Pierwszy program korzystający z klas pakietu Swing

Najprostszym chyba programem korzystającym z klas pakietu Swing jest HelloWorldSwing program wyświetlający komentarz w okienku:

Źródło HelloWorldSwing zawiera poniższy kod:

```
import javax.swing.*;

public class HelloWorldSwing {

   private static void createAndShowGUI() {
      // Ustawienie wyglądu
      JFrame.setDefaultLookAndFeelDecorated(true);

      // Utworzenie ramki oraz przypisanie zachowania pod przycisk zamykania okna
      //(Java >=1.3)
      JFrame frame = new JFrame("HelloWorldSwing");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     // alternatywnie do powyższej linii można zapisać co następuje (Java < 1.3)
     // frame.addWindowListener(new WindowAdapter() {
     //    public void windowClosing(WindowEvent e) { System.exit(0); }});

      //  Utworzenie etykiety "Hello World" i umieszczenie jej w ramce
      JLabel label = new JLabel("Hello World");
      frame.getContentPane().add(label);

      // Wyświetlenie ramki
      frame.pack();
      frame.setVisible(true);
   }

   public static void main(String[] args) {
      // Aby zabezpieczyć się przed kłopotami w wielowątkowej aplikacji
      // zadanie utworzenia i wyświetlenia ramki przekazane zostanie
      // do wątku: event-dispatching thread

      javax.swing.SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            createAndShowGUI();
         }
      });
   }
}
```

W powyższym przykładzie zaimplementowano dwie podstawowe operacje, które każda aplikacja korzystająca z pakietu klas Swing powinna posiadać:

- Importowanie pakietu.
- Stworzenie i zainicjalizowanie kontenera.
- Wyświetlenie kontenera
- Zapewnienie bezpieczeństwa wielowątkowowego

Pierwsza linia programu importuje klasy z pakietu Swing:

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

Często zdarza się, że oprócz pakietu Swing potrzebne są w aplikacji odpowiednie klasy z pakietu AWT. Importowanie tych klas odbywa się następująco:

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

Import ten jest konieczny z racji, iż komponenty klasy Swing używają infrastruktur AWT, w tym korzystają z zaimplementowanego tam modelu zdarzeń AWT. Model zdarzeń definiuje jak komponenty klasy reagują na takie zdarzenia jak, np. kliknięcie myszką lub jej przesuwanie.

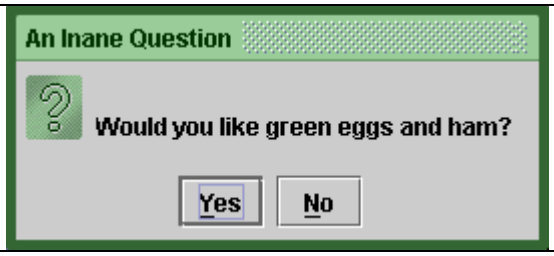
Komponenty Swing.

Komponenty Swing tworzą strukturę klas, której klasą bazową (poza komponentami najwyższego poziomu) jest klasa JComponent. Klasy bezpośrednio dziedziczące z tej klasy to:
AbstractButton, BasicInternalFrameTitlePane, Box, Box.Filler, JColorChooser, JComboBox, JFileChooser, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSpinner, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, JViewport

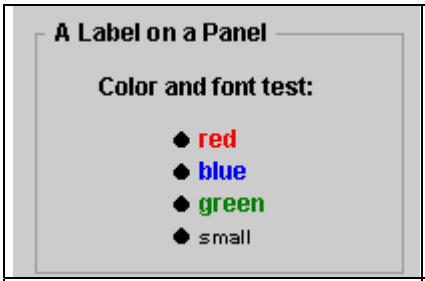
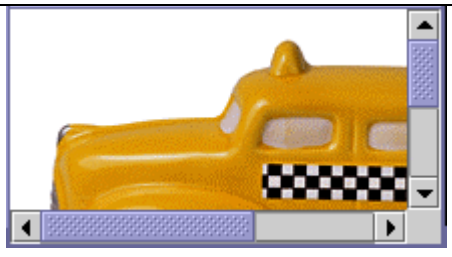
Kontenery bazowe (top-level).

Każdy program z graficznym interfejsem użytkownika bazującym na klasach Swing musi zawierać co najmniej jeden kontener bazowy. Kontener taki umożliwia innym komponentom Swing odmalowywanie i realizację obsługi zdarzeń. Kontenerami tego typu są JFrame, JDialog, oraz JApplet (dla apletów). Każdy obiekt klasy JFrame implementuje główne okno, każdy obiekt klasy JDialog implementuje okno pochodne (okno, które jest zależne od innego okna). Każdy obiekt klasy JApplet implementuje pole wewnątrz okna przeglądarki.

Jedynym kontenerem bazowym w przykładzie HelloWorldSwing jest JFrame. Instancje tej klasy, tj. ramka, wygląda na ekranie jak okienko ze wszystkimi atrybutami: granicą (obwoluta), tytułem, przyciskami do zamiany w ikonę, rozciągnięcia i do zamykania okna.

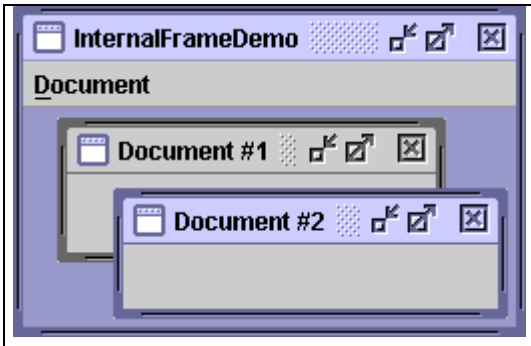
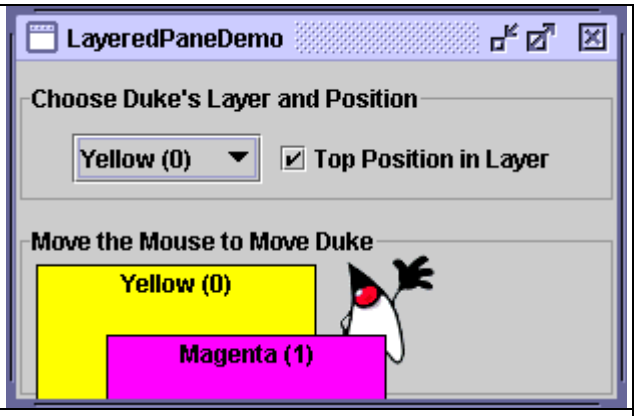
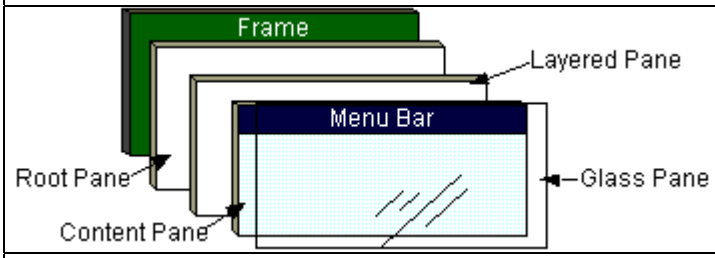
		
Applet	Dialog	Frame

Kontenery ogólnego zastosowania.

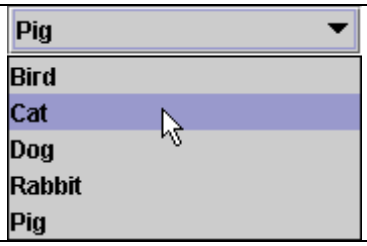
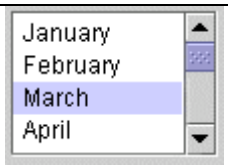
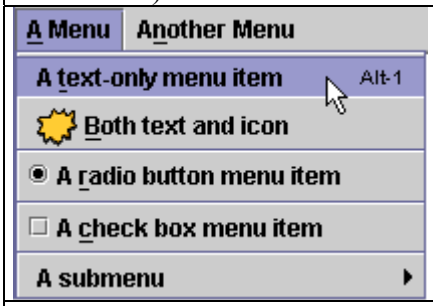
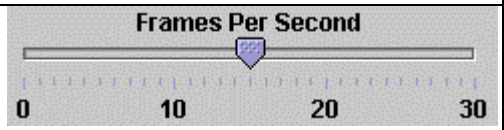
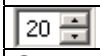
		
Panel	ScrollPane	SplitPane
		

TabbedPane	ToolBar	

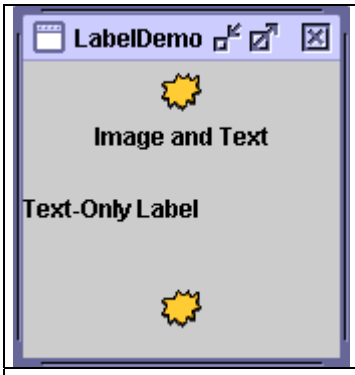
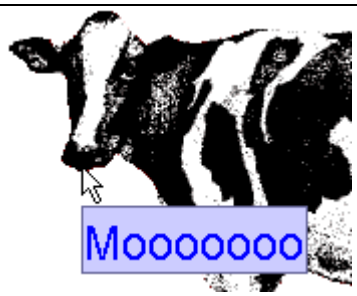
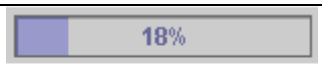
Kontenery specjalnego zastosowania.

