Metody zarządzania kolekcjami:

activeCount - liczba aktywnych wątków w grupie. Często używana w połączeniu z metodą enumerate, aby otrzymać tablicę referencji do wszystkich aktywnych wątków w grupie.

Przykład:
(tworzenie tablicy aktywnych wątków i wydrukowywanie ich nazw)

```
public class EnumerateTest {

   public void listCurrentThreads() {

       ThreadGroup currentGroup =Thread.currentThread().getThreadGroup();
       int numThreads = currentGroup.activeCount();
       Thread[] listOfThreads = new Thread[numThreads];

       currentGroup.enumerate(listOfThreads);
       for (int i = 0; i < numThreads; i++)
          System.out.println("Thread #" + i + " = " +
                              listOfThreads[i].getName());

  }

}
```

Metody operujące na grupie:

Metody operujące na poziomie grupy są następujące:
getMaxPriority, setMaxPriority
getDaemon, setDaemon
getName
getParent, parentOf
toString

Przykład:
(test ustawiania priorytetów)

```
public class MaxPriorityTest {
   public static void main(String[] args) {

      ThreadGroup groupNORM = new ThreadGroup( "A group with normal priority");
      Thread priorityMAX = new Thread(groupNORM, "A thread with maximum
priority");

      // set Thread's priority to max (10)
      priorityMAX.setPriority(Thread.MAX_PRIORITY);

      // set ThreadGroup's max priority to normal (5)
      groupNORM.setMaxPriority(Thread.NORM_PRIORITY);

      System.out.println("Group's maximum priority = " + groupNORM.getMaxPriority());
      System.out.println("Thread's priority = " + priorityMAX.getPriority());
   }
}
```

Wyjściem programu jest:
Group's maximum priority = 5
Thread's priority = 10

Metody operujące na wszystkich wątkach wewnątrz grupy:
Zalicza się do nich trzy metody, mogące zmienić stan wszystkich wątków w grupie i podgrupach jednocześnie:
resume
stop
suspend

Metody ograniczania dostępu:
Klasa ThreadGroup sama w sobie

Klasy Thread oraz ThreadGroup same w sobie nie daje żadnych możliwości, aby ograniczyć jakiekolwiek prawa dostępu. Obie klasy posiadają jednak metodę checkAccess, która wywołuje metodę checkAcces bieżącego menadżera bezpieczeństwa odpowiedzialnego za przyznawanie praw dostępu.

Metody klasy ThreadGroup wywołujące checkAccess	Metody klasy Thread wywołujące checkAccess
ThreadGroup(ThreadGroup *parent*, String *name*) setDaemon(boolean *isDaemon*) setMaxPriority(int *maxPriority*) stop suspend resume destroy	konstruktory, które definiuja grupę wątku stop suspend resume setPriority(int *priority*) setName(String *name*) setDaemon(boolean *isDaemon*)

Użycie klas Timer i TimerTask

Klasy timerów wprowadzone zostały w wersji Javy 1.3. Zdefiniowano je w pakiecie java.util. Klasa Timer służy do szeregowania (sheduling) instancji klasy nazywanej TimerTask.

Przykład:
(uruchomienie zadania po 5 sekundach)

Reminder.java

```
import java.util.Timer;
import java.util.TimerTask;

public class Reminder {
    Timer timer;

    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
    }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Time's up!");
            timer.cancel(); //Terminate the timer thread
        }
    }

    public static void main(String args[]) {
        System.out.println("About to schedule task.");
        new Reminder(5);
        System.out.println("Task scheduled.");
    }
}
```

Po uruchomieniu programu wyświetlone zostaną komunikaty:

```
Task scheduled.
Five seconds later, you see this:
Time's up!
```

Zatrzymywanie Timerów:
Domyślnie program działa tak długo, jak długo działa wątek timera. Wątek timera można przerwać na trzy sposoby:

1. przez wywołanie metody cancel na timerze. Wywołanie tej funkcji może nastąpić w dowolnym miejscu programu (np. w metodzie run).
2. przez stworzenie wątku timera jako wątku-demona (deamon): jeśli wszystkie pozostawione przez program wątkami są demonami, program kończy się. Aby stworzyć demona, w konstruktorze Timera należy umieścić argument true: new Timer (true).
3. gdy wszystkie zadania przeznaczone do uruchomienia zakończyły się, usunięcie wszystkich referencji do timera (obiektu klasy Timer) zakończy jego działanie (choć może nie natychmiast).
4. przez wywołanie metody System.exit. Ten sposób stosowany jest, gdy w użyciu są klasy AWT.

Wykonywanie zadań wielokrotnie:
schedule(TimerTask *task*, long *delay*, long *period*) // mogą pojawić się opóźnienia
schedule(TimerTask *task*, Date *time*, long *period*)
scheduleAtFixedRate(TimerTask *task*, long *delay*, long *period*) // opóźnienia są redukowane
scheduleAtFixedRate(TimerTask *task*, Date *firstTime*, long *period*)

Przykład:
(użycie metody schedule z czasem uruchomienia zadania podanym bezwzględnie)

```
//Get the Date corresponding to 11:01:00 pm today.
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();

timer = new Timer();
timer.schedule(new RemindTask(), time);
```

Implementacja menadżera bezpieczeństwa

W pakiecie java.lang umieszczono abstrakcyjna klasę SecurityManager, która dostarcza programowego interfejsu oraz częściowej implementacji dla wszystkich menadżerów bezpieczeństwa Javy. Domyślnie aplikacja Javy nie posiada żadnego menadżera bezpieczeństwa. Bieżący menadżer bezpieczeństwa obowiązujący w danej aplikacji można uzyskać przez wywołanie metody getSecurityManager() z klasy System:

```
SecurityManager appsm = System.getSecurityManager();
```

getSecurityManager() zwróci wartość null jeśli z bieżącą aplikacją nie jest związany żaden menadżer bezpieczeństwa.

