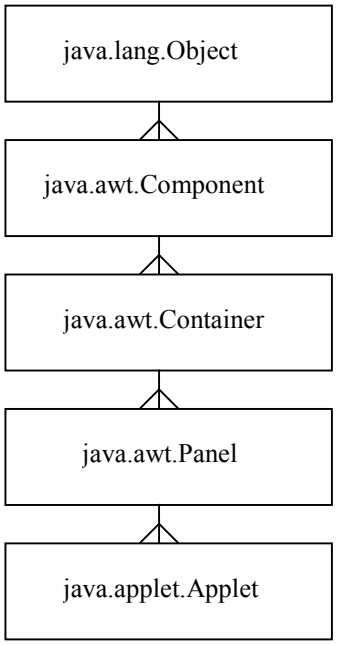
Rozdział 5 Aplety, grafika w Javie

5.1 Aplety
5.2 Grafika w Javie
5.2.1 Komponenty
5.2.2 Kontenery
5.2.3 Rozkłady
5.2.4 Zdarzenia

5.1 Aplety

Aplet jest programem komputerowym, stworzonym w ten sposób, że możliwe jest jego wykonywania tylko z poziomu innej aplikacji. Oznacza to, że aplet nie jest samodzielnym programem. Jak wspomniano już na początku tego materiału, definicja apletu odbywa się poprzez definicję klasy dziedziczącej z klasy Aplet. Konieczne jest więc jawne importowanie pakietu *java.applet.** zawierającego tylko jedną klasę *Applet*. Klasa *Applet* dziedziczy w następujący sposób z klasy *Object*:



Rysunek 5.1 Dziedziczenie dla klasy Applet.

Nadklasą klasy *Applet* jest klasa Panel zdefiniowana w graficznym *pakiecie AWT – Abstract Window Toolkit*. Łatwo się więc domyśleć, że również i aplet ma formę graficzną. Można w dużym uproszczeniu powiedzieć, że pole robocze apletu jest oknem graficznym, w którym wszystkie operacje edycyjne wykonuje się przez odwołanie do obiektu graficznego klasy *Graphics*. AWT zostanie omówiona poniżej i tam również zaprezentowane zostaną liczne przykłady apletów związane z grafiką.

Jak już wspomniano aplet jest programem wykonywany pod kontrolę innej aplikacji. Przykładową aplikacją może być przeglądarka stron HTML (np. Netscae Navigator, MS Internet Explorer, itp.). Oznacza to, że konieczne jest umieszczenie wywołania kodu apletu w ciele strony HTML obsługiwanej przez daną przeglądarkę. W tym celu

wprowadzono w HTML następujące tagi: <APPLET> oraz </APPLET> oznaczające odpowiednio początek i koniec pól definiujących aplet. Podstawowym polem (atrybutem) jest pole CODE informujące aplikację czytającą HTML gdzie znajduje się skompilowany kod apletu. Przykładowo w omawianym już przykładzie 1.4 atrybut CODE zdefiniowano następująco: *code=Jedi2.class.* W przykładzie tym aplikacja ma pobrać kod apletu umieszczony w pliku o nazwie Jedi2.class znajdujący się w aktualnym katalogu. W celu wskazania ewentualnego katalogu, w którym umieszczono kod apletu można posłużyć się polem CODEBASE, np. *code=Jedi2.class codebase=klasy*. Aplet może posiadać swoją nazwę określoną w kodzie HTML poprzez wykorzystanie pola NAME, np. *code=Jedi2.class name=aplecik.* Inne bardzo popularne pola wykorzystywane do opisu apletu to WIDTH oraz HEIGHT. Oba parametry oznaczają rozmiar (szerokość i wysokość) pola pracy apletu zawartego na stronie WWW. Przykładowo *code=Jedi2.class width=200 height=100* oznacza ustalenie pola pracy apletu na stronie WWW o szerokości 200 pikseli i wysokości 100 pikseli. Możliwe jest również sterowanie położeniem okna względem tekstu poprzez przypisanie odpowiedniej wartości do pola ALIGN, takich jak: LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM oraz ABSBOTTOM np. align=middle. Czasami przydatne jest odseparowanie pola apletu od otaczającego tekstu. Wykonuje się to poprzez właściwe ustawienie liczby pikseli stanowiących spację w poziomie i pionie, czyli poprzez wykorzystanie pól: HSPACE i VSPACE. Ponieważ z różnych przyczyn nie zawsze przeglądarka będzie w stanie uruchomić aplet dlatego warto wprowadzić zastępczy tekst co wykonuje się poprzez przypisanie tekstu do pola ALT, np. alt=”Aplet rysujący znak firmowy”

W pracy z apletami istotny jest również fakt, że często koniecznie trzeba wykorzystać kilka różnych plików związanych z apletem (np. kilka klas, obrazki, dźwięki, itp.). Efektywnie wygodne jest powiązanie wszystkich plików związanych z danym apletem w jeden zbiór (np. eliminacja czasu nawiązania połączeń osobno dla każdego pliku HTTP 1.0). Java daje taką możliwość poprzez stworzenie archiwum narzędziem dostarczanym w dystrybucji JDK a mianowicie: JAR. Wywołanie polecenia jar z odpowiednimi przełącznikami i atrybutami umożliwia liczne operacje na archiwach. Przykładowe opcje:

-c stwórz nowe archiwum
-t pokaż zawartość archiwum
-x pobierz podane pliki z archiwum
-u aktualizuj archiwum
-f określenie nazwy archiwum
-O stwórz archiwum bez domyślnej kompresji plików metodą ZIP

W celu stworzenia archiwum z wszystkich klas zawartych w bieżącym katalogu i nadać mu nazwę JediArchiwum można wywołać polecenie:

```
jar -cf JediArchiwum *.class
```

Tak stworzone archiwum można przejrzeć: jar –tf JediArchiwum, lub odtworzyć: jar –xf JediArchiwum.

W celu wykorzystania apletu, którego kod jest zawarty w pliku archiwum trzeba wykorzystać pole ARCHIVES i nadać mu wartość nazwy pliku archiwum np.: code=Jedi2 archives=JediArchiwum.jar. Należy pamiętać o tym, że w polu code podaje się tylko nazwę klasy apletu bez rozszerzenia *.class.

W celu demonstracji omówionych zagadnień rozbudujmy aplet, a właściwie wywołujący go kod HTML z przykładu 1.4:

Przykład 5.1:

```
// Jedi2.java :

import java.applet.Applet;
import java.awt.*;

public class Jedi2 extends Applet{

        public void paint(Graphics g){
                g.drawString("Rycerz Luke ma niebieski miecz.„, 15,15);
        }

} // koniec public class Jedi2.class extends Applet
```

Jedi2n.html *:*

```
<html>
<head>
<title> Przykładowy aplet</title>
</head>

<body>
<applet code=Jedi2.class name= aplecik
width=200 height=100
alt="Aplet drukujący tekst"
align=middle
hspace=5 vspace=5>
Tu pojawia się aplet
</applet>
</body>
</html>
```

Łatwo zauważyć, że opis i znaczenie pól dla elementu APPLET w HTML jest podobne jak dla elementu IMG. Spośród wymienionych pól konieczne do zastosowania w celu opisu apletu są pola CODE oraz WIDTH i HEIGHT. Te ostatnie dwa są wymagane przez program appletviewer, niemniej w niektórych przeglądarkach istnieje możliwość wprowadzenia opisu apletu bez podania rozmiaru okna apletu. Czy będzie wówczas wygenerowane domyślne okno? Tak, obszarem okna będzie wolne pole edycyjne dostępne przeglądarce (zależne od rozdzielczości ekranu). Oczywiście nie jest to właściwe działanie, i ważne jest ustawianie pól WIDTH oraz HEIGHT.

Poniższy program ukazuje możliwość wykorzystania nazwy apletu oraz pobrania rozmiaru apletu:

Przykład 5.2:

```
//Jedi2nn.java

import java.applet.Applet;
import java.awt.*;

class Sith2n{
        Applet app;
        Sith2n(Applet a){
                this.app=a;
        }
        public Dimension rozmiar(){
                Dimension d = app.getSize();
                return d;
        }

}// koniec class Sith

public class Jedi2n extends Applet{

        public void paint(Graphics g){
                g.drawString("Rycerz Luke ma niebieski miecz.", 15,15);
                Sith2n s = new Sith2n(this);
                Dimension d = s.rozmiar();
                String roz = new String("Szerokość="+d.width+
                "; wysokość="+d.height);
                g.drawString(roz,15,30);
                g.drawString(info(),15,45);
        }
        public String info(){
                Applet aplet=getAppletContext().getApplet("aplecik");
                return (aplet.getCodeBase()).toString();
        }

} // koniec public class Jedi2n extends Applet

//-----------------------------------------------------------
//Jedi2nn.html:

<html>
<head>
<title> Przykładowy aplet</title>
</head>

<p> Oto aplet: </p>
<body>
<applet code=Jedi2n.class name= aplecik
width=200 height=100
alt="Aplet drukujący tekst"
align=middle
hspace=5 vspace=5 >
```

```
Tu pojawia się aplet
</applet>
</body>
</html>
```

W przykładzie tym zastosowano metodę *getSize*() zwracającą obiekt klasy *Dimension* zawierający pola szerokości (width) i wysokość (height) opisujące rozmiar okna apletu. Metoda *info*() przekazuje tekst opisujący adres źródłowy apletu, którego obiekt jest uzyskiwany przez odwołanie się do nazwy apletu. Jest to typowo edukacyjny przykład (metoda *info*() mogła by być` zrealizowana tylko poprzez odwołanie się do *this.getCodeBase()).*

Powyższy kod został rozbity na dwie klasy celem dodatkowego pokazania możliwości pracy z archiwami. W celu stworzenia archiwum składającego się z dwóch klas Jedi2n.class oraz Sith.class należy wywołać program jar:

```
jar -cf JediArchiwum.jar Jedi2n.class Sith2n.class
```

Po stworzeniu archiwum warto sprwdzić jego zawartość poprzez:

```
jar -tf JediArchiwum.jar
```

Tak przygotowane archiwum możemy wykorzystać do konstrukcji kodu HTML wywołującego aplet Jedi2n:

```
//Jedi2nnn.html

<html>
<head>
<title> Przykładowy aplet</title>
</head>

<p> Oto aplet: </p>
<body>
<applet code=Jedi2n name= aplecik archives=JediArchiwum.jar
width=200 height=100
alt="Aplet drukujący tekst"
align=middle
hspace=5 vspace=5 >
Tu pojawia się aplet
</applet>
</body>
</html>
```

Oczywiście efekt działanie jest taki sam jak poprzednio.

