdzenne kontrolki poszczególnych platform, o wyglądzie i sposobie obsługi wszystkich komponentów decyduje wyłącznie kod biblioteki Swing, nie system operacyjny, w którym są wyświetlane. W efekcie każdy z tych komponentów działa tak samo niezależnie od platformy.
Biblioteka Swing obsługuje **dołączany wygląd i sposób obsługi** (ang. *pluggable look and feel* —PLAF). Ponieważ każdy komponent tej biblioteki jest renderowany przez kod Javy, nie przez kod rdzennego odpowiednika danej kontrolki, biblioteka Swing zachowuje pełną kontrolę nad wyglądem i sposobem obsługi tego komponentu. Oznacza to, że

istnieje możliwość oddzielenia wyglądu i sposobu obsługi komponentu od jego logiki — właśnie tak działa biblioteka Swing. Oddzielenie wyglądu i sposobu obsługi ma wiele zalet. Jedną z nich jest możliwość modyfikowania sposobu renderowania komponentu bez wpływu na pozostałe aspekty jego funkcjonowania. Innymi słowy, istnieje możliwość „dołączenia" nowego wyglądu i sposobu obsługi wybranego komponentu bez ryzyka wystąpienia jakichkolwiek skutków ubocznych w kodzie, który używa tego komponentu. Co więcej, zastosowany model umożliwia definiowanie całych zbiorów reguł opisujących wygląd i sposób obsługi właściwy różnym stylom graficznego interfejsu użytkownika. Aby użyć określonego stylu, wystarczy „dołączyć" właściwy wygląd i sposób obsługi. Od tego momentu wszystkie komponenty będą automatycznie renderowane przy użyciu tego stylu.

Dołączany wygląd i sposób obsługi ma wiele istotnych zalet. Istnieje możliwość zdefiniowania wyglądu i sposobu obsługi gwarantujących spójność na wszystkich platformach. I odwrotnie, można utworzyć wygląd i sposób obsługi, które będą przypominały wybraną platformę. Jeśli na przykład wiadomo, że aplikacja będzie używana tylko w systemach Windows, można zdefiniować wygląd i sposób obsługi przypominające graficzny interfejs użytkownika, który obowiązuje w tych systemach.
Programista może też zaprojektować własne, niestandardowe wygląd i sposób obsługi. I wreszcie wygląd i sposób obsługi można dynamicznie zmieniać w czasie wykonywania programu.

Podobieństwo do architektury MVC (*Model-View-Controller)*

Ogólnie na każdy komponent wizualny składają się trzy odrębne aspekty:

- wygląd komponentu uzyskiwany w wyniku jego renderowania na ekranie;
- sposób reagowania komponentu na czynności użytkownika;
- informacje o stanie komponentu.

Niezależnie od architektury używanej do implementacji komponentu każdy komponent musi obejmować te trzy elementy. Lata doświadczeń dowiodły, że jedna z tych architektur jest szczególnie efektywna — architektura **model-widok-kontroler** (ang. *Model-View-Controller — MVC*).
O skuteczności architektury MVC decyduje istnienie odrębnych fragmentów projektu odpowiadających trzem wymienionym powyżej aspektom komponentu. W terminologii MVC

- **model** reprezentuje informacje o stanie komponentu. Na przykład w przypadku pola wyboru model zawiera pole określające, czy kontrolka pola wyboru jest zaznaczona.
- **Widok** określa sposób wyświetlania komponentu na ekranie, w tym wszelkie skutki bieżącego stanu (reprezentowanego w modelu),które wpływają na sposób prezentacji komponentu.
- **Kontroler** określa sposób reagowania komponentu na czynności użytkownika. Kiedy użytkownika klika na przykład pole wyboru, kontroler powinien zareagować, tak zmieniając model, aby odzwierciedlał wybór użytkownika (zaznaczenie lub usunięcie zaznaczenia). Zmiana musi oczywiście skutkować stosowną aktualizacją widoku.

Podział komponentu na model, widok i kontroler umożliwia modyfikowanie implementacji każdego z tych elementów bez wpływu na działanie dwóch pozostałych. Na przykład odmienne implementacje widoku mogą wyświetlać ten sam komponent na wiele różnych sposobów bez wpływu na działanie modelu czy kontrolera.
W bibliotece Swing zastosowano zmodyfikowaną wersję architektury MVC z widokiem i kontrolerem połączonym w ramach jednego logicznego bytu nazwanego **delegacją interfejsu użytkownika** (ang. *UI delegate*). Rozwiązanie przyjęte przez twórców biblioteki Swing bywa więc określane mianem architektury **model-delegacja** (ang. *Model-Delegate*) lub architektury z **odrębnym modelem** (ang. *Separable Model*).

Oznacza to, że chociaż architekturę komponentów biblioteki Swing zbudowano na bazie MVC, w bibliotece Swing nie jest stosowana klasyczna implementacja tej architektury.