Napisanie własnego menadżera bezpieczeństwa ogranicza się do wykonania dwóch kroków:

- stworzenia klasy dziedziczącej po klasie SecurityManager
- przysłonięcia wybranych metod

W klasie SecurityManager istnieje szereg zdefiniowanych metod check*XXX*() służących weryfikacji praw dostępu dla różnorakich operacji. Wszystkie metody check*XXX*() klasy SecurityManager działają w ten sam sposób:

- jeśli dostęp jest dozwolony, po prostu kończą się
- jeśli dostęp jest zabroniony, zgłaszają wyjątek SecurityException.

Własne metody przysłaniające metody check*XXX*() z klasy SecurityManager powinny zachowywać się w ten sam sposób.

Przykład:
(zabezpieczenie otwarcia pliku do czytania i pisania: implementacja menadżera)

```
class PasswordSecurityManager extends SecurityManager {
private: String password;

PasswordSecurityManager(String password) {
   super();
   this.password = password;
}
private boolean accessOK() {
   int c;
   DataInputStream dis = new DataInputStream(System.in);
   String response;

   System.out.println("What's the secret password?");
   try {
      response = dis.readLine();
      if (response.equals(password))
        return true;
      else
         return false;
   } catch (IOException e) {
      return false;
   }
}
public void checkRead(FileDescriptor filedescriptor) {
   if (!accessOK())    throw new SecurityException("Not a Chance!");
}
public void checkRead(String filename) {
   if (!accessOK())   throw new SecurityException("No Way!");
}
public void checkRead(String filename, Object executionContext) {
   if (!accessOK())    throw new SecurityException("Forget It!");
}
public void checkWrite(FileDescriptor filedescriptor) {
   if (!accessOK())    throw new SecurityException("Not!");
}
public void checkWrite(String filename) {
   if (!accessOK())
      throw new SecurityException("Not Even!");
}
}
```

Obiekty, na których można przeprowadzać różnego rodzaju operacje (pierwsza kolumna) oraz zestaw metod z klasy SecurityManager odpowiedzialnych za kontrolę bezpieczeństwa wykonania tych operacji:

Operacja na	Dostępne metody
sockets	checkAccept(String *host*, int *port*) checkConnect(String *host*, int *port*) checkConnect(String *host*, int *port*, Object *executionContext*) checkListen(int *port*)
threads	checkAccess(Thread *thread*) checkAccess(ThreadGroup *threadgroup*)
class loader	checkCreateClassLoader()
file system	checkDelete(String *filename*) checkLink(String *library*) checkRead(FileDescriptor *filedescriptor*) checkRead(String *filename*) checkRead(String *filename*, Object *executionContext*) checkWrite(FileDescriptor *filedescriptor*) checkWrite(String *filename*)
system commands	checkExec(String *command*)
interpreter	checkExit(int *status*)
package	checkPackageAccess(String *packageName*) checkPackageDefinition(String *packageName*)
properties	checkPropertiesAccess() checkPropertyAccess(String *key*) checkPropertyAccess(String *key*, String *def*)
networking	checkSetFactory()
windows	checkTopLevelWindow(Object *window*)

Generalnie wywołania metod check*XXX*() zaimplementowane są w pakietach klas Javy, z którymi wiążą się zagadnienia bezpieczeństwa. Jeśli więc korzysta się z tych klas, metod check*XXX*() nie trzeba jawnie wywoływać. System sam wywoła odpowiednią metodę w odpowiednim czasie. Na przykład metoda checkAccess(ThreadGroup *g*) wywoływana jest w chwili tworzenia nowej grupy watków (obiektu klasy ThreadGroup), ustawiania statusu grupy jako demona, zatrzymania (wykonania stop), itp. Kiedy więc tworzy się własny menadżer bezpieczeństwa, należy pamiętać w jakich sytuacjach nastąpi niejawne wywołanie metody check*XXX*(), a kiedy wywołanie to będzie jawne. Ponadto, w zależności od przyjętej strategii bezpieczeństwa można dokonac wyboru, które z metod check*XXX*() należy przesłonić, a które nie.

Instlacja menadżera bezpieczeństwa

Do zaimplementowanego menadżera bezpieczeństwa służy metoda setSecurityManager() z klasy System, przy czym metoda System.setSecurityManager() może być wywołana tylko raz w przeciągu całego czasu działania aplikacji. Kolejne wywołania powodowałyby zgłaszanie wyjątku SecurityException.