Ostatnie pole w opisie apletu, które warto omówić to pole przechowujące parametr apletu. Parametr jest skonstruowany poprzez dwa argumenty: *name* oraz *value*. W polu *name* parametr przechowuje swoją nazwę, natomiast w polu *value* swoją wartość. Konstrukcja parametru jest następująca:
<param name=Nazwa value=Wartość>
np.:

```
<applet code=Jedi2n.class width=300 hight=100>
<param name= Opis value=”To jest aplet”>
</applet>
```

Dla obu pól parametru istotna jest wielkość znaków. Jeżeli pole wartość przechowuje spację lub znaki specjalne musi być ujęte w cudzysłowie. Ilość parametrów jest dowolna. W kodzie apletu można pobrać wartość parametru o danej nazwie wykorzystując metodę *getParameter*() z argumentem stanowiącym nazwę parametru, np. *getParameter(„Opis”).*

Przydatną metodą klasy *Applet* jest metoda *showStatus().* Podawany jako argument tej metody tekst jest wyświetlany w oknie statusu przeglądarki. Jest to często bardzo wygodna metoda powiadamiania użytkownika o działaniu apletu. Omawiane wcześniej (rozdział 1) metody obsługi apletu (*init(), start(), stop(), paint(), destroy())* pozwalają się domyśleć jaką formę przybiera każdy aplet. Otóż każdy aplet jest właściwie wątkiem (podobieństwo start(), stop()) należącym do domyślnej grupy o nazwie applet-tu_nazwa_klasy_z_rozszerzeniem, z domyślnym priorytetem 5 o domyślnej nazwie Thread-tu_kolejny_numer. Metody start() i stop() apletu nie odpowiadają oczywiście metodą o tych samych nazwach zdefiniowanych w klasie Thread. Należy więc pamiętać o tym, że zmiana strony WWW w przeglądarce wcale nie oznacza zakończenia pracy apletu. Zmiana strony oznacza jedynie wywołanie metody stop() apletu czyli żądanie zawieszenia aktywności apletu (np. zatrzymanie animacji) - przejście od statusu *start* do *init*. Standardowo metoda stop() nic nie robi. Jeżeli jest potrzebna jakaś akcja przy zmianie stron wówczas musi być ona zdefiniowana w ciele metody stop(). Podobnie jest z metodą *destroy*(), z tym, że wołana jest ona bezpośrednio przez zakończeniem pracy apletu (np. koniec pracy przeglądarki). Omawiane metody obsługi apletu są więc wywoływane przez aplikację kontrolującą aplet w wyniku wystąpienia określonych zdarzeń, i mogą być przedefiniowane przez programistę celem wprowadzenia zamierzonego działania.

Należy tu podkreślić bardzo ważny element w korzystaniu z przeglądarek w pracy z apletami. Aby uruchomić aplet dana przeglądarka musi być zgodna, tzn., dostarczać Maszynę Wirtualną do obsługi kodu bajtów. Obsługiwana wersja Javy może być łatwo uzyskana poprzez uruchomienie konsoli Javy, którą stanowi opcjonalne okno przeglądarki. W celu uniezależnienia się od zaimplementowanej wersji Javy w danej przeglądarce Sun opracował i udostępnia plug-in Javy. Po zainstalowaniu wtyczki i odpowiednim przygotowaniu kodu HTML (*HTMLconverter*) można uruchomić praktycznie każdy dobrze napisany aplet, nawet dla najnowszej wersji JDK obsługiwanej zazwyczaj przez plug-in. Przygotowanie kodu HTML zawierającego aplet do pracy z wtyczką wymaga znacznych zmian w opisie apletu. Jest to spowodowane tym, że w części kodu HTML gdzie był uruchamiany aplet musi być uruchamiana wtyczka, której parametrami wywołania są kod klasy apletu, nazwa apletu oraz typ elementu. Dotychczasowe wywołanie apletu przez tag <APPLET> musi być usunięte lub wzięte w komentarz. Pomocnym narzędziem wykonującym automatyczną konwersję kodu HTML do wersji zgodnej z wtyczką jest aplikacja Javy: *HTMLConverter*. Sun dostarcza tę aplikację w celu szybkiego dostosowania kodu HTML, bez konieczności ręcznej modyfikacji źródła HTML. *HTMLConverter* może dokonać konwersji pojedynczego pliku HTML lub całej serii plików wykonując przy tym kopie zapasowe wersji oryginalnych w jednym z katalogów. Przykładowa treść