Komponenty i kontenery

Graficzny interfejs użytkownika na bazie biblioteki Swing składa się z dwóch podstawowych elementów: **komponentów** i **kontenerów**.
Przytoczony podział ma jednak przede wszystkim charakter koncepcyjny, ponieważ wszystkie kontenery są jednocześnie komponentami. Tym, co naprawdę różni oba te elementy, jest zaplanowany przez programistę cel ich stosowania: Powszechnie uważa się, że **komponent** reprezentuje niezależną kontrolkę wizualną, na przykład przycisk czy suwak.
Kontener zawiera grupę komponentów. Oznacza to, że kontener jest specjalnym rodzajem komponentu zaprojektowanym z myślą o zawieraniu innych komponentów. Co więcej, warunkiem wyświetlenia komponentu jest jego umieszczenie w jakimś kontenerze. Oznacza to, że wszystkie graficzne interfejsy użytkownika na bazie biblioteki Swing muszą obejmować przynajmniej po jednym kontenerze.
Ponieważ kontenery są komponentami, każdy kontener może zawierać inne kontenery. Oznacza to, że biblioteka Swing umożliwia tworzenie tzw. **hierarchii zawierania** — na szczycie tej hierarchii zawsze musi znajdować się **kontener najwyższego poziomu.**

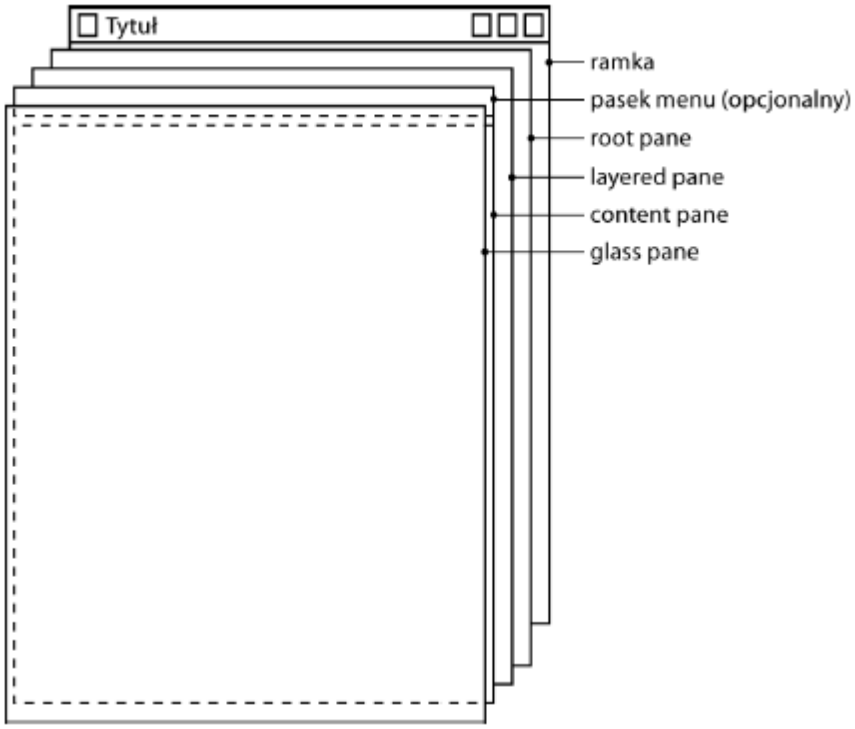
Kontenery

Biblioteka Swing definiuje dwa rodzaje kontenerów. Pierwszy z nich obejmuje tzw. kontenery najwyższego poziomu: **JFrame, JApplet, JWindow i JDialog.** Kontenery z tej grupy nie dziedziczą po klasie **JComponent**. Okazuje się jednak, że dziedziczą po klasach Container i Component należących do zestawu narzędzi AWT. W przeciwieństwie do pozostałych komponentów biblioteki Swing kontenery najwyższego poziomu są komponentami ciężkimi. Kontenery najwyższego poziomu są pod tym względem wyjątkowe w skali całej biblioteki komponentów Swing.
Kontener najwyższego poziomu musi znajdować się na szczycie hierarchii zawierania.
Kontener najwyższego poziomu nie może należeć do żadnego innego kontenera.
Każda hierarchia zawierania musi rozpoczynać się od jakiegoś kontenera najwyższego poziomu. W przypadku aplikacji najczęściej stosuje się kontener **JFrame.** Podczas tworzenia apletów zwykle jest używany kontener **JApplet.**
Drugim rodzajem kontenerów obsługiwanych przez bibliotekę Swing są kontenery lekkie. Wszystkie kontenery lekkie *dziedziczą* po klasie **JComponent** jak naprzykład kontener **JPanel.**
Kontenery lekkie często są używane do organizowania grup powiązanych komponentów i zarządzania tymi grupami. Takie rozwiązanie jest możliwe, ponieważ kontener lekki może zawierać inne kontenery lekkie. Oznacza to, że możemy wykorzystać kontenery lekkie, na przykład typu JPanel, do tworzenia podgrup powiązanych kontrolek zawartych w jakimś kontenerze zewnętrznym.