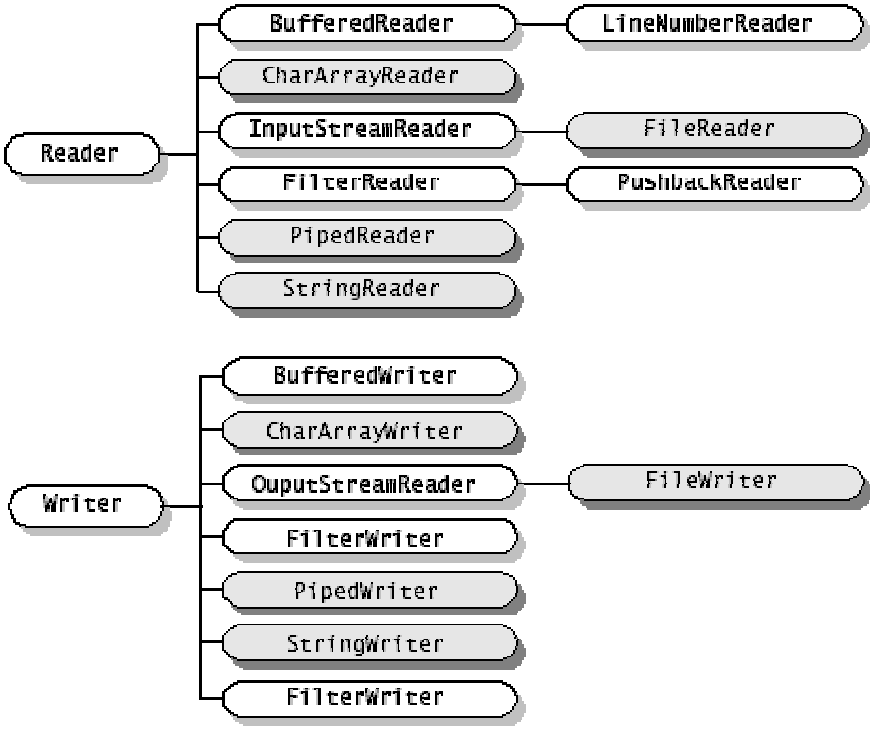
Przykład:
(zabezpieczenie otwarcia pliku do czytania i pisania: instalacja menadżera bezpieczeństwa)

```
import java.io.*;

public class SecurityManagerTest {
   public static void main(String[] args) throws Exception {

      BufferedReader buffy = new BufferedReader(
                              new InputStreamReader(System.in));

      try {
         System.setSecurityManager(
                     new PasswordSecurityManager("ThisIsPass", buffy));
      } catch (SecurityException se) {
         System.err.println("SecurityManager already set!");
      }

      BufferedReader in = new BufferedReader(new FileReader("inputtext.txt"));
      PrintWriter out = new PrintWriter(new FileWriter("outputtext.txt"));
      String inputString;
      while ((inputString = in.readLine()) != null)
         out.println(inputString);
      in.close();
      out.close();
   }
}
```

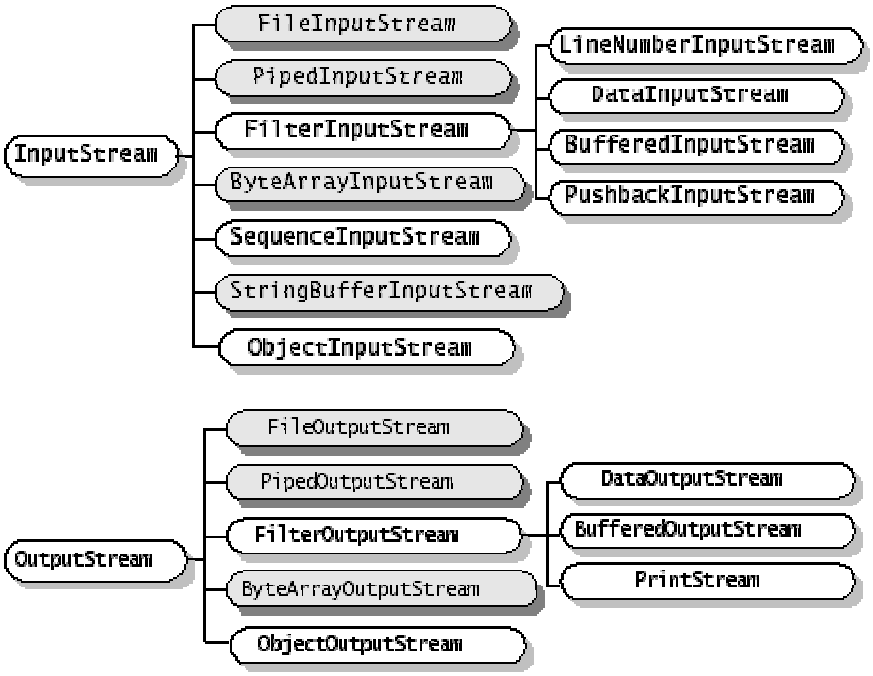
Po uruchomieniu powyższego przykładu użytkownik zostanie zapytany dwukrotnie o hasło: raz przy otwieraniu pierwszego z plików, drugi raz przy otwieraniu drugiego z plików. Jeśli wprowadzone hasło okaże się fałszywe, zostanie zgłoszony wyjątek i aplikacja zakończy się.