pliku HTML prezentowanego wcześniej (Jedi2n.html) po konwersji będzie wyglądała następująco:

```
//Jedi2n.html po konwersji

<html>
<head>
<title> Przykładowy aplet</title>
</head>

<body>
<p> To jest aplet </p>
<!--"CONVERTED_APPLET"-->
<!-- CONVERTER VERSION 1.0 -->
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 200 HEIGHT = 100 NAME = aplecik ALIGN = middle VSPACE = 5 HSPACE = 5 ALT = "Aplet drukujący tekst"  codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-win32.cab#Version=1,2,0,0">
<PARAM NAME = CODE VALUE = Jedi2.class >
<PARAM NAME = NAME VALUE = aplecik >

<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.2" java_CODE = Jedi2.class ALT = "Aplet drukujący tekst" NAME = aplecik WIDTH = 200 HEIGHT = 100 ALIGN = middle VSPACE = 5 HSPACE = 5 pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html"><NOEMBED></COMMENT>

</NOEMBED></EMBED>
</OBJECT>

<!--
<APPLET  CODE = Jedi2.class WIDTH = 200 HEIGHT = 100 NAME = aplecik ALIGN = middle VSPACE = 5 HSPACE = 5 ALT = "Aplet drukujący tekst" >

</APPLET>
-->
<!--"END_CONVERTED_APPLET"-->

<p> Fajny no nie </p>
</body>
</html>
```

Wyraźnie widać podział pliku na część związaną z tagiem OBJCET oraz na część wziętą w komentarz związaną z tagiem APPLET. Oczywiście aby można była skorzystać z tego kodu należy zainstalować wtyczkę do przeglądarki. Kod instalacyjny można pobrać z serwera opisanego w polu pluginspage równym "http://java.sun.com/products/plugin/1.2/plugin-install.html.

Powyżej omówione zostały podstawowe zagadnienia związane tylko z apletami. Bardzo ważne są również inne tematy dotyczące apletów jak np. wykorzystanie grafiki, bezpieczeństwo, itp., które będą systematycznie w tych materiałach omawiane.

5.2 Grafika w Javie

Biblioteka Abstract Window Toolkit - AWT jako historycznie pierwsza w JDK dostarcza szereg elementów wykorzystywanych do tworzenia interfejsów graficznych dla potrzeb komunikacji z
użytkownikiem i obsługi jego danych. Do najbardziej istotnych elementów tego pakietu należy zaliczyć:

komponenty (Components {widgets}),
rozkłady (Layout managers),
zdarzenia (Events),
rysunki i obrazy (Drawings and images).

Od wersji JAVA JDK 2 wprowadzono istotne rozszerzenia AWT związane z przygotowaniem GUI. Podstawowe elementy tej rozbudowy to pakiety *javax.swing* i tak zwany zestaw Java2D rozszerzający klasy biblioteki *java.awt*. Do najbardziej istotnych elementów *javax.swing* i Java2D należy zaliczyć:
- nowe, rozbudowane graficznie komponenty w SWING,
- nowe rozkłady i metody pracy z kontenerami,
- rozbudowa biblioteki *java.awt.image*,
- dodanie nowych klas do biblioteki *java.awt* (np. Graphics2D).

Platforma 2 Javy w sposób znaczny zmienia możliwości tworzenia aplikacji stąd krótko zostaną omówione podstawowe elementy pakietu AWT i SWING wraz z
ich porównaniem.

5.2.1 Komponenty

Komponenty to podstawowe elementy graficzne aktywne lub bierne służące do tworzenia interfejsu graficznego. Komponenty są reprezentowane przez klasy głównie dziedziczące z klasy *Component:*
Box.Filler, Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent

Każdy komponent posiada odpowiednio skonstruowaną klasę i metody, które opisane są w dokumentacji API.
Klasa *Component* stanowi klasę abstrakcyjną tak więc nie odwołuje się do niej bezpośrednio (przez stworzenie obiektu za pomocą *new*) lecz przez klasy potomne. Klasa *Component* dostarcza szereg ogólnie wykorzystywanych metod np.:

public Font getFont() - zwraca informacje o czcionce wybranej dla komponentu
public Graphics getGraphics() - zwraca kontekst graficzny komponentu potrzebny przy wywoływaniu metod graficznych
public void paint(Graphics g) - rysuje komponent,
itp.

Jednym z komponentów jest kontener reprezentowany przez klasę Container. Kontener jest komponentem, w którym można umieszczać inne komponenty.

Przykładowym kontenerem wykorzystywanym dotąd w tym materiale w przykładach jest *Applet* (dziedziczy z klasy Panel, a ta z klasy *Container*). Podstawowe klasy (kontenery) dziedziczące z klasy *Container* to:
BasicSplitPaneDivider, Box, CellRendererPane, DefaultTreeCellEditor.EditorContainer, JComponent, Panel, ScrollPane, Window

Najczęściej wykorzystywane kontenery w AWT to *Panel* oraz *Window*. Ten ostatni nie jest wprost wykorzystywany lecz poprzez swoje podklasy:
BasicToolBarUI.DragWindow, Dialog, Frame, JWindow

Klasa *Frame* jest kontenerem bezpośrednio wykorzystywanym do tworzenia okien graficznych popularnych w GUI. *Dialog* jest kontenerem dedykowanym komunikacji z użytkownikiem i stanowi okno o zmienionej funkcjonalności względem *Frame* (brak możliwości dodania paska Menu).