	
InternalFrame	Layered pane (LayeredPane, DesktopPane)
	
RootPane (nie jest tworzony, lecz uzyskiwany)	

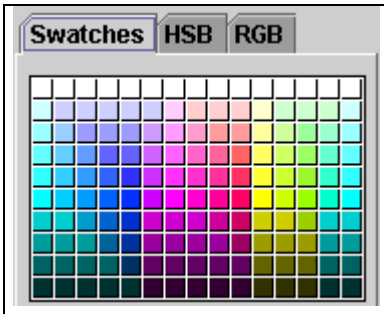
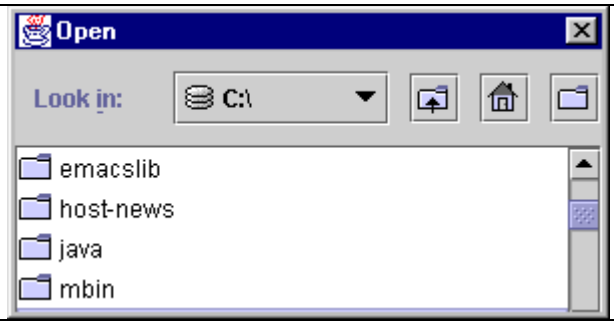
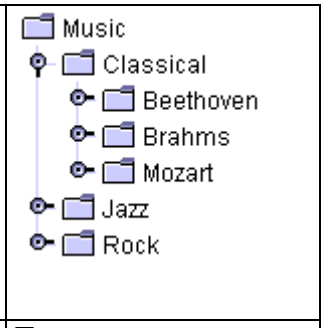
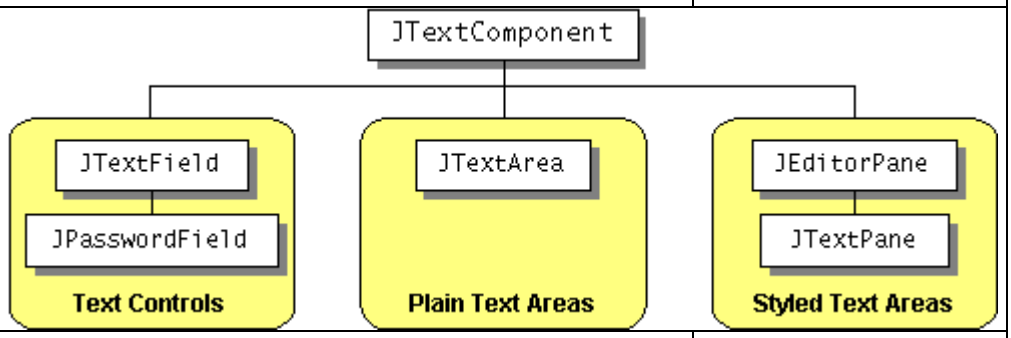
Podstawowe kontrolki.

		
Przyciski (Button, CheckBox, RadioBox)	ComboBox	List
		
Menu	Slider	TextField lub FormattedTextField
		
Spinner		

Nieedytowalne komponenty niosące informacje

		
Label	ToolTip	ProgressBar

Edytowalne komponenty niosące informacje w postaci sformatowanej

		
ColorChooser	FileChooser	Tree
		
Tekst (zobacz obok)		
		
Table		

Obsługa zdarzeń

Komponenty Swing generują zdarzenia. Generowane są one za każdym razem, kiedy użytkownik używa klawiatury bądź myszki. Każdy obiekt może być informowany o takich zdarzeniach. Jedyne, co trzeba w tym kierunku zrobić, to wyposażyć obiekt w odpowiedni interfejs i zarejestrować go jako słuchacza zdarzeń pochodzących od konkretnego źródła.

Implementacja obsługi zdarzeń

Klasę, której instancje obsługiwać będą zdarzenia (klasę słuchacza zdarzeń) można zaimplementować na dwa sposoby:

1. Deklarując klasę, która implementuje odpowiedni do nasłuchiwanych zdarzeń interfejs
2. Deklarując klasę, która dziedziczy po klasie implementującej odpowiedni interfejs (i/lub przysłaniającej metody tego interfejsu).

public class MyClass extends WindowAdapter{
public void actionPerformed(ActionEvent e) {
// kod obsługi zdarzenia w przysłoniętej
// metodzie
}
}
	public class MyClass implements ActionListener {
	public void actionPerformed(ActionEvent e) {
	// kod obsługi zdarzenia w metodzie
	// zaimplementowanej
	}
	}

Aby zarejestrować słuchacza zdarzeń pochodzących od konkretnego komponentu (np. sc), należy wywołać metodę add*NazwaInterfejsu* tego komponentu. *NazwaInterfejsu* odpowiadać ma nazwie implementowanego przez słuchacza interfejsu (typowi słuchacza).

sc.addWinowListener(instanceOfMyClass);	sc.addActionListener(instanceOfMyClass);

Słuchaczy można również implementować jako klasy anonimowe:

button.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
numClicks++;
label.setText(labelPrefix + numClicks);
}
});
	

Dla implementacji jak wyżej kliknięcie podczas wykonywania programu na przycisku button spowoduje wygenerowanie zdarzenia, które obsłużone zostanie w metodzie actionPerformed. Jedyny argument tej metody, ActionEvent e, zawierać będzie wtedy informacje o źródle zdarzenia.

Tabela przykładowych zdarzeń i interfejsów ich słuchaczy:

Działanie wywołujące zdarzenie	**Typ słuchacza**
Kliknięcie na przycisku, Enter podczas pisania w polu tekstowym, wybór pozycji z menu	ActionListener
Zamknięcie ramki (okna głównego)	WindowListener
Naciśnięcie na klawisz myszki, gdy kursor jest nad komponentem	MouseListener
Przesunięcie kursorem myszki ponad komponentem	MouseMotionListener
Poruszanie kółkiem myszki ponad komponentem	MouseWheelListener
Zmiana rozmiaru, położenia, widzialności komponentu	ComponentListener
Otrzymanie lub utrata fokusu z klawiatury	FocusListener
Zmiana selekcji w tabeli lub liście	ListSelectionListener
Naciśnięcie klawisza (przy fokusie)	KeyListener

Uwaga: Obsługa zdarzeń odbywa się w wątku *event-dispatching thread*, podobnie jak odmalowywanie komponentów. Dlatego podczas obsługi zdarzeń wygląd komponentów „zamraża się”.

Zestaw komponentów Swinf wraz ze słuchaczami, które są z nimi kojarzone.

Komponent Swing	**Słuchacz**							
	action	caret	change	document, undoable edit	item	list selection	window	inne
button	✔		✔		✔			
check box	✔		✔		✔			
color chooser			✔					
combo box	✔				✔			
dialog							✔	
editor pane		✔		✔				hyperlink
file chooser	✔							
formatted text field	✔	✔		✔				
frame							✔	
internal frame								internal frame
list						✔		list data
menu								menu
menu item	✔		✔		✔			menu key menu drag mouse
option pane								
password field	✔	✔		✔				
popup menu								popup menu
progress bar			✔					
radio button	✔		✔		✔			
slider			✔					
spinner			✔					
tabbed pane			✔					
table						✔		table model table column model cell editor
text area		✔		✔				
text field	✔	✔		✔				
text pane		✔		✔				hyperlink
toggle button	✔		✔		✔			
tree								tree expansion tree will expand tree model tree selection
viewport (używany przez scrollpane)			✔					

Zestaw słuchaczy z odpowiadającymi im adapterami oraz wyszczególnieniem metod interfejsu.