Strumienie

Strumienie znaków



Strumienie bajtów



Reader i InputStream dostarczają podobnych metod służących do odczytu danych:

Reader (czytanie znaków i tablic znaków)	InputStream (czytanie bajtów i tablic bajtów)
int read() int read(char cbuf[]) int read(char cbuf[], int offset, int length)	int read() int read(byte cbuf[]) int read(byte cbuf[], int offset, int length)

Writer i OutputStream dostarczają podobnych metod służących do zapisu danych:

Writer (pisanie znaków i tablic znaków)	OutputStream (pisanie bajtów i tablic bajtów)
int write(int c) int write(char cbuf[]) int write(char cbuf[], int offset, int length)	int write(int c) int write(byte cbuf[]) int write(byte cbuf[], int offset, int length)

Ponadto Reader i InputStream dostarczają metod:

- do zaznaczania położenia w strumieniu i resetowania położenia (mark() oraz reset() w powiązaniu z markSupported())
- opuszczania pewnej ilości danych (skip() – ile bajtów opuścić),
- określających ilość dostępnych danych (available() z InputStream) – ile bajtów dostępnych bez blokowania strumienia oraz ready() z Reader - czy są jakieś znaki do odczytania),
- zamykających jawnie strumień (close())

Domyślnie implementacje metod markSupported() oraz reset() w klasach InputStream i Reader działają w identyczny sposób: markSupportred() zwraca false, reset() zgłasza wyjątek IOException. Jednakże mark() klasy InputStream nie wykonuje żadnych czynności, zaś mark() klasy InputStream zgłasza wyjątek IOException.

Przykład:

```
import java.io.*;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");

        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);
        int c;

        while ((c = in.read()) != -1)
           out.write(c);

        in.close();
        out.close();
    }
}
```

```
DataInput d=new DataInputStream(new BufferedInputStream(new
FileInputStream("farrago.txt")));

try {
while (true) {
 byte b = (byte) d.readByte();
...
}
} catch (EOFException e) {
... //koniec pliku
}

String linia;
while ((linia=d.readLine()) != null) {
...
}
```

```
import java.io.*;
class PersonalHello {
  public static void main (String args[])
    {
      byte name[] = new byte[100];
      int n_read = 0;
      System.out.println("What is your name?");
      try {
        n_read = System.in.read(name);
        System.out.print("Hello ");
        System.out.write(name,0,n_read);
      }
      catch (IOException e) {
        System.out.print("Sorry.  can't catch your name.");
      }
    }
}
```

```
static int getNextInteger() {
    String line;
    DataInputStream in = new DataInputStream(System.in);
    try {
      line = in.readLine();
      int i = Integer.valueOf(line).intValue();
      return i;
    }
    catch (Exception e) {
      return -1;
    }
  } // getNextInteger ends here
```

StreamTokenizer, StringTokenizer

```
import java.io.*;
public class FormattedInput {
  public String stringRead() {
    try {
      for (int i=0; i<5;i++) {
        int tokenType = tokenizer.nextToken();

        // If input is a string of alphabets then accept it and return the string value.
        if (tokenType == tokenizer.TT_WORD ||  tokenType=='\"')
          return tokenizer.sval;
        // If input is ! then return it to signal the end of input.
        else if (tokenType=='!') return "!";
            else {
              // Strings with non-alphabets must be  specified within double quotes.
              System.out.println("Incorrect input.
                                  Reenter a 0string
                                  between double
                                  qoutes");
              continue;
            }
      }
      // Upto five failures are tolerated during
      // input.
      System.out.println("five failures reading a
                          string. Program
                          terminated");
      System.exit(1);
      return null;
    } catch (IOException e) {
          // Catch error generating IOException and
          // indicate the error condition.
          System.out.println(e);
          System.exit(1);
          return null;
      }
  }

  // Instance variable for the StreamTokenizer object
  // is initialized as a instance of input stream
  // object.
  private StreamTokenizer tokenizer=new
           StreamTokenizer(
                 new InputStreamReader(System.in));
}
```

Własności Systemowe (System Properties)

Klucz	**Znaczenie**	Dostępność dla apletów
"file.separator"	Separator plików (na przykład “/”)	+
"java.class.path"	Ścieżka dostępu do klas Javy	-
"java.class.version"	Numer wersji klas Javy	+
"java.home"	Katalog domowy instalacji Javy	-
"java.vendor"	Opis dostawcy (vendor) Javy	+
"java.vendor.url"	Adres URL dostawcy Javy	+
"java.version"	Numer wersji Javy	+
"line.separator"	Znak separatora linii	+
"os.arch"	Architektura systemu operacyjnego	+
"os.name"	Nazwa systemu operacyjnego	+
"os.version"	Wersja systemu operacyjnego	
"path.separator"	Separator ścieżki (na przykład “:”)	+
"user.dir"	Bieżący katalog roboczy użytkownika	-
"user.home"	Katalog domowy użytkownika	-
"user.name"	Nazwa konta użytkownika	-

Ustawianie własności systemowych

```
import java.io.FileInputStream;
import java.util.Properties;

public class PropertiesTest {
    public static void main(String[] args) throws Exception {

 // nadpisz własności systemowe własnościami zdefiniowanymi w pliku
// myProperies.txt  (np. java.vendor=Moja Firma)

        FileInputStream propFile = new FileInputStream(
                                    "myProperties.txt");

        Properties p = new Properties(System.getProperties());
        p.load(propFile);

        // set the system properties
        System.setProperties(p);
         // display new properties
        System.getProperties().list(System.out);
        FileOutputStream out = new FileOutputStream("newSysProperties.txt");
        p.store(out, "---No Comment---");
        out.close();
    }
}
```

Klasa Properties zdefiniowana jest w pakiecie java.util. Klasa ta zawiera zestaw par *klucz/wartość*. Dziedziczy ona z klasy HashtableProperties.