Panele kontenerów najwyższego poziomu

Każdy kontener najwyższego poziomu definiuje zbiór paneli. Na szczycie tej hierarchii znajduje się obiekt typu JRootPane. JRootPane to kontener lekki, który odpowiada za zarządzanie pozostałymi panelami. Kontener dodatkowo ułatwia zarządzanie opcjonalnym paskiem menu. Na panel najwyższego poziomu składają się trzy panele: **panel szklany**, **panel treści** oraz **panel wielowarstwowy.**
Szklany panel znajduje się na najwyższym poziomie. Jest umieszczany ponad pozostałymi panelami i całkowicie pokrywa te panele. Szklany panel domyślnie ma postać przezroczystego obiektu typu JPanel. Szklany panel umożliwia programiście zarządzanie zdarzeniami związanymi z myszą, które wpływają na cały kontener (nie na poszczególne kontrolki), oraz na przykład na rysowanie ponad pozostałymi

komponentami. W większości przypadków bezpośrednie odwoływanie się do tego panelu nie jest konieczne, jednak w razie potrzeby warto pamiętać o jego istnieniu.
Panel wielowarstwowy jest obiektem typu JLayeredPane. Ten panel umożliwia nadawanie komponentom wartości reprezentujących głębokość. Właśnie głębokość decyduje o tym, który komponent jest wyświetlany przez innymi. Panel wielowarstwowy zawiera panel treści i opcjonalny pasek menu. Mimo że panele szklany i wielowarstwowy są integralną częścią kontenera najwyższego poziomu i pełnią ważne funkcje, większość udostępnianych przez nie rozwiązań jest wykorzystywana w tle. Elementem, do którego tworzone przez nas aplikacje odwołują się najczęściej, jest panel treści, ponieważ to na nim są umieszczane komponenty wizualne graficznego interfejsu użytkownika. Oznacza to, że kiedy programista dodaje jakiś komponent, na przykład przycisk, do kontenera najwyższego poziomu, w rzeczywistości umieszcza go na panelu treści. Panel treści domyślnie ma postać nieprzezroczystego obiektu typu JPanel.



Przykładowe programy tworzone z wykorzystaniem biblioteki Swing
Cechą charakterystyczną biblioteki Swing jest to że okno jest tworzone w osobnym wątku. Wątek ten jest tworzony przez następujące prawie równoważne konstrukcje:

```
// Tworzy ramkę w wątku wątku dystrybucji zdarzeń (ang. event dispatch thread)
EventQueue.invokeLater(new Runnable()
{
public void run()
{
Okno frame = new Okno();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
});

// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new Okienko_J1();
}
});
```

Umieszczanie komponentów podstawowych komponentów w oknie przy użyciu dwóch różnych metod tworzenia wątków.

```
// Prosta aplikacja na bazie biblioteki Swing.
import javax.swing.*;
class Okienko_J1 {
      Okienko_J1() {
// Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Prosta aplikacja na bazie biblioteki Swing");
// Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);
// Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Tworzy etykietę tekstową.
JLabel jlab = new JLabel("Swing to rozbudowane interfejsy GUI.");
// Dodaje etykietę do panelu treści.
jfrm.add(jlab);
// Wyświetla ramkę.
jfrm.setVisible(true);
}
public static void main(String args[]) {
// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new Okienko_J1();
}
});
}
}
```

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

public class Okienko_J2
{
public static void main(String[] args)
{
      // Tworzy ramkę w wątku wątku dystrybucji zdarzeń (ang. event dispatch
thread)
EventQueue.invokeLater(new Runnable()
{
public void run()
{
Okno frame = new Okno();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
});
}
}
class Okno extends JFrame
{
private static final int DEFAULT_WIDTH = 300;
private static final int DEFAULT_HEIGHT = 200;
public Okno()
{
setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
JLabel jlab = new JLabel("Swing to rozbudowane interfejsy GUI.");
//Dodaje etykietę do panelu treści.
add(jlab);
}
}
```

Umieszczenie menedżera komponentów i przycisków z obsługą zdarzeń.

```
 // Aplikacja z komponentami i obsługa zdarzeń na bazie biblioteki Swing.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Okienko_JK {
       Okienko_JK() {
// Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, komponentów i zdarzeń w
bibliotece Swing");
//Wskazuje obiekt FlowLayout jako menedżera układu graficznego.
jfrm.setLayout(new FlowLayout());
// Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);

// Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Tworzy etykietę tekstową.
JLabel jlab = new JLabel("Nowy Layout - FlowLayout i przyciski");
//Tworzy dwa przyciski.
JButton jbtnAlpha = new JButton("Alfa");
JButton jbtnBeta = new JButton("Beta");
//Dodaje obiekt nasłuchujący do przycisku Alfa.
jbtnAlpha.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
jlab.setText("Naciśnięto przycisk Alfa.");
}
});
//Dodaje obiekt nasłuchujący do przycisku Beta.
jbtnBeta.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
jlab.setText("Naciśnięto przycisk Beta.");
}
});
// Dodaje etykietę do panelu treści.
jfrm.add(jlab);
//Dodaje oba przyciski do panelu treści.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);