Podstawowa praca w konstrukcji interfejsu graficznego w Javie polega na:
- zadeklarowaniu i zdefiniowaniu kontenera,
- zadeklarowaniu komponentu,
- zdefiniowaniu (zainicjowanie) komponentu
- dodanie komponentu do kontenera.

Ostatni krok w konstrukcji interfejsu związany jest ze sposobem umieszczania komponentów w kontenerze. Sposób te określany jest w Javie rozkładem (Layout). Z każdym kontenerem jest więc związany rozkład, w ramach którego umieszczane są komponenty.

Zanim omówione zostaną kontenery i rozkłady warto zapoznać się z typami komponentów, które w tych kontenerach mogą być umieszczane.
Do podstawowych komponentów należy zaliczyć:
a.) dziedziczące pośrednio i bezpośrednio z klasy *Component*:
- *Label* – pole etykiety
- *Button* – przycisk
  Canvas – pole graficzne
  Checkbox – element wyboru (logiczny)
  Choice -element wyboru (z listy)
  List – lista elementów
  Scrollbar – suwak
  TextComponent:
    TestField: pole tekstowe
    TextArea: obszar tekstowy

b) nie dziedziczące z klasy *Component*:
    MenuBar – pasek menu,
    MenuItem - element menu:
        Menu – menu (zbiór elementów)
        PopupMenu – menu podręczne.

Komponent, którego klasa nie dziedziczy z klasy *Component* to *MenuComponent*. Klasa ta jest nadklasą komponentów: *MenuBar*, *MenuItem*. Ta ostatnia dostarcza również poprzez klasy dziedziczące z niej następujące komponenty: Menu oraz *PopupMenu*. Łącznie cztery klasy tj. *MenuBar*, *MenuItem*, Menu oraz *PopupMenu* są

wykorzystywane do tworzenia menu programu graficznego.

Wszystkie komponenty biblioteki AWT są dobrze opisane w dokumentacji (Java API) Javy, i dlatego nie będą szczegółowo omawiane w tym materiale. Rozpatrzmy jedynie metodę tworzenia komponentu i dodawania go do kontenera. Jedynym kontenerem poznanym do tej pory był *Applet* i dlatego będzie on wykorzystany w poniższych przykładach.

Przykładowy komponent - *Button* - umożliwia stworzenie przycisku graficznego wraz z odpowiednią etykietą tekstową. Stworzenie obiektu typu *Button* polega na implementacji prostej instrukcji:

```
Button przyciskKoniec = new Button("Koniec");
```

Dodanie tak stworzonego elementu do tworzonego interfejsu wymaga jeszcze wywołania metody:

```
add(przyciskKoniec);
```

Po uaktywnieniu interfejsu przycisk będzie widoczny dla użytkownika. W podobny sposób tworzy się obiekty pozostałych klas komponentów, manipuluje się ich metodami dla uzyskania odpowiedniego ich wyglądu, oraz dodaje się je do tworzonego środowiska graficznego.

Przykładowy kod źródłowy dla komponentu *Button* może wyglądać w następujący sposób:

Przykład 5.3:

```
//Ognia.java:

import java.awt.*;
import java.applet.*;

public class Ognia extends Applet {

                public void init() {
                   Button przyciskTest = new Button("przycisnij mnie...");
                   add(przyciskTest);
                }
}// koniec public class Ognia

//************************************

Ognia.html :
...
<APPLET  CODE="Ognia.class" WIDTH=70 HEIGHT=40 ALIGN=CENTER></APPLET>
```

Praca z pozostałymi komponentami wygląda podobnie. Poniższy przykład demonstruje różne typy komponentów.

Przykład 5.4:

```
//Pulpit.java:

public class Pulpit extends Applet {

        public void init() {
                Button przyciskTest = new Button("ognia...");
                 add(przyciskTest);

                 Label opis = new Label("Strzelac");
                add(opis);

                Checkbox rak = new Checkbox("Rakieta");
                Checkbox bomb = new Checkbox("Bomba");
                add(rak);
                add(bomb);

                Choice kolor = new Choice();
                kolor.add("zielona");
               kolor.add("czerwona");
               kolor.add("niebieska");
                add(kolor);

                List lista = new List(2, false);
                  lista.add("mała");
                  lista.add("malutka");
                  lista.add("wielka");
                  lista.add("duża");
                  lista.add("ogromna");
                  lista.add("gigant");
                add(lista);

                TextField param = new TextField("Podaj parametry");
                add(param);

        }
}// koniec public class Pulpit

//*******************************************************

<html>
<APPLET  CODE="Pulpit.class" WIDTH=800 HEIGHT=80 ALIGN=CENTER></APPLET>
</html>
```

Komponenty związane z menu są często wykorzystywane przy konstrukcji GUI, niemniej wymagają kontenera typu *Frame*, dlatego zostaną omówione później (wraz z tym kontenerem).