Metody tej klasy pozwalają na:

- wczytywanie par klucz/wartość ze strumienia
- odczytywanie wartości na podstawie klucza
- wylistowanie kluczy i wartości
- enumerację kluczy
- zapisywania własności do strumienia

Metody odziedziczone z HashtableProperties umożliwiają:

- sprawdzanie, czy istnieje zadana para klucz/wartość w obiekcie klasy Properties
- uzyskanie bieżącego liczby par klucz/wartość
- usunięcie klucza i wartości
- dodanie nowej pary klucz/wartość do listy własności
- odczytywanie wartości na podstawie klucza
- sprawdzenie, czy obiekt Properties jest pusty

setProperties: pozwala na modyfikacje własności i ich wartości. Jej argumentem jest zainicjalizowany obiekt klasy Properties zawierający pożądane zestaw własności systemowych
store: wymaga podania dwóch argumentów (strumienia i komentarza) (wprowadzona w Java 1.2)
contains(Object *value*)
containsKey(Object *key*) //
getProperty(String *key*)
getProperty(String *key*, String *default*)
list(PrintStream *s*)
list(PrintWriter *w*) // zapisuje wszystkie własności do strumienia albo writer
elements() // zwraca numerator (Enumeration)
keys() // zwraca enumerator
propertyNames() //
size()
put(Object *key*, Object *value*) // umieszcza parę klucz/wartość w obiekcie
remove(Object *key*) // usuwa parę klucz/wartość na podstawie klucza

Niektóre z powyższych metod akceptują Object, ale powinno się używać String

Swingowe „Hello World”


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

public class HelloWorldSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("HelloWorldSwing");
        final JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Kontenerami najwyższego poziomu są JFrame, JDialog, oraz JApplet (dla apletów). Każdy obiekt klasy JFrame implementuje główne okno, każdy obiekt klasy JDialog implementuje okno drugorzędne (okno, które jest zależne od innego okna). Każdy obiekt klasy JApplet implementuje pole wewnątrz okienka przetwarzającego (przeglądarki).

W powyższym przykładzie (Java 1.3) mamy

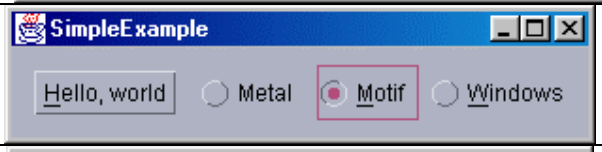
```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

w poprzednich wersjach Javy wymaga to stworzenia adaptera:

```
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
```

Swing pozwala na definiowanie wyglądu okienek (tzw. look and feel)

```
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel(

UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { }
    ...// Create and show the GUI...
}
```

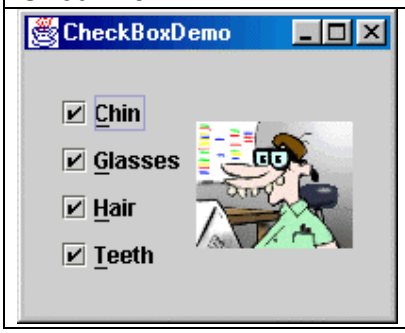
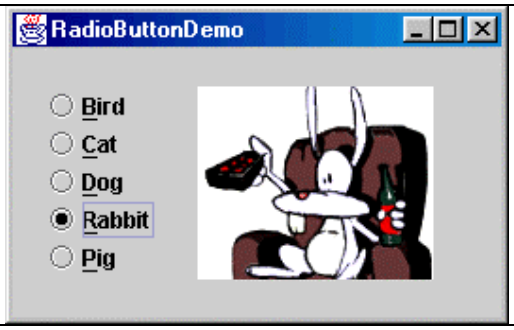
	Java look and feel
	CDE/Motif look and feel
	Windows look and feel

Przyciski i etykiety

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

```
...// where instance variables are declared:
private static String labelPrefix = "Number of button clicks: ";
private int numClicks = 0;

...// in GUI initialization code:
final JLabel label = new JLabel(labelPrefix + "0    ");
...
label.setLabelFor(button);

...// in the event handler for button clicks:
label.setText(labelPrefix + numClicks);
```

Potomkowie klasy AbstractButton

CheckBox	RadioBox
	

Obsługa zdarzeń

Implementacja obsługi zdarzeń wymaga wykonania trzech kroków.
1. Zadeklarowania klasy obsługującej zdarzenia jako klasy implementującej interfejs słuchacza (listener interface) lub też jako klasy dziedziczącej po klasie, która implementuje interfejs słuchacza

```
public class MyClass implements ActionListener {
```

2. Zarejestrowania instancji klasy obsługującej zdarzenia w jednym lub kilku komponentach

```
someComponent.addActionListener(instanceOfMyClass);
```

3. Zaimplementowania metod interfejsu słuchacza w zadeklarowanej klasie

```
public void actionPerformed(ActionEvent e) {
  ...//code that reacts to the action...
}
```

Schemat obsługi zdarzenia:

Klasy obsługujące zdarzenia mogą być instancjami dowolnych klas. Choć często są one implemenetowane w kilku linijkach kodu jako anonimowe klasy wewnętrzne (*anonymous inner class)*

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

});
```