// Wyświetla ramkę.
jfrm.setVisible(true);
}
public static void main(String args[]) {
// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new Okienko_JK();
}
});
}
}
```

Komponenty
Klasa JTextField

JTextField jest najprostszym komponentem tekstowym w bibliotece Swing. Jest to jednocześnie jeden z najczęściej stosowanych komponentów tekstowych. Kontrolka JTextField umożliwia edycję pojedynczego wiersza tekstu. Klasa JTextField dziedziczy po klasie JTextComponent, która udostępnia podstawowe funkcje wspólne dla wszystkich komponentów tekstowych biblioteki Swing. Komponent JTextField używa interfejsu Document w roli swojego modelu.
Poniżej przedstawiono trzy spośród konstruktorów klasy JTextField:
JTextField(int *cols*)
JTextField(String *str*, int *cols*)
JTextField(String *str*)
Parametr *str* reprezentuje łańcuch początkowo wyświetlany w polu tekstowym, zaś parametr *cols* określa liczbę kolumn tego pola. Jeśli nie zostanie przekazany żaden łańcuch, pole tekstowe początkowo będzie puste. Jeśli nie zostanie określona liczba kolumn, szerokość pola tekstowego zostanie dopasowana do przekazanego łańcucha.
Komponent JTextField generuje zdarzenia w odpowiedzi na czynności użytkownika. Na przykład naciśnięcie klawisza *Enter* powoduje wygenerowanie zdarzenia ActionEvent. Każda zmiana położenia karetki (kursora) powoduje wygenerowanie zdarzenia CaretEvent. (Zdarzenie CaretEvent zdefiniowano w pakiecie javax.swing.event). Pola tekstowe mogą generować także inne zdarzenia.
Pobranie tekstu zapisanego w polu tekstowym realizujemy poprzez metodę getText().
Przykładowy kod programu:

```
// Przykład użycia komponentu JTextField.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Kontrolka_JText {
      Kontrolka_JText() {
// Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, pola tekstowegi i zdarzeń w
bibliotece Swing");
//Wskazuje obiekt FlowLayout jako menedżera układu graficznego.
jfrm.setLayout(new FlowLayout());
// Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);
// Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Tworzy etykietę tekstową.
JLabel jlab = new JLabel("Wartośc domyślna");
//Tworzy pole tekstowe .
JTextField jtf  = new JTextField(15);

// Dodaje etykietę do panelu treści.
jfrm.add(jlab);
//Dodaje pole tekstowe do panelu treści.
jfrm.add(jtf);
jtf.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
//Po naciśnięciu klawisza Enter wyświetla wpisany tekst.
jlab.setText(jtf.getText());
}
});
// Wyświetla ramkę.
jfrm.setVisible(true);
}
public static void main(String args[]) {
// Tworzy ramkę w wątku rozdzielającym zdarzenia.
```

```
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new Kontrolka_JText();
}
});
}
}
```

Klasa JButton

Klasa JButton definiuje przycisk. Klasa JButton umożliwia przypisanie do przycisku ikony, łańcucha lub obu tych elementów jednocześnie. Konstruktory wyglądają następująco:
JButton(Icon *icon*)
JButton(String *str*)
JButton(String *str*, Icon *icon*)
Parametry *str* i *icon* reprezentują odpowiednio łańcuch i ikonę przycisku.
Naciśnięcie przycisku powoduje wygenerowanie zdarzenia ActionEvent. Za pośrednictwem obiektu klasy ActionEvent przekazanego na wejściu metody actionPerformed() zarejestrowanego obiektu nasłuchującego ActionListener można uzyskać **łańcuch polecenia akcji** powiązany z danym przyciskiem.
Łańcuch polecenia domyślnie jest taki sam jak wyświetlana etykieta przycisku. Okazuje się jednak, że łańcuch polecenia można zmienić za pomocą setActionCommand() wywołanej dla obiektu przycisku. Łańcuch polecenia akcji można odczytać za pomocą metody getActionCommand() obiektu zdarzenia.
Deklarację tej metody pokazano poniżej:
String getActionCommand()
Polecenie akcji identyfikuje przycisk. Oznacza to, że jeśli jedna aplikacja zawiera co najmniej dwa przyciski, właśnie na podstawie poleceń akcji można łatwo określać, który z tych przycisków został naciśnięty.

```
// Aplikacja z komponentami i obsługa zdarzeń na bazie biblioteki Swing.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Okno_JPrzycisk  implements ActionListener{
      JLabel jlab;
      Okno_JPrzycisk() {
// Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, komponentów i zdarzeń w
bibliotece Swing");
//Wskazuje obiekt FlowLayout jako menedżera układu graficznego.
jfrm.setLayout(new FlowLayout());
// Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);

// Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Tworzy etykietę tekstową.
 jlab = new JLabel("Flagi państw ");
//Tworzy cztery przyciski przyciski.
//Dodaje przyciski do panelu treści.
ImageIcon pl = new ImageIcon("Poland.png");
JButton plb = new JButton(pl);
plb.setActionCommand("Polska");
plb.addActionListener(this);
jfrm.add(plb);

ImageIcon ua = new ImageIcon("Ukraine.png");
JButton uab = new JButton(ua);
uab.setActionCommand("Ukraina");
uab.addActionListener(this);
jfrm.add(uab);
```

```
ImageIcon sk = new ImageIcon("Slovakia.png");
JButton skb = new JButton(sk);
skb.setActionCommand("Słowacja");
skb.addActionListener(this);
jfrm.add(skb);

ImageIcon cz = new ImageIcon("Czech_Republic.png");
JButton czb = new JButton(cz);
czb.setActionCommand("Republika Czeska");
czb.addActionListener(this);
jfrm.add(czb);
// Dodaje etykietę do panelu treści.
jfrm.add(jlab);

// Wyświetla ramkę.
jfrm.setVisible(true);