Pakiet *javax.swing* dostarcza szeregu zmodyfikowanych i nowych kontenerów i komponentów. Między innymi zmodyfikowano system umieszczania komponentów w kontenerach (co jest omówione dalej) oraz dodano możliwość wyświetlania ikon na komponentach. Przykładowe komponenty z pakietu *javax.swing* można znaleźć w dokumentacji SWING. Wszystkie komponenty tej biblioteki są reprezentowane przez

odpowiednie klasy, których nazwa zaczyna się od J, np. *JButton*. Komponenty biblioteki SWING dziedziczą z AWT (np. z klasy *Component*). Podstawowe komponenty tej biblioteki są podobne jak dla AWT z tym, że oznaczane z dodatkową literą J. Wprowadzono również nowe i przeredagowane elementy jak np.:

JColorChooser – pole wyboru koloru
JFileChooser – pole wyboru pliku
JPasswordField – pole hasła,
JProgressBar – pasek stanu
JRadioButton – element wyboru
JScrollBar - suwak
JTable - tabela
JTabbedPane - pole zakładek
JToggleButton –przycisk dwu stanowy
JToolBar – pasek narzędzi
JTree – lista w postaci drzewa

Implementacja komponentów SWING jest podobna do implementacji komponentów zdefiniowanych w AWT.

Przykład 5.5:

```
//Przycisk.java:

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

public class Przycisk extends JApplet {

        public void init() {
                String s. = getCodeBase().toString()+"/"+"jwr2.gif";
                /* dla apletu uruchamianego z serwera HTTP
                w przeciwnym wypadku może nastąpić błąd bezpieczeństwa
                ➔ próba odczytu z dysku */
                JButton przyciskTest = new JButton(new ImageIcon(s));
                 getContentPane().add(przyciskTest);


        }
}// koniec public class Przycisk
```

W powyższym przykładzie zastosowano kontener z biblioteki SWING - *JApplet*, i dlatego inna jest forma dodania tworzonego komponentu (*JButton*) do tego kontenera (*getContentPane().add*()). Różnica ta zostanie opisana w podrozdziale dotyczącym rozkładów.

5.2.2 Kontenery

Do tej pory przedstawione zostały dwa kontenery (komponenty zawierające inne komponenty): *Applet* oraz *JApplet*. Warto zapoznać się z pozostałymi.

Zasadniczym kontenerem jest w Javie jest okno reprezentowane przez klasę *Window*. Kontener taki nie posiada ani granic okna (ramy) ani możliwości dodania menu. Stanowi więc jakby pole robocze. Dlatego nie korzysta się z tego kontenera wprost, lecz poprzez klasy potomne: *Frame* oraz Dialog. Obiekt klasy *Frame* jest związany z kontenerem okna graficznego o sprecyzowanych cechach (ramie - ang. *Frame*). Kontener taki stanowi więc to, co jest odbierane w środowisku interfejsu graficznego przez okno graficzne. Stąd w Javie przyjęło się określać kontener klasy *Frame* mianem okna. Okno może posiadać pasek menu. Drugim kontenerem reprezentowanym przez klasę dziedziczącą z klasy *Window* - jest kontener typu *Dialog*. Kontener ten podobnie jak okno graficzne (*Frame*) posiada określone ramy, lecz różni się od tego okna tym, że nie można mu przypisać paska menu. Okno dialogowe (kontener reprezentowany przez klasę *Dialog*) posiada zawsze właściciela, jest zatem zależne. Inne, często wykorzystywane kontenery to Panel oraz *ScrollPane*. Panel służy do wyodrębnienia w kontenerze głównym kilku obszarów roboczych. Klasa Panel nie posiada więc prawie w ogóle żadnych metod (1) własnych. *ScrollPane* jest kontenerem umożliwiającym wykorzystanie pasków przesuwu poziomego i pionowego dla kontenera głównego. Oczywiście biblioteka SWING definiuje własne kontenery, których nazwa zaczyna się od J, i tak np. *JFrame, JWindow, JDialog, JPanel, JApplet, JFileDialog*, itp.
Znając charakterystykę kontenerów warto zapoznać się z możliwością ich wykorzystania. Zasada jest bardzo prosta - kontenery tworzy się dla:

1. aplikacji - zasadniczy kontener to *Frame* lub *JFrame*,
2. apletu - zasadniczy kontener to *Applet* lub *JApplet*.