Listener Interface	Adapter Class	Listener Methods
ActionListener	none	actionPerformed(ActionEvent)
AncestorListener	none	ancestorAdded(AncestorEvent) ancestorMoved(AncestorEvent) ancestorRemoved(AncestorEvent)
CaretListener	none	caretUpdate(CaretEvent)
CellEditorListener	none	editingStopped(ChangeEvent) editingCanceled(ChangeEvent)
ChangeListener	none	stateChanged(ChangeEvent)
ComponentListener	ComponentAdapter	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
ContainerListener	ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
DocumentListener	none	changedUpdate(DocumentEvent) insertUpdate(DocumentEvent) removeUpdate(DocumentEvent)
ExceptionListener (od wersji 1.4)	none	exceptionThrown(Exception)
FocusListener	FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
HierarchyBoundsListener (od wersji 1.3)	HierarchyBoundsAdapter	ancestorMoved(HierarchyEvent) ancestorResized(HierarchyEvent)
HierarchyListener (introduced in 1.3)	none	hierarchyChanged(HierarchyEvent)
HyperlinkListener	none	hyperlinkUpdate(HyperlinkEvent)
InputMethodListener	none	caretPositionChanged(InputMethodEvent) inputMethodTextChanged(InputMethodEvent)
InternalFrameListener	InternalFrameAdapter	internalFrameActivated(InternalFrameEvent) internalFrameClosed(InternalFrameEvent) internalFrameClosing(InternalFrameEvent) internalFrameDeactivated(InternalFrameEvent) internalFrameDeiconified(InternalFrameEvent) internalFrameIconified(InternalFrameEvent) internalFrameOpened(InternalFrameEvent)
ItemListener	none	itemStateChanged(ItemEvent)
KeyListener	KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
ListDataListener	none	contentsChanged(ListDataEvent) intervalAdded(ListDataEvent) intervalRemoved(ListDataEvent)
ListSelectionListener	none	valueChanged(ListSelectionEvent)
MenuDragMouseListener	none	menuDragMouseDragged(MenuDragMouseEvent) menuDragMouseEntered(MenuDragMouseEvent) menuDragMouseExited(MenuDragMouseEvent) menuDragMouseReleased(MenuDragMouseEvent)
MenuKeyListener	none	menuKeyPressed(MenuKeyEvent) menuKeyReleased(MenuKeyEvent) menuKeyTyped(MenuKeyEvent)
MenuListener	none	menuCanceled(MenuEvent) menuDeselected(MenuEvent) menuSelected(MenuEvent)
MouseInputListener (extends MouseListener and MouseMotionListener)	MouseInputAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseDragged(MouseEvent) mouseMoved(MouseEvent)
MouseListener	MouseAdapter, MouseInputAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent)

		mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener	MouseMotionAdapter, MouseInputAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
MouseWheelListener (od wersji 1.4)	none	mouseWheelMoved(MouseWheelEvent)
PopupMenuListener	none	popupMenuCanceled(PopupMenuEvent) popupMenuWillBecomeInvisible(PopupMenuEvent) popupMenuWillBecomeVisible(PopupMenuEvent)
PropertyChangeListener	none	propertyChange(PropertyChangeEvent)
TableColumnModelListener	none	columnAdded(TableColumnModelEvent) columnMoved(TableColumnModelEvent) columnRemoved(TableColumnModelEvent) columnMarginChanged(ChangeEvent) columnSelectionChanged(ListSelectionEvent)
TableModelListener	none	tableChanged(TableModelEvent)
TreeExpansionListener	none	treeCollapsed(TreeExpansionEvent) treeExpanded(TreeExpansionEvent)
TreeModelListener	none	treeNodesChanged(TreeModelEvent) treeNodesInserted(TreeModelEvent) treeNodesRemoved(TreeModelEvent) treeStructureChanged(TreeModelEvent)
TreeSelectionListener	none	valueChanged(TreeSelectionEvent)
TreeWillExpandListener	none	treeWillCollapse(TreeExpansionEvent) treeWillExpand(TreeExpansionEvent)
UndoableEditListener	none	undoableEditHappened(UndoableEditEvent)
VetoableChangeListener	none	vetoableChange(PropertyChangeEvent)
WindowFocusListener (od wersji 1.4)	WindowAdapter	windowGainedFocus(WindowEvent) windowLostFocus(WindowEvent)
WindowListener	WindowAdapter	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)
WindowStateListener (od wersji 1.4)	WindowAdapter	windowStateChanged(WindowEvent)

The Event-Dispatching Thread

Obsługa zdarzeń w klasach Swing oraz kod odświeżania (odmalowywania) wykonywane są w pojedynczym wątku, nazywanym *the event-dispatching thread*. Dzięki temu obsługa zdarzeń może odbywać się w sposób sekwencyjny (program obsługi jakiegoś zdarzenia nie rozpocznie swojego działania, dopóki program obsługujący wcześniejsze zdarzenie się nie zakończy) oraz procedury odmalowywania mogą działać nie będąc przerywane przez zdarzenia.

Aby nie dopuścić do zakleszczeń wątków (deadlock) programista powinien szczególnie zwracać uwagę, aby komponenty i modele Swing były tworzone, modyfikowane i osiągane tylko z wnętrza wątku *event-dispatching thread*. Przesłanie polecenia do wykonania przez wątek *event-dispatching thread* można zrealizować poprzez wywołanie metody invokeLater().

Uwaga: Mówi się, że możliwe jest tworzenie graficznego interfejsu użytkownika w wątku głównym (z metodą main) pod warunkiem, że komponenty już zrealizowane nie są modyfikowane. Realizacja znaczy tu wyświetlenie komponentu na ekranie lub jego przygotowanie do wyświetlenia. Metody setVisible(true) oraz pack() powodują, że okno jest realizowane, co z kolei pociąga za sobą realizację komponentów, które ono zawiera.

Podczas tworzenia aplikacji wielowątkowej można ominąć kilka najczęściej spotykanych niebezpieczeństw implementując wątek z użyciem klasy pomocniczej SwingWorker lub klasy Timer. Obiekt klasy SwingWorker tworzy wątek, aby w nim wykonać operacje czasowo długie operacje. Z chwilą zakończenia

operacji, SwingWorker daje możliwość wykonania dodatkowego fragmentu programu w ramach wątku *event-dispatching thread*. Klasa Timer jest użyteczna w wykonywaniu zadań powtarzających się lub pojawiających się w określonym czasie.

Metody tworzenia wielowątkowych, bezpiecznych programów korzystających z pakietu Swing:

- do uaktualniania komponentów, gdy odpowiedni kod nie jest wewnątrz słuchacza zdarzeń, należy użyć metod klasy SwingUtilities: invokeLater (zalecany) lub invokeAndWait.
- jeśli nie jest pewnym, czy kod wykonywany jest w słuchaczu zdarzeń, należy zanalizować kod programu, aby wyróżnić, która metoda jest wywoływana przez który wątek. Gdy analiza taka nie przyniesie efektu, można wtedy użyć metody SwingUtilities.isEventDispatchThread(), która zwraca prawdę, jeśli rozpatrywany kod jest wykonywany w wątku *event-dispatching thread*. Z dowolnego wątku można bezpiecznie wywoływać invokeLater, zaś z wywołaniem invokeAndWait wiąże się możliwość zgłoszenia przez tą metodę wyjątku, jeśli nie będzie ona wywołana wewnątrz wątku *event-dispatching thread*.
- Jeśli istnieje potrzeba uaktualnienia komponentu po opóźnieniu, (niezależnie od tego, czy kod wykonywany jest aktualnie w słuchaczu zdarzeń, czy też nie) należy użyć klasy Timer z pakietu Swing.
- W przypadku uaktualniania komponentów w regularnych odstępach czasu, należy użyć klasy Timer z pakietu Swing.

Użycie klasy Timer z pakietu Swing

Klasa Timer z pakietu Swing (javax.swing.Timer) używana jest głównie do implementacji graficznego interfejsu użytkownika (nie mylić z java.util.Timer – klasą Timer do zastosowań ogólnych). Można tej klasy używać w dwojaki sposób:

1. wykonując jakieś zadanie jeden raz, z zadanym opóźnieniem (jak w implementacji menadżera tzw. tooltip)
2. wykonując zadanie wielokrotnie w zadanych odstępach czasu (jak w implementacjach animacji, wyświetlaniu paska postępu, itp.)

W ogólności, wykorzystanie klasy javax.swing.Timer wiąże się z możliwością generacji zdarzeń i ich obsługi. To właśnie instancje klasy Timer generuje zdarzenia, które obsługiwane są przez dostarczone do ich konstruktorów instancje słuchaczy.

Klasa Timer powinna być stosowana do zadań krótkich (tj. realizowanych w krótkim czasie). Dla zadań długich (tj. o dłuższym czasie wykonania) zalecane jest stosowanie klasy SwingWorker.