Tabela przykładowych zdarzeń wraz z listą obsługujących je słuchaczy

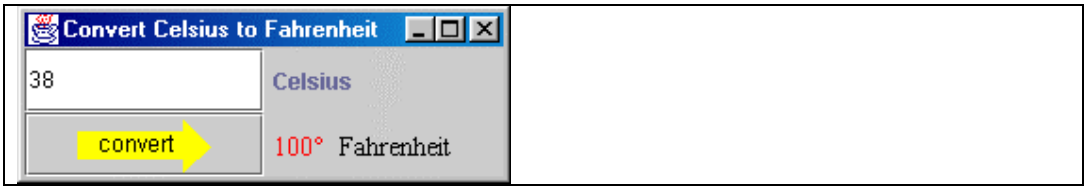
zdarzenie	**słuchacz**
kliknięcie na przycisku, naciśnięcie klawisza Enter w polu tekstowym, wybór opcji z menu	ActionListener
zamknięcie ramki (głównego okna)	WindowListener
kliknięcie klawiszem myszki nad komponentem	MouseListener
przesunięcie kursora myszki nad komponentem	MouseMotionListener
uwidocznienie się komponentu	ComponentListener
uzyskanie przez komponent strumienia wprowadzania z klawiatury	FocusListener
zmiana w tabeli lub liście wyboru	ListSelectionListener

Dodawanie otoczki do komponentów

```
panel.setBorder(BorderFactory.createEmptyBorder(
                    30, //top
                    30, //left
                    10, //bottom
                    30) //right
                    );
```

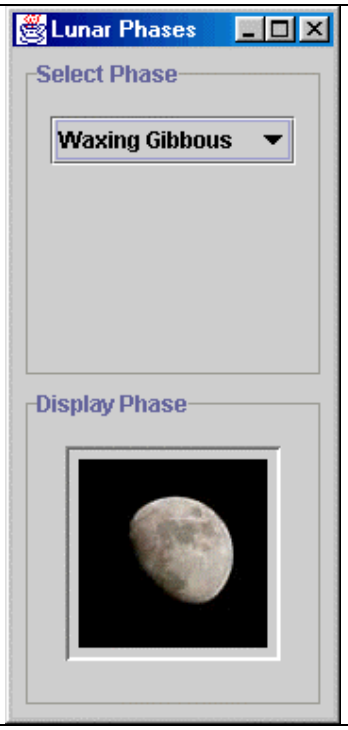
Korzystanie z ikon i tekstu html

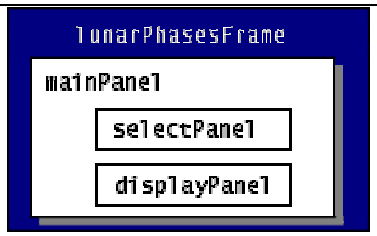

```
JTextField tempCelsius = null;
...
tempCelsius = new JTextField(5); //5 – liczba kolumn w polu
tekstowym

JButton convertTemp;
...
convertTemp.addActionListener(this);
...
public void actionPerformed(ActionEvent event) {
    // Parse degrees Celsius as a double and convert to Fahrenheit.
    int tempFahr = (int)((Double.parseDouble(tempCelsius.getText()))
                        * 1.8 + 32);
    fahrenheitLabel.setText(tempFahr + " Fahrenheit");
}
//In the constructor for a JDialog subclass:
getRootPane().setDefaultButton(setButton);
```




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


// Set fahrenheitLabel to new value and font color based on temperature.
// Swing > 1.1
if (tempFahr <= 32) {
    fahrenheitLabel.setText("<html><font color=blue>" + tempFahr
                            + "&#176  Fahrenheit </font></html>");

} else if (tempFahr <= 80) {
    fahrenheitLabel.setText("<html><font color=green>" + tempFahr
                            + "&#176  Fahrenheit </font></html>");

} else {
    fahrenheitLabel.setText("<html><font color=red>" + tempFahr
                            + "&#176  Fahrenheit </font></html>");
}

// setFont also can be used
```

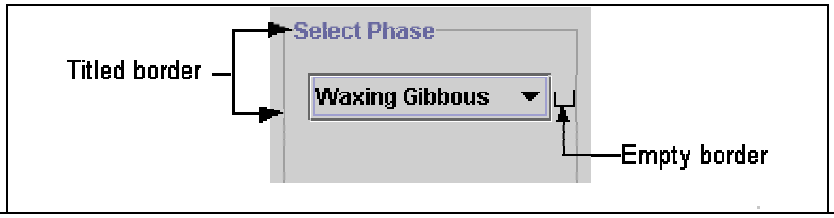
Compound borders, combo boxes, loading multiple images





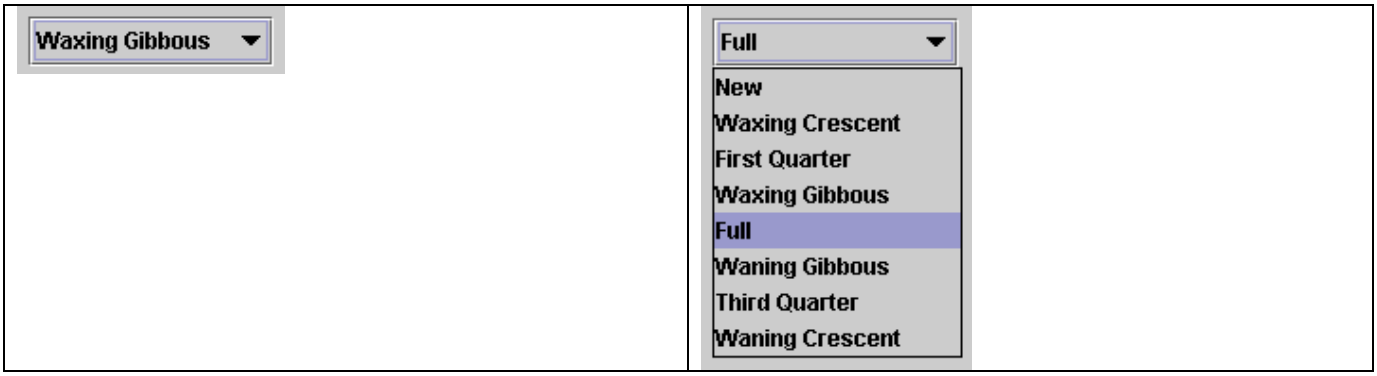