}
      // Obsługuje zdarzenia związane z przyciskami.
      public void actionPerformed(ActionEvent ae) {
      jlab.setText("Wybrałeś " + ae.getActionCommand());
      }
public static void main(String args[]) {
// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new Okno_JPrzycisk();
}
});
}
}
```

Pola wyboru

Klasa JCheckBox definiujące kontrolkę pola wyboru. Klasa JCheckBox definiuje wiele konstruktorów. W tym punkcie będzie używana następująca wersja:
JCheckBox(String *str*)
Konstruktor tworzy pole wyboru z etykietą przekazaną za pośrednictwem parametru *str*. Pozostałe konstruktory tej klasy umożliwiają dodatkowe określanie początkowego stanu (zaznaczenia) oraz ewentualnej ikony. Zaznaczenie lub usunięcie zaznaczenia pola wyboru powoduje wygenerowanie zdarzenia typu ItemEvent. Aby uzyskać referencję do obiektu klasy JCheckBox, który wygenerował dane zdarzenie, wystarczy w kodzie metody itemStateChanged() (zdefiniowanej w interfejsie ItemListener) wywołać metodę getItem() dla otrzymanego obiektu klasy ItemEvent. Najprostszym sposobem sprawdzenia stanu pola wyboru (tego, czy jest zaznaczony) jest wywołanie metody isSelected() dla obiektu klasy JCheckBox.
Poniższy przykład ilustruje działanie pól wyboru.

```
// Przykład użycia klasy JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Okno_PW  implements ItemListener {

JLabel jlab;
Okno_PW() {
//Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, komponentów i zdarzeń w
bibliotece Swing");
//Zmiana menedżera układu graficznego na FlowLayout.
```

```
jfrm.setLayout(new FlowLayout());

//Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);

//Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Dodaje pola wyboru do panelu treści.
JCheckBox cb = new JCheckBox("C");
cb.addItemListener(this);
jfrm.add(cb);
cb = new JCheckBox("C++");
cb.addItemListener(this);
jfrm.add(cb);
cb = new JCheckBox("Java");
cb.addItemListener(this);
jfrm.add(cb);
cb = new JCheckBox("Perl");
cb.addItemListener(this);
jfrm.add(cb);
//Tworzy etykietę i dodaje ją do panelu treści.
jlab = new JLabel("Wybierz języki");
jfrm.add(jlab);
//Wyświetla ramkę.
jfrm.setVisible(true);
}

public static void main(String args[]) {
try {// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
      new Okno_PW();
}
}
);
} catch (Exception exc) {
System.out.println("Nie można utworzyć GUI z powodu wyjątku " + exc);
}
}


// Obsługuje zdarzenia ItemEvent dla pól wyboru.
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
if(cb.isSelected())
jlab.setText("Pole " + cb.getText() + " jest zaznaczone.");
else
jlab.setText("Pole " + cb.getText() + " nie jest zaznaczone.");
}
}
```

Przyciski opcji

Przyciski opcji to grupa wzajemnie wykluczających się przycisków, z których tylko jeden może być jednocześnie zaznaczony. Przyciski opcji są obsługiwane przez klasę JRadioButton. Klasa JRadioButton udostępnia wiele konstruktorów. W przykładzie prezentowanym w tym punkcie zostanie użyta następująca wersja konstruktora tej klasy:
JRadioButton(String *str*)
Parametr *str* reprezentuje etykietę tworzonego przycisku opcji. Pozostałe konstruktory dodatkowo umożliwiają określanie stanu początkowego przycisku oraz przekazanie ewentualnej ikony.
Aby aktywować wzajemne wykluczanie przycisków opcji, należy skonfigurować grupę łączącą te przyciski. Jednocześnie może być zaznaczony tylko jeden przycisk opcji w grupie.
Grupę przycisków można utworzyć za pomocą klasy ButtonGroup.
Aby utworzyć grupę, wystarczy wywołać domyślny konstruktor tej klasy. Do dodawania elementów do grupy przycisków służy następująca metoda:
void add(AbstractButton *ab*)
Parametr *ab* reprezentuje referencję do przycisku dodawanego do grupy.
Klasa JRadioButton generuje zdarzenia akcji, zdarzenia elementów i zdarzenia zmian w odpowiedzi na każde zaznaczenie lub usunięcie zaznaczenia. W większości przypadków obsługuje się tylko zdarzenia akcji, zatem zwykle programista musi zaimplementować tylko interfejs ActionListener.
Kliknięty przycisk można zidentyfikować na wiele różnych sposobów. W pierwszym kroku należy uzyskać polecenie akcji za pomocą metody getActionCommand(). Polecenie akcji domyślnie jest takie samo jak etykieta przycisku, ale można je zmienić, wywołując metodę setActionCommand() dla obiektu przycisku opcji. Istnieje też możliwość uzyskania referencji do przycisku, który wygenerował zdarzenie — wystarczy wywołać metodę getSource() dla obiektu zdarzenia typu ActionEvent.
Oprócz tego programista może użyć metody isSelected() do sprawdzenia stanu poszczególnych przycisków opcji, aby zidentyfikować aktualnie zaznaczony przycisk.