Schamat aplikacji korzystającej z klasy Timer może być następujący

1. Inicjacja timera z parametrem określającym słuchacza zdarzeń.

```
public final static int ONE_SECOND = 1000;
...
timer = new Timer(ONE_SECOND, new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        //...Wykonaj zadanie...
    }
});
```

2. Uruchomienie timera

```
timer.start();
```

3. Zatrzymanie timera:

```
if (/* warunek zakończenia zadania spełniony */) {
    ...
    timer.stop();
    ...
}
```

Interfejs klasy Timer

Poniżej przedstawione zostaną najczęściej wykorzystywane konstruktory i metody klasy javax.swing.Timer. Metody te można podzielić na dwie kategorie:

- Tworzenie i inicjalizacja timera
- Uruchamianie i zatrzymywanie timera

Tworzenie i inicjalizacja timera	
Metoda lub konstruktor	**Opis**
Timer(int, ActionListener)	Tworzy timer. Argument typu int określa liczbę mililisekund przerwy pomiędzy generowanymi zdarzeniami (*action events*). Do zmiany tego okresu użyj metody setDelay. Argument ActionListener jest referencją do słuchacza zdarzeń, który zostanie zarejestrowany z timerem. Słuchaczy zdarzeń rejestruje się również za pomocą addActionListener, usuwa zaś metodą removeActionlistener.
void setDelay(int) int getDelay()	Ustawia lub odczytuje liczbę milisekund przerwy pomiędzy generowanymi zdarzeniami.
void setInitialDelay(int) int getInitialDelay()	Ustawia lub odczytuje liczbę milisekund okresu oczekiwania przed odpaleniem pierwszego zdarzenia. Domyślnie jest to liczba równa liczbie milisekund przerwy pomiędzy zdarzeniami.
void setRepeats(boolean) boolean isRepeats()	Ustawia lub odczytuje wartość logiczną, informującą, czy timer odpalać ma zdarzenia wielokrotnie. Domyślnie jest to wartość true. Aby timer odpalił zdarzenie tylko jeden raz, należy wywołać setRepeats(false)
void setCoalesce(boolean) boolean isCoalesce()	Ustawia albo odczytuje wartość logiczną, czy timer zunifikuje wielokrotne zdarzenia w toku w jedno zdarzenie, czy też nie. Domyślnie wartość ta jest true.
Uruchamianie i zatrzymywanie timera	
Metoda	**Opis**
void start() void restart()	Włącza timer. Dodatkowo restart anuluje wszystkie zdarzenia w toku.
void stop()	Wyłącza timer.
boolean isRunning()	Sprawdza, czy timer pracuje

Przykład animacji:

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

/*
 * Szablon dla aplikacji z animacją
 */
public class AnimatorApplicationTimer extends JFrame implements ActionListener {
   int frameNumber = -1;
   Timer timer;
   boolean frozen = false;
   JLabel label;

   AnimatorApplicationTimer(int fps, String windowTitle) {
      super(windowTitle);
      int delay = (fps > 0) ? (1000 / fps) : 100;
```

```
    // Utwórz timer, który odpalał będzie zdarzenia w określonych odstępach czasu (delay)
    // których obsługą zajmie się zadeklarowany słuchacz (this)
    timer = new Timer(delay, this);
    timer.setInitialDelay(0);
    timer.setCoalesce(true);

    addWindowListener(new WindowAdapter() {
        public void windowIconified(WindowEvent e) {
            stopAnimation();
        }
        public void windowDeiconified(WindowEvent e) {
            startAnimation();
        }
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });

    label = new JLabel("Frame     ", JLabel.CENTER);
    label.addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            if (frozen) {
                frozen = false;
                startAnimation();
            } else {
                frozen = true;
                stopAnimation();
            }
        }
    });

    getContentPane().add(label, BorderLayout.CENTER);
}

// Poniższą metodę można wywołać z dowolnego wątku
public void startAnimation() {
    if (frozen) {
        // Nic nie rób. Użytkownik zażądał, aby skończyć ze zmianami obrazka
    } else {
        // Uruchom animację!
        if (!timer.isRunning()) {
            timer.start();
        }
    }
}

// Poniższą metodę można wywołać z dowolnego wątku
public void stopAnimation() {
    // zatrzymaj wątek odpowiedzialny za animację
    if (timer.isRunning()) {
        timer.stop();
    }
}

public void actionPerformed(ActionEvent e) {
    // Zwiększ licznik i wyświetl go – to jest jedna klatka animacji
    frameNumber++;
    label.setText("Frame " + frameNumber);
}

public static void main(String args[]) {
    AnimatorApplicationTimer animator = null;
    int fps = 10;

    // Odczytaj szybkość animacji (klatki na sekundę) z linii wywołania
    if (args.length > 0) {
        try {
```

```
            fps = Integer.parseInt(args[0]);
        } catch (Exception e) {}
    }
    animator = new AnimatorApplicationTimer(fps, "Animator with Timer");
    animator.pack();
    animator.setVisible(true);

    // uruchamiamy animację w wątku aplikacji (bo startAnimation można wywołać
    // w dowolnym wątku
    animator.startAnimation();
  }
}
```

Użycie klasy SwingWorker

Klasa SwingWorker została utworzona, aby czasochłonne operacje można było wykonywać w tle, nie blokując interfejsu użytkownika. Implementację klasy SwingWorker zawiera plik SwingWorker.java (tzn. klasa ta nie należy do dytrybucji pakietu Swing). Aby skorzystać z klasy SwingWorker należy zadeklarować klasę dziedziczącą z niej. Tak zdefiniowana klasa potomna musi implementować metodę construct(). W metodzie tej umieszcza się kod czasochłonnych operacji.

Kiedy tworzony jest obiekt klasy dziedziczącej ze SwingWorker, klasa SwingWorker tworzy wątek, jednak go nie startuje (dotyczy to SwingWorker 3 – trzeciej edycja klasy). Aby go wystartować, należy uruchomić metodę start utworzonego obiektu. Długa operacja zostanie wykonana przez tak utworzony wątek.

Przykład użycia SwingWorker, w którym przesunięto czasochłonne operacje do wątku działającego w tle, dzięki czemu interfejs użytkownika pozostał aktywny.

```
//OLD CODE:
public void actionPerformed(ActionEvent e) {
    ...
   //   kod, wykonanie którego zajmuje
  //   dużo czasu
    ...
}
```

```
//BETTER CODE:
public void actionPerformed(ActionEvent e) {
    ...
    final SwingWorker worker = new SwingWorker() {
        public Object construct() {
    //   kod, wykonanie którego zajmuje dużo czasu
            return someValue;
        }
    };
    worker.start();  // wymagane dla SwingWorker 3
    ...
}
```

Wartością zwracaną przez construct() może być dowolny obiekt. Do odzyskania wartości można również zaimplementować metodę get – jednak takie postępowanie może prowadzić do blokad. W razie konieczności można też przerwać wątek (co zakończy get).
Jeśli wymagane jest, aby po zakończeniu długiej operacji uaktualniony został graficzny interfejs użytkownika, można albo:

- użyć get (co jest niebezpieczne ze względu na możliwość blokad), albo
- przysłonić metodę finished w zaimplementowanej podklasie SwingWorker. Metoda finished wywoływana jest, gdy skończy swoje działanie metoda construct. Ponieważ finished wykonywana jest w *event-dispatching thread*, można bezpiecznie modyfikować w niej komponenty Swing (jednak nie należy umieszczać w tej metodzie operacji, które są zbyt długie).

Przykład pochodzący z IconDemoApplet.java

```
public void actionPerformed(ActionEvent e) {
    ...
    if (icon == null) {     //haven't viewed this photo before
        loadImage(imagedir + pic.filename, current);
    } else {
```

```
        updatePhotograph(current, pic);
    }
}
...
// Wczytanie obrazka w osobnym wątku

private void loadImage(final String imagePath, final int index) {
    final SwingWorker worker = new SwingWorker() {
        ImageIcon icon = null;

        public Object construct() {
            icon = new ImageIcon(getURL(imagePath));
            return icon; // wartość zwracana nie jest wykorzystywana
        }