```
// Create the phase selection and display panels.
selectPanel = new JPanel();
displayPanel = new JPanel();

// Add various widgets to the subpanels.
addWidgets();

// Create the main panel to contain the two subpanels.
mainPanel = new JPanel();

mainPanel.setLayout(new GridLayout(2,1,5,5));
mainPanel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

// Add the select and display panels to the main panel.
mainPanel.add(selectPanel);
mainPanel.add(displayPanel);
```

Złożone obramowania




```
// Add border around the select panel
selectPanel.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createTitledBorder("Select Phase"),
                BorderFactory.createEmptyBorder(5,5,5,5)));
```

Combo Boxes



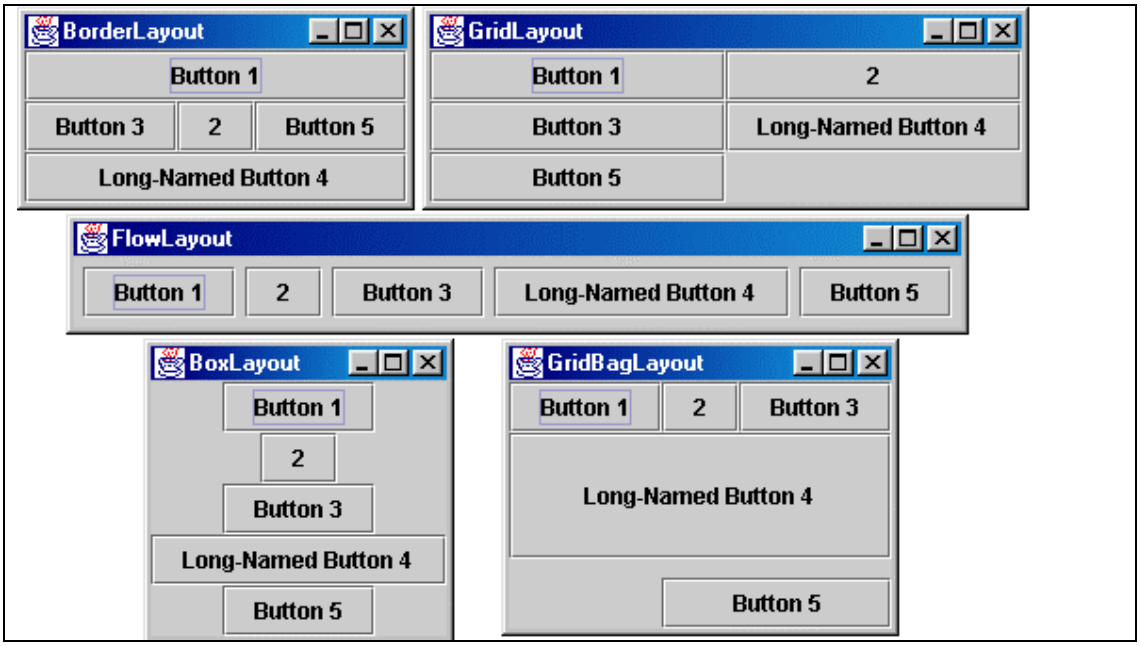

```
JComboBox phaseChoices = null;
...
// stworzenie combo box
String[] phases = { "New", "Waxing Crescent", "First Quarter",
                    "Waxing Gibbous", "Full", "Waning Gibbous",
                    "Third Quarter", "Waning Crescent" };
phaseChoices = new JComboBox(phases);
phaseChoices.setSelectedIndex(START_INDEX);

// obsługa zdarzeń
phaseChoices.addActionListener(this);
...
public void actionPerformed(ActionEvent event) {
    if ("comboBoxChanged".equals(event.getActionCommand())) {
        // update the icon to display the new phase
        phaseIconLabel.setIcon(images[phaseChoices.getSelectedIndex()]);
    }
}
```

:

Radio Buttons

```
final int numButtons = 4;
JRadioButton[] radioButtons = new JRadioButton[numButtons];

final ButtonGroup group = new ButtonGroup();
...

final String defaultMessageCommand = "default";
final String yesNoCommand = "yesno";
final String yeahNahCommand = "yeahnah";
final String yncCommand = "ync";

radioButtons[0] = new JRadioButton("<html>Candidate 1:
     <font color=red>Sparky the Dog</font></html>");
radioButtons[0].setActionCommand(defaultMessageCommand);