```
// Przykład użycia klasy JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Okno_JPO  implements ActionListener {

JLabel jlab;

Okno_JPO() {
//Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, komponentów i zdarzeń w
bibliotece Swing");
//Zmiana menedżera układu graficznego na FlowLayout.
jfrm.setLayout(new FlowLayout());

//Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);

//Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Tworzy przyciski opcji i dodaje je do panelu treści.
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
jfrm.add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
```

```
jfrm.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
jfrm.add(b3);
//Definiuje grupę przycisków.
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);

//Tworzy etykietę i dodaje ją do panelu treści.
jlab = new JLabel("Wybierz opcję");
jfrm.add(jlab);
//Wyświetla ramkę.
jfrm.setVisible(true);
}

public static void main(String args[]) {
try {// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
      new Okno_JPO();
}
}
);
} catch (Exception exc) {
System.out.println("Nie można utworzyć GUI z powodu wyjątku " + exc);
}
}
```

+

```
//Obsługuje wybór przycisku opcji.
public void actionPerformed(ActionEvent ae) {
jlab.setText("Wybrałeś opcję " + ae.getActionCommand());
}
}
```

Klasa JTabbedPane

Klasa JTabbedPane reprezentuje **panel podzielony na zakładki**. Taki panel zarządza zestawem swoich komponentów, kojarząc je z kilkoma odrębnymi zakładkami. Wybór zakładki powoduje wyświetlenie na pierwszym planie komponentu powiązanego z tą zakładką. Panele podzielone na zakładki cieszą się sporą popularnością we współczesnych graficznych interfejsach użytkownika i jako takie są doskonale znane użytkownikom. Zważywszy na złożony charakter takich okien, ich tworzenie i stosowanie w Javie jest zadziwiająco proste.
Klasa JTabbedPane definiuje trzy konstruktory. W tym punkcie będzie stosowany konstruktor domyślny, który tworzy pustą kontrolkę z zakładkami widocznymi w górnej części panelu. Dwa pozostałe konstruktory umożliwiają dodatkowe określenie położenia zakładek (można je wyświetlać przy wszystkich czterech krawędziach panelu). Klasa JTabbedPane korzysta z modelu SingleSelectionModel.
Do dodawania zakładek służy metoda addTab(), której ogólną postać pokazano poniżej:
void addTab(String *name*, Component *comp*)
Parametr *name* oznacza tytuł zakładki, a *comp* — komponent, który ma zostać do niej dodany.
Do zakładek często dodaje się komponent typu JPanel, który zawiera grupę powiązanych ze sobą komponentów. Takie rozwiązanie umożliwia umieszczanie całych zbiorów komponentów w zakładkach.
Ogólna procedura tworzenia panelu z zakładkami jest następująca:
1. Utworzenie obiektu klasy JTabbedPane.
2. Dodanie zakładki za pomocą metody addTab().
3. Dodanie panelu z zakładkami do panelu treści.

```
// Przykład użycia klasy JTabbedPane.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Okno_JZakladki   {

JLabel jlab;

Okno_JZakladki() {
//Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, komponentów i zdarzeń w
bibliotece Swing");
//Zmiana menedżera układu graficznego na FlowLayout.
jfrm.setLayout(new FlowLayout());

//Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);

//Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Miasta", new CitiesPanel());
jtp.addTab("Kolory", new ColorsPanel());
jtp.addTab("Smaki", new FlavorsPanel());
jfrm.add(jtp);



//Wyświetla ramkę.
jfrm.setVisible(true);
}
```

```
public static void main(String args[]) {
try {// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
      new Okno_JZakladki();
}
}
);
} catch (Exception exc) {
System.out.println("Nie można utworzyć GUI z powodu wyjątku " + exc);
}
}