        // Poniższa funkcja wywołana zostanie w event-dispatching thread
        public void finished() {
            Photo pic = (Photo)pictures.elementAt(index);
            pic.setIcon(icon);
            if (index == current)
                updatePhotograph(index, pic);
        }
    };

    worker.start();
}
```

SwingWorker.java

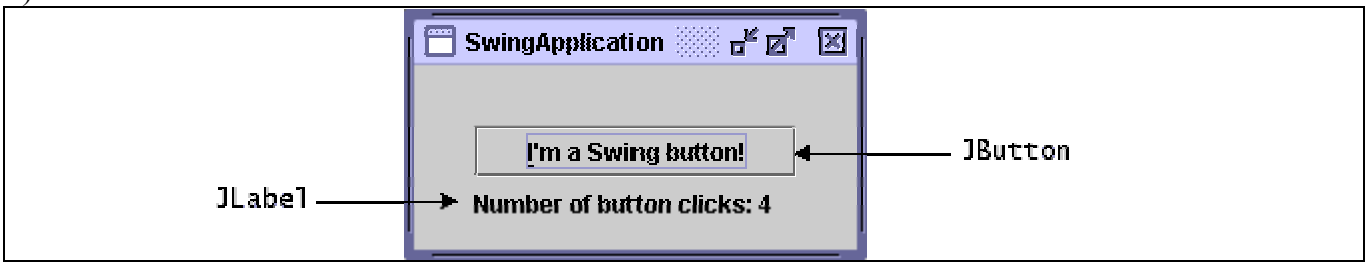
```
import javax.swing.SwingUtilities;

/**
 * This is the 3rd version of SwingWorker (also known as
 * SwingWorker 3), an abstract class that you subclass to
 * perform GUI-related work in a dedicated thread.  For
 * instructions on and examples of using this class, see:
 *
 * http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html
 *
 * Note that the API changed slightly in the 3rd version:
 * You must now invoke start() on the SwingWorker after
 * creating it.
 */
public abstract class SwingWorker {
    private Object value;  // see getValue(), setValue()

    /**
     * Class to maintain reference to current worker thread
     * under separate synchronization control.
     */
    private static class ThreadVar {
        private Thread thread;
        ThreadVar(Thread t) { thread = t; }
        synchronized Thread get() { return thread; }
        synchronized void clear() { thread = null; }
    }

    private ThreadVar threadVar;

    /**
     * Get the value produced by the worker thread, or null if it
     * hasn't been constructed yet.
     */
    protected synchronized Object getValue() {
        return value;
    }

    /**
```

```
     * Set the value produced by worker thread
     */
    private synchronized void setValue(Object x) {
        value = x;
    }

    /**
     * Compute the value to be returned by the <code>get</code> method.
     */
    public abstract Object construct();

    /**
     * Called on the event dispatching thread (not on the worker thread)
     * after the <code>construct</code> method has returned.
     */
    public void finished() {
    }

    /**
     * A new method that interrupts the worker thread.  Call this method
     * to force the worker to stop what it's doing.
     */
    public void interrupt() {
        Thread t = threadVar.get();
        if (t != null) {
            t.interrupt();
        }
        threadVar.clear();
    }

    /**
     * Return the value created by the <code>construct</code> method.
     * Returns null if either the constructing thread or the current
     * thread was interrupted before a value was produced.
     *
     * @return the value created by the <code>construct</code> method
     */
    public Object get() {
        while (true) {
            Thread t = threadVar.get();
            if (t == null) {
                return getValue();
            }
            try {
                t.join();
            }
            catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // propagate
                return null;
            }
        }
    }


    /**
     * Start a thread that will call the <code>construct</code> method
     * and then exit.
     */
    public SwingWorker() {
        final Runnable doFinished = new Runnable() {
           public void run() { finished(); }
        };

        Runnable doConstruct = new Runnable() {
            public void run() {
                try {
                    setValue(construct());
```

```
            }
            finally {
                threadVar.clear();
            }

            SwingUtilities.invokeLater(doFinished);
        }
    };

    Thread t = new Thread(doConstruct);
    threadVar = new ThreadVar(t);
  }

  /**
   * Start the worker thread.
   */
  public void start() {
    Thread t = threadVar.get();
    if (t != null) {
        t.start();
    }
  }
}
```

Drugi program korzystający z klas pakietu Swing

Działanie przykładu: za każdym razem, kiedy kliknięty zostanie przycisk (JButton), zmienia się etykieta (JLabel).

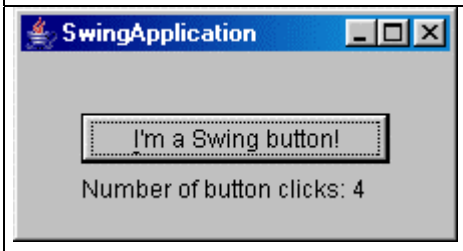
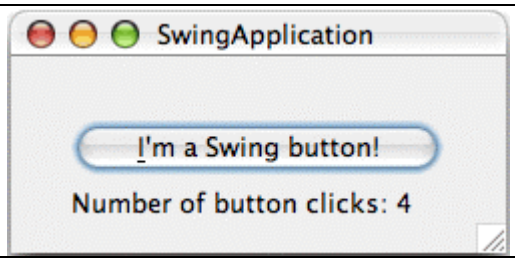



```
/*
 * SwingApplication.java is a 1.4 example that requires
 * no other files.
 */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SwingApplication implements ActionListener {
    private static String labelPrefix = "Number of button clicks: ";
    private int numClicks = 0;
    final JLabel label = new JLabel(labelPrefix + "0    ");

    //Specify the look and feel to use.  Valid values:
    //null (use the default), "Metal", "System", "Motif", "GTK+"
    final static String LOOKANDFEEL = null;

    public Component createComponents() {
        JButton button = new JButton("I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(this);
        label.setLabelFor(button);

        /*
         * An easy way to put space between a top-level container
         * and its contents is to put the contents in a JPanel
         * that has an "empty" border.
         */
```

```
    JPanel pane = new JPanel(new GridLayout(0, 1));
    pane.add(button);
    pane.add(label);
    pane.setBorder(BorderFactory.createEmptyBorder(
                                30, //top
                                30, //left
                                10, //bottom
                                30) //right
                                );

    return pane;
}

public void actionPerformed(ActionEvent e) {
    numClicks++;
    label.setText(labelPrefix + numClicks);
}

private static void initLookAndFeel() {
    String lookAndFeel = null;

    if (LOOKANDFEEL != null) {
        if (LOOKANDFEEL.equals("Metal")) {
            lookAndFeel = UIManager.getCrossPlatformLookAndFeelClassName();
        } else if (LOOKANDFEEL.equals("System")) {
            lookAndFeel = UIManager.getSystemLookAndFeelClassName();
        } else if (LOOKANDFEEL.equals("Motif")) {
            lookAndFeel = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
        } else if (LOOKANDFEEL.equals("GTK+")) { //new in 1.4.2
            lookAndFeel = "com.sun.java.swing.plaf.gtk.GTKLookAndFeel";
        } else {
            System.err.println("Unexpected value of LOOKANDFEEL specified: "
                               + LOOKANDFEEL);
            lookAndFeel = UIManager.getCrossPlatformLookAndFeelClassName();
        }

        try {
            UIManager.setLookAndFeel(lookAndFeel);
        } catch (ClassNotFoundException e) {
            System.err.println("Couldn't find class for specified look and feel:"
                               + lookAndFeel);
            System.err.println("Did you include the L&F library in the class path?");
            System.err.println("Using the default look and feel.");
        } catch (UnsupportedLookAndFeelException e) {
            System.err.println("Can't use the specified look and feel ("
                               + lookAndFeel
                               + ") on this platform.");
            System.err.println("Using the default look and feel.");
        } catch (Exception e) {
            System.err.println("Couldn't get specified look and feel ("
                               + lookAndFeel
                               + "), for some reason.");
            System.err.println("Using the default look and feel.");
            e.printStackTrace();
        }
    }
}

/**
 * Create the GUI and show it.  For thread safety,
 * this method should be invoked from the
 * event-dispatching thread.
 */
private static void createAndShowGUI() {
    //Set the look and feel.
    initLookAndFeel();
```

```
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);

        //Create and set up the window.
        JFrame frame = new JFrame("SwingApplication");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        SwingApplication app = new SwingApplication();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);

        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Wygląd:

Java	GTK+
	
Windows	Mac OS
	

Ustawienie wyglądu programowo
Ustawienie wyglądu powinno odbywać się w pierwszych linijkach kodu programu. Do ustawienia wyglądu służy metoda UIManager.setLookAndFeel. Argumentem tej metody jest w pełni kwalifikowana nazwa klasy będącej pochodną klasy LookAndFeel. Klasa odpowiedzialna za wygląd musi istnieć (tzn. powinna być widoczna w ścieżkach Javy). W przypadku jej braku ustawiony zostanie wygląd domyślny.

Do automatycznego wygenerowania nazwy klasy odpowiedzialnej za wygląd używa się metod klasy UIManager:

- getCrossPlatformLookAndFeelClassName() - dostarcza klasę implementującą standardowy wygląd Javy.
- getSystemLookAndFeelClassName() – dostarcza klasę implementującą wygląd natywny dla systemu, w którym uruchamiana jest aplikacja. Dla systemu Microsoft Windows będzie to

Windows look and feel, dla systemu Mac OS, będzie to *Mac OS look and feel*, dla platform Unix (Solaris, Linux) będzie to *CDE/Motif look and feel*.