…….
radioButtons[3] = new JRadioButton("<html>Candidate 4:
     <font color=maroon>Duke the Java<font size=-2><sup>TM</sup>
     </font size> Platform Mascot</font></html>");
radioButtons[3].setActionCommand(yncCommand);

for (int i = 0; i < numButtons; i++) {
   group.add(radioButtons[i]);
}

// Select the first button by default.
radioButtons[0].setSelected(true);
```

Zarządzanie wyglądem

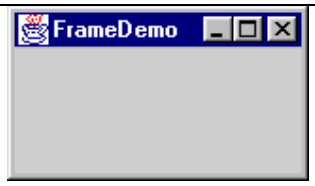
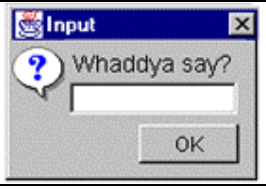


Menadżerowie położenia

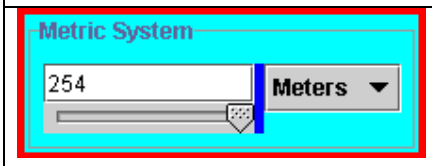
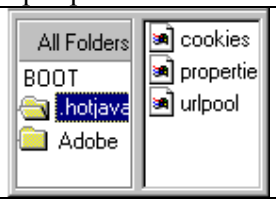
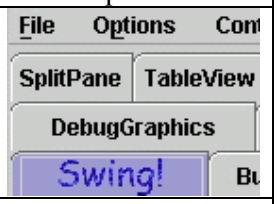
Domyślnie mamy: Panel -> FlowLayout, JApplet, JDialog, JFrame -> BorderLayout.
Jednak można to zmienić:

```
JPanel pane = new JPanel();
pane.setLayout(new BorderLayout());
```

Top-Level Containers

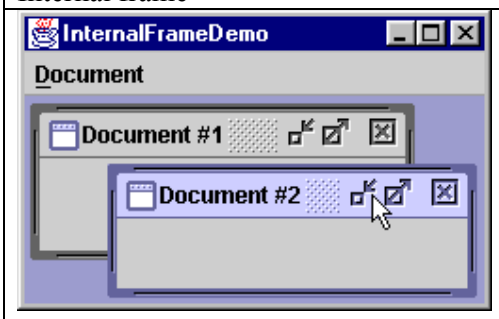
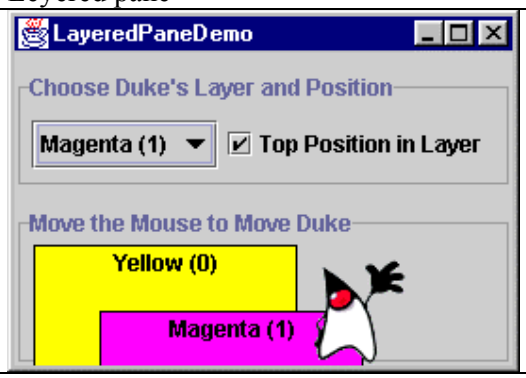
Applet	Frame	Dialog
		

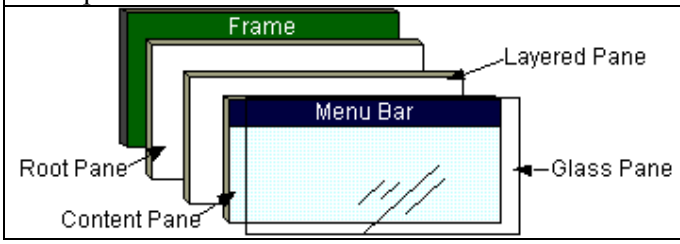
General-Purpose Containers

Panel	Scroll pane	Split pane	Tabbed pane
			

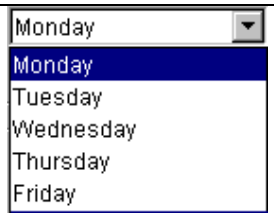
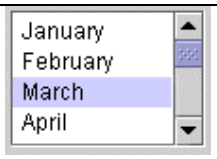
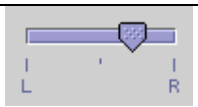
Tool bar


Special-Purpose Containers

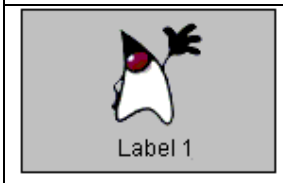
Internal frame	Leyered pane
	

Root pane


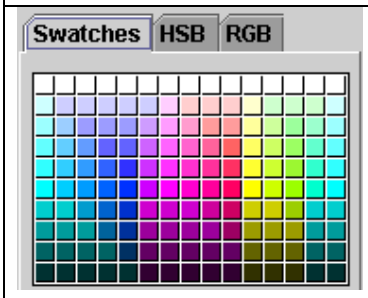
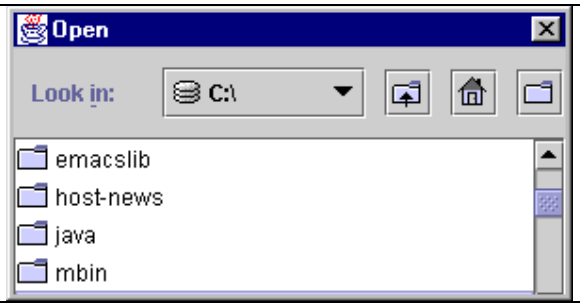
Basic Controls

Buttons	Combo box	List
		
Menu	**Slider**	**Text Fields**
		

Uneditable Information Displays

Label	Tool tip	Progress bar
		

Editable Displays of Formatted Information

Color chooser	File chooser	Table
		
Text	**Tree**