//Tworzy panele, które zostaną umieszczone na panelu podzielonym na zakładki.
class CitiesPanel extends JPanel {
public CitiesPanel() {
JButton b1 = new JButton("Nowy Jork");
add(b1);
JButton b2 = new JButton("Londyn");
add(b2);
JButton b3 = new JButton("Hong Kong");
add(b3);
JButton b4 = new JButton("Tokio");
add(b4);
}
}
class ColorsPanel extends JPanel {
public ColorsPanel() {
JCheckBox cb1 = new JCheckBox("Czerwony");
add(cb1);
JCheckBox cb2 = new JCheckBox("Zielony");
add(cb2);
JCheckBox cb3 = new JCheckBox("Niebieski");
add(cb3);
}
}
class FlavorsPanel extends JPanel {
public FlavorsPanel() {
JComboBox jcb = new JComboBox();
jcb.addItem("Waniliowy");
jcb.addItem("Czekoladowy");
jcb.addItem("Truskawkowy");
add(jcb);
}
}
}
```

Klasa JList

Podstawową klasą list biblioteki Swing jest JList. Klasa JList obsługuje wybór co najmniej jednego elementu wyświetlanej listy. Mimo że w zdecydowanej większości przypadków lista składa się z łańcuchów, istnieje możliwość utworzenia listy obejmującej dowolne obiekty, pod warunkiem że można je wyświetlić.
W wydaniu JDK 7 zdecydowano się jednak przekształcić klasę JList w typ sparametryzowany (uogólniony), zatem teraz jej deklaracja ma następującą postać:
class JList<E>
Element E reprezentuje typ elementów listy.
Klasa JList udostępnia wiele konstruktorów. Jedna z możliwych wersji:
JList(E[] *items*)
Konstruktor w tej formie tworzy obiekt klasy JList zawierający elementy tablicy przekazanej za pośrednictwem parametru *items*.
Klasę JList zbudowano na bazie dwóch modeli. Pierwszym z nich jest ListModel. Interfejs ListModel definiuje sposób dostępu do danych listy. Drugi model jest reprezentowany przez interfejs ListSelectionModel, który definiuje metody niezbędne do identyfikacji zaznaczonego elementu (lub zaznaczonych elementów).
Komponent JList jest co prawda przystosowany do samodzielnego działania, jednak w większości przypadków jest dodatkowo opakowywany w ramach panelu JScrollPane. Dzięki temu dla dłuższych list automatycznie są stosowane paski przewijania, co znacznie ułatwia proces projektowania graficznego interfejsu użytkownika. Panel z paskami przewijania dodatkowo ułatwia zmianę liczby elementów na liście bez konieczności dostosowywania wymiarów samego komponentu JList.
Komponent JList generuje zdarzenie typu ListSelectionEvent w odpowiedzi na dokonanie lub zmianę wyboru elementu bądź elementów listy. Wspomniane zdarzenie jest generowane także w przypadku usunięcia zaznaczenia jakiegoś elementu listy. Obsługa tego zdarzenia wymaga implementacji interfejsu ListSelectionListener. Interfejs ListSelectionListener definiuje tylko jedną metodę nazwaną valueChanged().
Składnia tej metody pokazano poniżej:
void valueChanged(ListSelectionEvent *le*)
Parametr *le* reprezentuje referencję do obiektu zdarzenia..
Komponent JList domyślnie umożliwia zaznaczanie wielu zakresów elementów na liście, ale można to zachowanie zmienić, wywołując metodę setSelectionMode() (zdefiniowaną w klasie JList).
Składnia tej metody pokazano poniżej:
void setSelectionMode(int *mode*)
Parametr *mode* reprezentuje tryb zaznaczania elementów. Za pośrednictwem tego parametru należy przekazać jedną z wartości zdefiniowanych w interfejsie ListSelectionModel:
SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION
Domyślny tryb zaznaczania wielu zakresów elementów umożliwia użytkownikowi wybór kilku przedziałów. W trybie zaznaczania pojedynczego zakresu użytkownik może wybrać tylko jeden przedział elementów. W trybie pojedynczego wyboru użytkownik może zaznaczyć tylko jeden element. Oczywiście także w dwóch pierwszych trybach istnieje możliwość wyboru jednego elementu. Różnica polega więc tylko na możliwości dodatkowego zaznaczania zakresów.
Indeks pierwszego zaznaczonego elementu (a więc także indeks jedynego zaznaczonego elementu w trybie pojedynczego wyboru) można uzyskać za pomocą metody getSelectedIndex(), której sygnaturę przedstawiono poniżej:
int getSelectedIndex()
Indeksowanie elementów rozpoczyna się od zera. Oznacza to, że jeśli użytkownik zaznaczył pierwszy element listy, metoda zwróci wartość 0. Jeśli nie zaznaczono żadnego

elementu, metoda zwraca wartość –1.
Uzyskiwanie indeksów zaznaczonych elementów to nie jedyne rozwiązanie — równie dobrze można uzyskać zaznaczone wartości za pomocą metody getSelectedValue():
E getSelectedValue()
Metoda zwraca referencję do pierwszej zaznaczonej wartości. Jeśli nie zaznaczono żadnej wartości, metoda zwraca wartość null.
Przykład kodu:

```
// Przykład użycia klasy JList.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Okno_JLista   {

       JList<String> jlst;
       JLabel jlab;
       JScrollPane jscrlp;
       // Tworzy tablicę z nazwami miast.
       String Cities[] = { "Nowy Jork", "Chicago", "Houston",
       "Denver", "Los Angeles", "Seattle",
       "Londyn", "Paryż", "Nowe Delhi",
       "Hongkong", "Tokio", "Sydney" };

Okno_JLista() {
//Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, komponentów i zdarzeń w
bibliotece Swing");
//Zmiana menedżera układu graficznego na FlowLayout.
jfrm.setLayout(new FlowLayout());

//Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);

//Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


//Tworzy komponent typu JList.
jlst = new JList<String>(Cities);
//Ustawia tryb jednokrotnego wyboru.
jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
//Dodaje listę do panelu z paskami przewijania.
jscrlp = new JScrollPane(jlst);
//Ustawia preferowane wymiary panelu z przewijaniem.
jscrlp.setPreferredSize(new Dimension(120, 90));
//Tworzy etykietę wyświetlającą nazwę wybranego miasta.
jlab = new JLabel("Wybierz miasto");
//Dodaje obiekt nasłuchujący zdarzeń wyboru generowanych przez komponent listy.
jlst.addListSelectionListener(new ListSelectionListener() {
public void valueChanged(ListSelectionEvent le) {
//Uzyskuje indeks zmienionego elementu.
int idx = jlst.getSelectedIndex();
//Wyświetla wybrany element (jeśli został zaznaczony).
if(idx != -1)
jlab.setText("Bieżący wybór: " + Cities[idx]);
else // W przeciwnym razie ponownie prosi o wybór miasta.
jlab.setText("Wybierz miasto");
}
});
//Dodaje listę i etykietę do panelu treści.
```