Przykłady:
Aby osiągnąć wygląd zgodny z *GTK+ look and feel* należy wywołać:

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.gtk.GTKLookAndFeel");
```

przy czym można podać określony temat wyglądu za pomocą pliku zasobów lub parametru gtkthemefile w linii komend, np.:

```
java -Dswing.gtkthemefile=customTheme/gtkrc Application
```

Aby otrzymać typowy wygląd Javy, który jest domyślny, należy wywołać:

```
public static void main(String[] args) {
   try {
      UIManager.setLookAndFeel(
         UIManager.getCrossPlatformLookAndFeelClassName());
   } catch (Exception e) { }

   new SwingApplication(); //Create and show the GUI.
}
```

"javax.swing.plaf.metal.MetalLookAndFeel" – Java Metal
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel" – Windows (Windows XP) look and feel (na platformie Windows (XP))
"com.sun.java.swing.plaf.motif.MotifLookAndFeel" – CDE/Motif look and feel (na dowolnej platformie)

Ustawienie wyglądu za pomocą parametrów w linii wywołania
Korzysta się tutaj z flagi –D oraz własności swing.defaultlaf. Na przykład:

```
java -Dswing.defaultlaf=com.sun.java.swing.plaf.gtk.GTKLookAndFeel MyApp

java -Dswing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel MyA
```

Ustawienie wyglądu poprzez swing.properties
Można ustawić wygląd aplikacji przez modyfikację pliku swing.properties, gdzie ustawia się własność swing.defaultlaf. Plik ten zlokalizowany jest w katalogu lib dystrybucji Javy i zawiera:

```
# Swing properties
swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Kolejność wyboru wyglądu przez UI Manager

1. Jeśli program ustawił wygląd przed stworzeniem komponentów, wtedy UI manager próbuje utworzyć instancję odpowiedniej do wyglądu klasy. Komponenty tworzone następnie będą miały zadeklarowany wygląd.
2. Jeśli program nie zdołał utworzyć zadanego wyglądu, UI manager używa wyglądu wyspecyfikowanego we własności swing.defaultlaf. W przypadku wystąpienia swing.defaultlaf w pliku swing.properties oraz w linii wywołania, definicja z linii wywołania brana jest jako ważniejsza.
3. Jeśli w powyższych krokach nie udało się ustawić zadanego wyglądu, program użyje domyślny (Javy) wygląd.

Zmiana wyglądu po uruchomieniu

Zmiana wyglądu komponentów utworzonych już przez program możliwa jest przez wykonanie metody updateComponentTreeUI należącej do SwingUtilities dla kontenera górnego poziomu. Ponieważ ze zmianą wyglądu wiązać się może zmiana rozmiaru komponentów, należałoby te nowe rozmiary uwzględnić na ekranie:

```
UIManager.setLookAndFeel(lnfName);
SwingUtilities.updateComponentTreeUI(frame);
frame.pack();
```

Przyciski i etykiety

Inicjalizacja klawisza, na którym wypisane jest tekst "I'm a Swing button!", z którym związany jest mnemonik 'i' (który może posłużyć użytkownikowi do symulowania kliknięcia na tym przycisku), i dla którego rejestrowany jest słuchacz zdarzeń.

```
JButton button = new JButton("I'm a Swing button!");
button.setMnemonic('i');
button.addActionListener(...create an action listener...);
```

Jeden z przycisków umieszczanych na kontenerach poziomu najwyższego może być przyciskiem domyślnym. Taki przycisk zazwyczaj jest podświetlony i zadziała po naciśnięciu klawisza Enter (pod warunkiem, że fokus klawiatury jest ustawiony na kontener). Aby ustawić przycisk jako przycisk domyślny, korzysta się z metody setDefaultButton panelu root należącego do kontenera poziomu najwyższego:

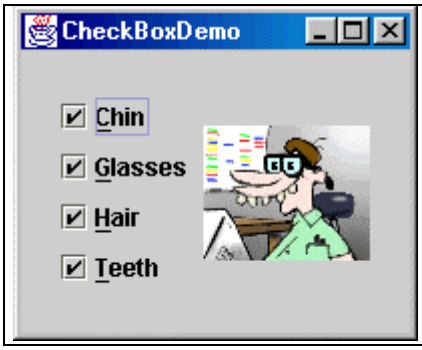
```
 // np. w konstruktorze klasy dziedziczącej po JDialog:
getRootPane().setDefaultButton(setButton);
```

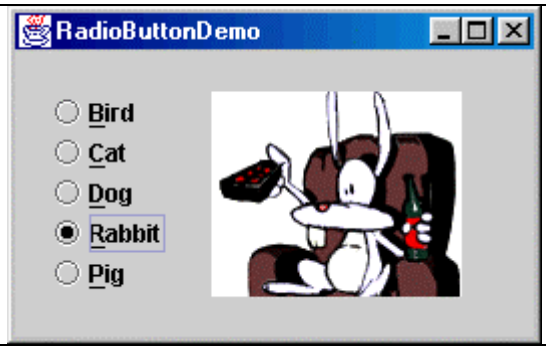
Inicjalizacja etykiety i manipulacja nią:

```
private static String labelPrefix = "Number of button clicks: ";
private int numClicks = 0;
...
// w części inicjalizującej GUI:
final JLabel label = new JLabel(labelPrefix + "0    ");
...
// assistive technologies – aby zaznaczyć, że etykieta związana jest z klawiszem
label.setLabelFor(button);


...
// w obsłudze zdarzenia kliknięcia na przycisku:
label.setText(labelPrefix + numClicks);
```

Część z komponentów Swing umieszczona jest w tym samym drzewie dziedziczenia i, w związku z tym, ma podobne cechy. Takimi klasami są JButton, JMenuItem, JToggleButton (z pochodnymi klasami JCheckBox, JRadioButton). Wszystkie one dziedziczą z klasy AbstractButton.

Klasy JCheckBox służą do zaznaczania dowolnej kombinacji wśród możliwych opcji wyboru. JRadioButton pozwalają na selekcję tylko jednej wartości (jeśli powiązane są ButtonGroup).






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

/* CheckBoxDemo.java – używa obrazków z katalogu images/geek/geek
*/

public class CheckBoxDemo extends JPanel {
   JCheckBox chinButton;
   JCheckBox glassesButton;
   JCheckBox hairButton;
   JCheckBox teethButton;

   /*
    * Cztery opcje dają możliwość wyboru 16 różnych ich kombinacji
    * Kombinacjom odpowiadają obrazki, przechowywane w osobnych
    * plikach o nazwach zgodnych ze schematem "geek-XXXX.gif"
    * gdzie XXXX jest reprezentacją znakową jednej z 16 możliwości.
    * np.  ----  (nic nie wybrano),   cg--  (wybrano dwie pierwsze opcje)
    *       -ght  (wybrano trzy ostatnie opcje),  cght  (wybrano wszystkie opcje)
    *
    */

   StringBuffer choices;
   JLabel pictureLabel;

   public CheckBoxDemo() {

      // Utworzenie check boxes
      chinButton = new JCheckBox("Chin");
      chinButton.setMnemonic(KeyEvent.VK_C);
      chinButton.setSelected(true);

      glassesButton = new JCheckBox("Glasses");
      glassesButton.setMnemonic(KeyEvent.VK_G);
      glassesButton.setSelected(true);

      hairButton = new JCheckBox("Hair");
      hairButton.setMnemonic(KeyEvent.VK_H);
      hairButton.setSelected(true);

      teethButton = new JCheckBox("Teeth");
      teethButton.setMnemonic(KeyEvent.VK_T);
      teethButton.setSelected(true);

      // Rejestracja słuchacza
      CheckBoxListener myListener = new CheckBoxListener();
      chinButton.addItemListener(myListener);
      glassesButton.addItemListener(myListener);
      hairButton.addItemListener(myListener);
      teethButton.addItemListener(myListener);

      // początkowe XXXX.
      choices = new StringBuffer("cght");
```

```
        // Ustawienie etykiety rysunku
        pictureLabel = new JLabel(new ImageIcon("images/geek/geek-"  + choices.toString() + ".gif"));
        pictureLabel.setToolTipText(choices.toString());

        // Wstawienie check boxes do kolumny na panelu
        JPanel checkPanel = new JPanel();
        checkPanel.setLayout(new GridLayout(0, 1));
        checkPanel.add(chinButton);
        checkPanel.add(glassesButton);
        checkPanel.add(hairButton);
        checkPanel.add(teethButton);

        setLayout(new BorderLayout());
        add(checkPanel, BorderLayout.WEST);
        add(pictureLabel, BorderLayout.CENTER);
        setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
    }