Nie oznacza to jednak, że nie można używać okien w apletach i apletów w oknach!
Tworząc okno graficzne należy kierować się następującą procedurą:

1. deklaracja i zainicjowanie okna, np.: Frame okno = new Frame(„Program”);
2. ustawienie parametrów okna, np. okno.setSize(300,300);
3. dodanie komponentów do okna, np.: okno.add(new Button(„Ognia”));
4. ustawienie okna jako aktywnego, np.: okno.setVisible(true);

Brak ostatniego kroku spowoduje, że zdefiniowane okno będzie niewidoczne dla użytkownika. Rozpatrzmy następujący przykład:

Przykład 5.6:

```
//Dzialo.java:

import java.awt.*;

class Okno extends Frame{
        Okno(String nazwa){
                super(nazwa); //metoda super wywołuje konstruktor nadklasy
                setResizable(false);
                setSize(400,400);
        }
}// koniec class Okno

public class Dzialo{
```

```
	public static void main(String args[]){
		Okno o = new Okno("Panel sterujący działem");
		o.add(new Button("Ognia"));
		o.setVisible(true);
	}
}// koniec class Dzialo
```

Efektem działania kodu będzie stworzenie okna graficznego o niezmiennych rozmiarach 400/400 pikseli. Okno zawiera jeden przycisk pokrywający cały obszar roboczy. Ponieważ nie omówiono jeszcze metod obsługi zdarzeń nie możliwe jest zakończenie pracy okna przez wykorzystanie kontrolek okna. Dlatego trzeba przerwać pracę aplikacji w konsoli (CTRL-C). Okno (podobnie jak Aplet) może być podzielone na obszary poprzez wykorzystanie paneli. Co więcej panele mogą również być podzielone na inne panele. Powyższy kod można rozwinąć do postaci:

Przykład 5. 7:

```
//DzialoS.java:

import java.awt.*;

class Okno extends Frame{
	Okno(String nazwa){
		super(nazwa);
		setResizable(false);
		setSize(400,400);
	}
}// koniec class Okno

public class DzialoS{

	public static void main(String args[]){
		Okno o = new Okno("Panel sterujący działem");
		o.setLayout(new FlowLayout()); //zmiana rozkładu
		Panel p = new Panel();
		p.add(new Button("Ognia"));
		p.add(new Button("Stop"));
		o.add(p);
		Panel p1 = new Panel();
		p.add(new Label("Kontrola działa"));
		o.add(p1);
		o.setVisible(true);
	}
}// koniec class DzialoS
```

W kodzie powyższym wprowadzono dwa panele dzielące okno na dwie części. W panelu pierwszy umieszczono dwa przyciski, natomiast w panelu drugim jedną etykietę. Bezpośrednio po zainicjowaniu obiektu okna dokonano zmiany metody rozkładu elementów aby umożliwić wyświetlenie wszystkich tworzonych komponentów. Rozkłady są omówione dalej. Omawiany przykład wskazuje na ciekawy sposób konstrukcji interfejsu graficznego w Javie. Praktycznie interfejs składany jest z klocków (kontenery i komponenty) co znacznie upraszcza procedurę

tworzenia programów graficznych.
Z kontenerem typu *Frame* związany jest zbiór komponentów menu. W celu stworzenia menu okna graficznego należy wykonać następujące kroki:

1. zainicjować obiekt klasy *MenuBar* reprezentujący pasek menu;
2. zainicjować obiekt klasy Menu reprezentujący jedną kolumnę wyborów,
3. zainicjować obiekt klasy *MenuItem* reprezentujący element menu w danej kolumnie
4. dodać element menu do Menu;
5. powtórzyć kroki 2,3,4 tyle razy ile ma być pozycji w kolumnie i kolumn
6. dodać pasek menu do okna graficznego.

Realizację menu przedstawia następujący program:

Przykład 5. 8:

```
//DzialoM.java:

import java.awt.*;

class Okno extends Frame{
        Okno(String nazwa){
                super(nazwa);
                setResizable(false);
                setSize(400,400);
        }
}// koniec class Okno

public class DzialoM{

        public static void main(String args[]){
                Okno o = new Okno("Panel sterujący działem");
                MenuBar pasek = new MenuBar();

                Menu plik = new Menu("Plik");
                plik.add(new MenuItem("-")); //separator
                plik.add(new MenuItem("Ognia"));
                plik.add(new MenuItem("Stop"));
                pasek.add(plik);

                Menu edycja = new Menu("Edycja");
                edycja.add(new MenuItem("-"));
                edycja.add(new MenuItem("Pokaż dane działa"));
                pasek.add(edycja);

                o.setMenuBar(pasek);

                o.add(new Button());
                o.setVisible(true);

                Dialog d = new Dialog(o,"Dane działa", false);
                d.setSize(200,200);
                d.setVisible(true);
```

```
                d.show();
        }
}// koniec class DzialoM
```

Przykład ten demonstruje mechanizm tworzenia menu w oknie graficznym. Warto zauważyć, że dodanie paska menu do okna odbywa się nie poprzez metodę *add*() lecz przez *setMenuBar*(). Jest to istotne ze względu na ukazanie różnicy pomiędzy komponentem typu menu a innymi komponentami, które dziedziczą z klasy *Component* jak *Button, Label, Panel*. W omawianym przykładzie ukazano również metodę tworzenia okien dialogowych. Wywołany konstruktor powoduje utworzenie okna dialogowego właścicielem którego będzie okno graficzne „o"; nazwą okna dialogowego będzie „Dane działa" oraz okno to nie będzie blokować operacji wejścia.