```
jfrm.add(jscrlp);
jfrm.add(jlab);
//Wyświetla ramkę.
jfrm.setVisible(true);
}

public static void main(String args[]) {
try {// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
      new Okno_JLista();
}
}
);
} catch (Exception exc) {
System.out.println("Nie można utworzyć GUI z powodu wyjątku " + exc);
}
}



}
```

Klasa JComboBox

Pakiet Swing umożliwia stosowanie tzw. **listy kombinowanej** (połączenia pola tekstowego z listą rozwijaną), zdefiniowanej w klasie JComboBox, która poszerza klasę JComponent. Taka lista normalnie wyświetla tylko jedną pozycję. Może jednak również wyświetlić listę rozwijaną, pozwalającą na wybranie innej. Istnieje również możliwość wpisania wybranej opcji bezpośrednio w polu tekstowym.
Zaczynając od wydania JDK 7, klasa JComboBox jest typem sparametryzowanym. Klasa JComboBox jest teraz deklarowana w następujący sposób:
class JComboBox<E>
Element E reprezentuje typ elementów na liście kombinowanej.
W tym punkcie będzie stosowany następujący konstruktor klasy JComboBox:
JComboBox(E[] *items*)
Parametr *items* reprezentuje tablicę elementów, które zostaną umieszczone na tworzonej liście. Klasa JComboBox definiuje też inne konstruktory.
Komponent JComboBox używa interfejsu ComboBoxModel w roli swojego modelu. Zmienne listy kombinowane (czyli takie, których elementy mogą być modyfikowane) stosują model MutableComboBoxModel.
Oprócz przekazywania tablicy elementów, które mają być wyświetlane na liście rozwijanej, istnieje możliwość dynamicznego dodawania elementów za pomocą metody addItem(), której sygnaturę przedstawiono poniżej:
void addItem(E *obj*)
Parametr *obj* reprezentuje obiekt, który ma zostać dodany do listy. Metody addItem() można używać tylko dla zmiennych list kombinowanych.
Komponent JComboBox generuje zdarzenie akcji w momencie wybrania przez użytkownika elementu listy. Komponent JComboBox generuje też zdarzenie elementu w odpowiedzi na zmianę stanu zaznaczenia, czyli zaznaczenie lub usunięcie zaznaczenia jakiegoś elementu. Oznacza to, że każda zmiana zaznaczenia powoduje wygenerowanie dwóch zdarzeń: jednego dla elementu, którego zaznaczenie usunięto, i jednego dla nowo zaznaczonego elementu. W wielu przypadkach wystarczy nasłuchiwanie zdarzeń akcji, jednak warto pamiętać o dostępności obu rodzajów zdarzeń.
Jednym ze sposobów uzyskiwania elementu wybranego z listy jest wywołanie metody getSelectedItem() dla obiektu tej listy. Oto sygnatura tej metody:
Object getSelectedItem()
Obiekt zwrócony przez tę metodę należy rzutować na typ obiektów przechowywanych na liście.
Przykład kodu:

```
// Przykład użycia klasy JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;

public class Okno_JCBox   {
       JLabel jlab;
       ImageIcon Polska, Ukraina, Słowacja, Republika_Czeska;
       JComboBox<String> jcb;
       // Tworzy tablicę z nazwami miast.
       String Panstwa[] = { "Polska", "Ukraina", "Słowacja",
       "Republika Czeska" };
       Hashtable<String,String> images = new Hashtable<String,String>();

Okno_JCBox() {
//Tworzy nowy kontener typu JFrame.
JFrame jfrm = new JFrame("Obsługa menedżera okien, komponentów i zdarzeń w
bibliotece Swing");
//Zmiana menedżera układu graficznego na FlowLayout.
```

```
jfrm.setLayout(new FlowLayout());

//Określa początkowe wymiary ramki.
jfrm.setSize(300, 200);

//Kończy program w momencie zamknięcia aplikacji przez użytkownika.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


//Tworzy obiekt listy kombinowanej i dodaje ten komponent do panelu treści.
jcb = new JComboBox<String>(Panstwa);
jfrm.add(jcb);

//Obsługuje wybór elementu listy.
jcb.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
String s = (String) jcb.getSelectedItem();
jlab.setIcon(new ImageIcon(images.get(s) + ".png"));
}
});
//Tworzy etykietę i dodaje ją do panelu treści.
jlab = new JLabel(new ImageIcon("Poland.png"));
jfrm.add(jlab);
//Kojarzy nazwy opcji listy z nazwami plików z obrazkami.
images.put("Polska", "Poland");
images.put("Ukraina", "Ukraine");
images.put("Słowacja", "Slovakia");
images.put("Republika Czeska","Czech_Republic");

//Wyświetla ramkę.
jfrm.setVisible(true);
}
+

public static void main(String args[]) {
try {// Tworzy ramkę w wątku rozdzielającym zdarzenia.
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
      new Okno_JCBox();
}
}
);
} catch (Exception exc) {
System.out.println("Nie można utworzyć GUI z powodu wyjątku " + exc);
}
}



}
```

Opracowano na podstawie:
HERBERT SCHILDT - „Java. Kompendium programisty. Wydanie IX” Wydawnictwo Helion.