    /** Implementacja słuchacza. */
    class CheckBoxListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            int index = 0;  char c = '-';
            Object source = e.getItemSelectable();

            if (source == chinButton) {
                index = 0;  c = 'c';
            } else if (source == glassesButton) {
                index = 1;  c = 'g';
            } else if (source == hairButton) {
                index = 2;  c = 'h';
            } else if (source == teethButton) {
                index = 3;  c = 't';
            }

            if (e.getStateChange() == ItemEvent.DESELECTED)   c = '-';
            choices.setCharAt(index, c);
            pictureLabel.setIcon(new ImageIcon("images/geek/geek-" + choices.toString() + ".gif"));
            pictureLabel.setToolTipText(choices.toString());
        }
    }

    public static void main(String s[]) {
        JFrame frame = new JFrame("CheckBoxDemo");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        frame.setContentPane(new CheckBoxDemo());
        frame.pack();
        frame.setVisible(true);
    }
}
```

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

/*
 * RadioButtonDemo.java korzysta z obrazków z katalogu images
 */
public class RadioButtonDemo extends JPanel implements ActionListener {
    static String birdString = "Bird";
```

```
  static String catString = "Cat";
  static String dogString = "Dog";
  static String rabbitString = "Rabbit";
  static String pigString = "Pig";

  JLabel picture;

  public RadioButtonDemo() {
     super(new BorderLayout());

     // utworzenie radio buttons
     JRadioButton birdButton = new JRadioButton(birdString);
     birdButton.setMnemonic(KeyEvent.VK_B);
     birdButton.setActionCommand(birdString);
     birdButton.setSelected(true);

     JRadioButton catButton = new JRadioButton(catString);
     catButton.setMnemonic(KeyEvent.VK_C);
     catButton.setActionCommand(catString);

     JRadioButton dogButton = new JRadioButton(dogString);
     dogButton.setMnemonic(KeyEvent.VK_D);
     dogButton.setActionCommand(dogString);

     JRadioButton rabbitButton = new JRadioButton(rabbitString);
     rabbitButton.setMnemonic(KeyEvent.VK_R);
     rabbitButton.setActionCommand(rabbitString);

     JRadioButton pigButton = new JRadioButton(pigString);
     pigButton.setMnemonic(KeyEvent.VK_P);
     pigButton.setActionCommand(pigString);

     // zgrupowanie radio buttons
     ButtonGroup group = new ButtonGroup();
     group.add(birdButton);
     group.add(catButton);
     group.add(dogButton);
     group.add(rabbitButton);
     group.add(pigButton);

     // rejestracja słuchacza
     birdButton.addActionListener(this);
     catButton.addActionListener(this);
     dogButton.addActionListener(this);
     rabbitButton.addActionListener(this);
     pigButton.addActionListener(this);

     // Ustawienie etykiety obrazka
     picture = new JLabel(createImageIcon("images/" + birdString+ ".gif"));

     // preferowany rozmiar (wpisany na twardo)
     picture.setPreferredSize(new Dimension(177, 122));


     // ustawienie radio buttons w kolumnie na panelu
     JPanel radioPanel = new JPanel(new GridLayout(0, 1));
     radioPanel.add(birdButton);
     radioPanel.add(catButton);
     radioPanel.add(dogButton);
     radioPanel.add(rabbitButton);
     radioPanel.add(pigButton);

     add(radioPanel, BorderLayout.LINE_START);
     add(picture, BorderLayout.CENTER);
     setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
  }
```

```
    /** Implementacja słuchacza */
    public void actionPerformed(ActionEvent e) {
        picture.setIcon(createImageIcon("images/" + e.getActionCommand() + ".gif"));
    }

    /** Zwraca ImageIcon, lub null, gdy ścieżka jest niepoprawna */
    protected static ImageIcon createImageIcon(String path) {
        java.net.URL imgURL = RadioButtonDemo.class.getResource(path);
        if (imgURL != null) {
            return new ImageIcon(imgURL);
        } else {
            System.err.println("Couldn't find file: " + path); return null;
        }
    }

    /**
     * Utworzenie graficznego interfejsu użytkownika
     */
    private static void createAndShowGUI() {
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new JFrame("RadioButtonDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JComponent newContentPane = new RadioButtonDemo();
        newContentPane.setOpaque(true); //content panes musi być nieprzeźroczysty
        frame.setContentPane(newContentPane);

        frame.pack();
        frame.setVisible(true);
    }

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

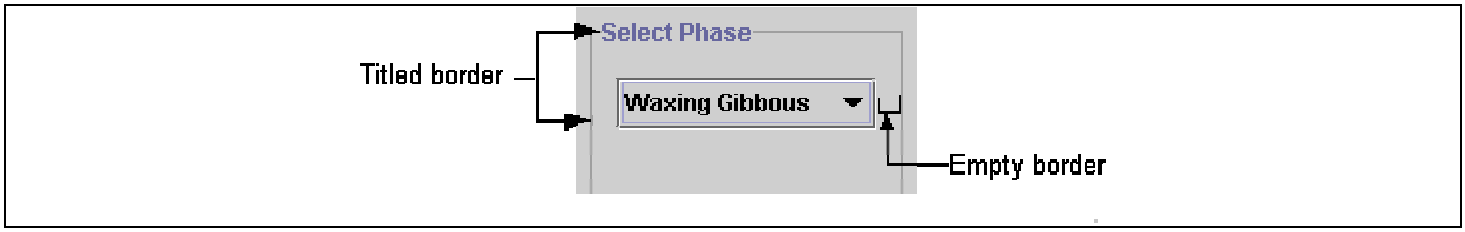
Dodawanie obwiedni wokół komponentów

Komponenty dziedziczące z klasy JComponent mają obwiednie (tj. dodatkowe pole wokół komponentu, *border*), której wielkość można zmieniać. Aby ustawić obwiednię, można użyć się metody setBorder klasy JComponent oraz metodę createEmptyBorder klasy BorderFactory:

```
pane.setBorder(BorderFactory.createEmptyBorder(
                        30, // dodatkowe 30 punktów od góry
                        30, // dodatkowe 30 punktów z lewej strony
                        10, // dodatkowe 10 punktów od dołu
                        30) // dodatkowe 30 punktów z prawej strony
                        );
```

Oprócz zwykłych obwiedni do komponentu można dodać obwiednię złożoną (*compound border*). Obwiednia taka składa się z obwiedni tytułowej (*titled border*) oraz obwiedni wewnętrznej (*empty border*). Obwiednię złożoną tworzy się metodą createCompoundBorder klasy BorderFactory:

```
panel.setBorder(BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder("Select Phase"),
BorderFactory.createEmptyBorder(5,5,5,5)));
```



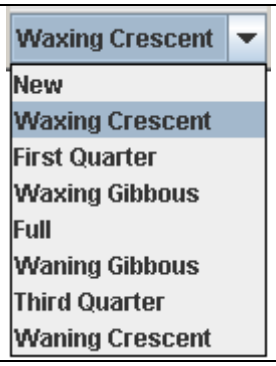
HTML w komponentach
Na niektórych komponentach Swing (przyciskach, etykietach) można umieszczać tekst w formacie HTML. Pozwala to na wyświetlanie specjalnych znaków, zmianę linii, użycie różnych fontów i kolorów. Jeśli cały wyprowadzany tekst jest tego samego typu, to można użyć zamiast kodów HTML metody setFont() komponentu.

```
JLabel label = new JLabel();
label.setText("<html><font color=blue>" + tempFahr   + "&#176  Fahrenheit </font></html>");
```

Ikony na przyciskach
Przyciski mogą zawierać ikony. W klasach Swing ikona jest obiektem, który dysponuje interfejsem Icon. W klasach Swing dostarczono implementację tego interfejsu: ImageIcon. Klasa ta pozwala na wyświetlanie ikon utworzonych na bazie obrazków GIF i JPEG. Pierwszy parametr konstruktora klasy to nazwa pliku obrazka, drugi to tekst, który wykorzystany być może przez *assistive technologies*.

```
ImageIcon icon = new ImageIcon("images/convert.gif", "Convert temperature");
JButton convertTemp = new JButton(icon);
```

Listy wyboru combo
ComboBox umożliwia dokonywanie wyboru jednego elementu z listy. ComboBox może być zarówno edytowalne, jak i nieedytowalne (domyślnie jest nieedytowalne).
ComboBox wygląda jak przycisk, dopóki użytkownik nie zacznie operacji na nim. Jeśli użytkownik zacznie operację, np. klikając na nim, to ComboBox rozwija listę wszystkich elementów z nim związanych.




```
JComboBox phaseChoices = null;

String[] phases = { "New", "Waxing Crescent", "First Quarter",
                    "Waxing Gibbous", "Full", "Waning Gibbous",
                    "Third Quarter", "Waning Crescent"};
phaseChoices = new JComboBox(phases);
phaseChoices.setSelectedIndex(1);
```

Kiedy użytkownik wybierze element z listy, ComboBox generuje zdarzenie, które obsłużyć można w słuchaczu ActionListener.

```
phaseChoices.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent event) {
        if ("comboBoxChanged".equals(event.getActionCommand())) {
            phaseIconLabel.setIcon(images[phaseChoices.getSelectedIndex()]);
        }
  }
});
```

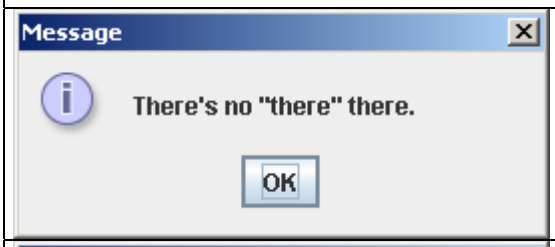
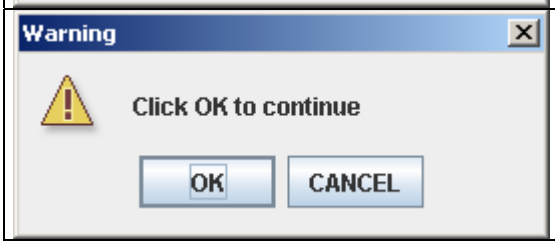
Okna dialogowe
Istnieje kilka klas dostarczających okna dialogowe – okienka, które posiadają ograniczone możliwości (w porównaniu do JFrame). Każde okno dialogowe jest zależne od głównej ramki (JFrame). Jeśli ramka główna

jest zamieniana w ikonę, wtedy dialog znika z ekranu. Jeśli ramka główna pojawia się na ekranie, pojawia się również i dialog.

Do utworzenia prostego okna dialogowego można użyć metod klasy JOptionPane.
Wygląd okienek dialogowych uzyskiwanych w ten sposób:

<table>
<tr><td rowspan="2">ikona</td><td>wiadomość</td></tr>
<tr><td>pole wprowadzania</td></tr>
<tr><td colspan="2">przyciski opcji</td></tr>
</table>

Dopóki okno dialogowe, zadeklarowane jak w poniższych przykładach, istnieć będzie na ekranie, dopóty przysłaniać będzie (blokować) inne okna tej samej aplikacji. Jest to tryb pracy *modal*.

<table>
<tr><td> 
</td><td>//default title and icon
Object[] options = {"Yes!",
"No, I'll pass",
"Well, if I must"};
int n = JOptionPane.showOptionDialog(frame,
"Duke is a cartoon mascot. \n"
+ "Do you still want to cast your vote?",
"A Follow-up Question",
JOptionPane.YES_NO_CANCEL_OPTION,
JOptionPane.QUESTION_MESSAGE,
null,
options,
options[2]);</td></tr>
<tr><td> 
</td><td>JOptionPane.showMessageDialog(frame, "There's no \"there\" there.");</td></tr>
<tr><td> 
</td><td>JOptionPane.showMessageDialog(null, "alert", "alert", JOptionPane.ERROR_MESSAGE);</td></tr>
<tr><td> 
</td><td>Object[] possibleValues = { "First", "Second", "Third" };
Object selectedValue =
JOptionPane.showInputDialog(null, "Choose one", "Input", JOptionPane.INFORMATION_MESSAGE, null, possibleValues, possibleValues[0]);</td></tr>
<tr><td> 
</td><td>Object[] options = { "OK", "CANCEL" };
JOptionPane.showOptionDialog(null, "Click OK to continue", "Warning",
JOptionPane.DEFAULT_OPTION,
JOptionPane.WARNING_MESSAGE, null, options, options[0]);</td></tr>
</table>

JOptionPane pozwala definiować komponenty na nim wyświetlane oraz pozwala określać sposób ich ułożenia. Dostarcza ponadto ikon, pozwala na określenie tytułu okienka i tekstu na nim, pozwala definiować tekst przycisku, pozwala na określenie miejsca na ekranie, w którym dialog ma się pojawiać. Służą do tego metody:

showConfirmDialog	Dialog będzie okienkiem zatwierdzania z klawiszami YES/NO/CANCEL
showInputDialog	Dialog będzie okienkiem wprowadzania danych
showMessageDialog	Dialog służył będzie wyświetlaniu aktualnych informacji
showOptionDialog	Dialog jest uogólnieniem wszystkich trzech powyższych dialogów

Każda z tych metod ma odpowiednik postaci showInternalXXX, który jest używany do wyświetlania dialogów w wewnętrzych ramkach (obiektach typu JInternalFrame).

Parametry metod showXXXDialog

Component parentComponent	parentComponent musi być ramką, komponentem w ramce lub wartością null. Jeśli jesto to: ramka - dialog wyświetlony zostanie nad centrum ramki i będzie zależeć od ramki. komponent wewnątrz ramki – dialog wyświetlony zostanie nad centrum komponentu null – dialog nie będzie zależał od ramki, pojawi się w centrum ekranu Konstruktory JOptionPane nie posiadają takiego parametru. Dlatego ramkę ojcowską definiuje się, kiedy tworzony jest JDialog zawierający JOptionPane - wykorzystuje się wtedy setLocationRelativeTo (metodę JDialog) do zdefiniowania położenia okna dialogowego.
Object message	argument ten określa, co dialog powinien wyświetlać w swoim głównym polu. Sposób wyświetlania zależy od typu argumentu: String – wyświetlana jest etykieta z podanym tekstem Object[] – wyświetlana jest seria obiektów uporządkowana w pionowy stos. Postać obiektów jest wynikiem interpretacji typu danego obiektu (mamy tu do czynienia z rekurencyjną interpretacją) Component – wyświetlany jest komponent Icon – wyświetlana jest ikona upakowana w obiekt typu JLabel inne – wyświetlany jest upakowany w obiekt JLabel rezultat wywołania metody toString danego obiektu.
String title	Tytuł okna dialogowego
int optionType	pozwala określić, które z przycisków powinny pojawić się w dolnej części okienka dialogowego. Można wybierać pomiędzy: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION
int messageType	jest to argument określający ikonę, która będzie wyświetlana na okienku dialogowym. Można wybrać jedną z opcji: PLAIN_MESSAGE (brak ikony), ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE
Icon icon	Definiuje ikonę, która ma być wyświetlana na okienku dialogu (domyślna wartość zdefiniowana przez messageType)
Object[] options	Specyfikacja obiektów, które mogą pojawić się na oknie dialogowym poza zadeklarowanymi przez optionType przyciskami opcji. Faktyczny to, co się pojawi, zależy od typu obiektów w tablicy oraz typu okna dialogowego. W ogólności: String – wstawia przycisk JButton z tekstową etykietą w obszar przycisków lub listę wyboru ComboBox Component – wstawia komponent Icon – wstawia przycisk JButton z ikoną inne – wstawia przycisk JButton z tekstową etykietą, dla której wartość uzyskana jest w metody toString obiektu
Object initialValue	Parametr ten pozwala zdefiniować opcję (wartość) domyślną.

Rozpoznawanie akcji użytkownika w okienku dialogu

Metody showMessageDialog oraz showOptionDialog zwracają liczbę całkowitą, odpowiadającą wyborowi dokonanego przez użytkownika. Wartościami tymi są:

YES_OPTION,
NO_OPTION,
CANCEL_OPTION,
OK_OPTION,
CLOSED_OPTION.

Każda z opcji, za wyjątkiem CLOSED_OPTION, odpowiada przyciskowi, jaki został wybrany przez użytkownika. Otrzymanie wartości CLOSED_OPTION oznacza, że użytkownik zamknął raczej okienko dialogu niż wybrał jakiś przycisk.

Bezpośrednie użycie JOptionPane

```
JOptionPane pane = new JOptionPane(arguments);
 pane.set.Xxxx(...); // Configure
 JDialog dialog = pane.createDialog(parentComponent, title);
 dialog.show();
 Object selectedValue = pane.getValue();
 if(selectedValue == null)
   return CLOSED_OPTION;
 //If there is not an array of option buttons:
 if(options == null) {
   if(selectedValue instanceof Integer)
      return ((Integer)selectedValue).intValue();
   return CLOSED_OPTION;
 }
 //If there is an array of option buttons:
 for(int counter = 0, maxCounter = options.length;
    counter < maxCounter; counter++) {
    if(options[counter].equals(selectedValue))
    return counter;
 }
 return CLOSED_OPTION;
```