W pakiecie SWING również zdefiniowano klasy komponentów związane z menu. Jak w większości przypadków, główna różnica pomiędzy analogicznymi komponentami AWT i SWING polega na tym, że te ostatnie umożliwiają implementację elementów graficznych (ikon). Możliwe jest więc na przykład skonstruowanie menu, w którym każdy element jest reprezentowany tylko przez ikonę graficzną.

5.2.3 Rozkłady

Bardzo częstym dylematem programisty jest problem rozkładu komponentów w ramach tworzonego interfejsu graficznego. Problem rozdzielczości ekranu, liczenia pikseli to typowe zadania do rozwiązania przed przystąpieniem do projektu interfejsu. W języku JAVA problemy te w znaczny sposób zredukowano wprowadzając tzw. rozkłady. Rozkład oznacza nic innego jak sposób układania komponentów na danej formie (np. aplet , panel). System zarządzający rozkładami umieszcza dany komponent (jako rezultat działania metody *add*()) zgodnie z przyjętym rozkładem. Definiowane w AWT rozkłady to:

BORDER LAYOUT - (domyślny dla kontenerów*: Window, Frame, Dialog, JWindow, JFrame, JDialog*) komponenty są umieszczane i dopasowywane do pięciu regionów: północ, południe, wschód, zachód oraz centrum. Każdy z pięciu regionów jest identyfikowany przez stałą z zakresu: NORTH, SOUTH, EAST, WEST, oraz CENTER.

CARD LAYOUT - każdy komponent w danej formie (kontenerze) jest rozumiany jako karta. W danym czasie widoczna jest tylko jedna karta, a forma jest rozumiana jako stos kart. Metody omawianej klasy umożliwiają zarządzanie przekładaniem tych kart.

FLOW LAYOUT - (domyślny dla kontenerów*: Panel, Applet, JPanel, JApplet*) komponenty są umieszczane w ciągu "przepływu" od lewej do prawej (podobnie do kolejności pojawiania się liter przy pisaniu na klawiaturze).

GRID LAYOUT - komponenty są umieszczane w elementach regularnej siatki (gridu). Forma (kontener) jest dzielona na równe pola, w których kolejno umieszczane są komponenty.

GRIDBAG LAYOUT - komponenty umieszczane są w dynamicznie tworzonej siatce regularnych pól, przy czym komponent może zajmować więcej niż jedno pole.

Przykład zastosowania rozkładu BORDER LAYOUT ukazano poniżej:

Przykład 5.9 :

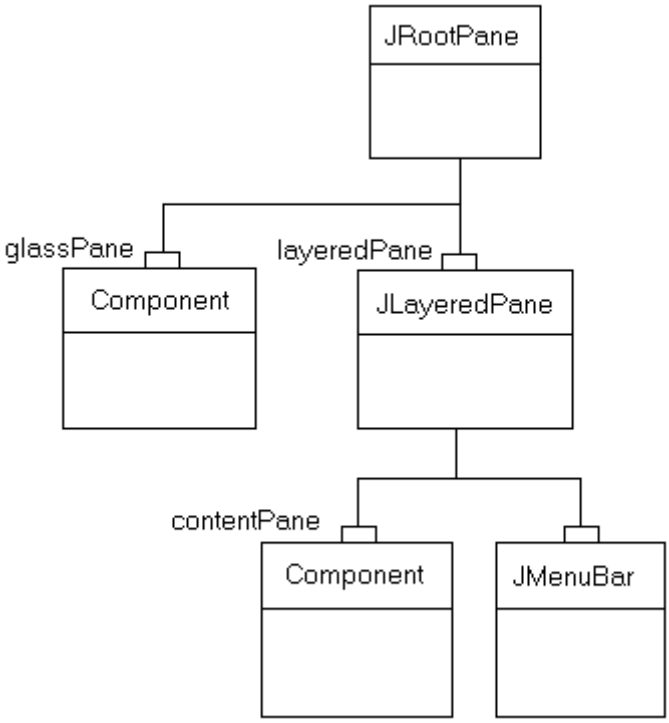
```
//ButtonTest4.java

import java.awt.*;
import java.applet.Applet;

public class ButtonTest4 extends Applet {
      public void init() {
        setLayout(new BorderLayout());
        add(new Button("North"), BorderLayout.NORTH);
        add(new Button("South"), BorderLayout.SOUTH);
        add(new Button("East"), BorderLayout.EAST);
        add(new Button("West"), BorderLayout.WEST);
        add(new Button("Center"), BorderLayout.CENTER);
      }
} // koniec public class ButtonTest4
```

Pakiet Swing wprowadzający nowe kontenery i komponenty ustala również nowe metody używania rozkładów oraz wprowadza nowe rozkłady. Kontener (forma) zawiera podstawowe komponenty zwane PANE (płyty).

Podstawowym komponentem jest tu płyta główna (korzeń) - *JRootPane*. Płyta ta jest fundamentalną częścią kontenerów w Swing takich jak *JFrame*, *JDialog*, *JWindow*, *JApplet*. Oznacza to, że umieszczenie komponentów nie odbywa się bezpośrednio przez odwołanie do tych kontenerów lecz poprzez *JRootPane*. Dla potrzeb konstrukcji GUI z wykorzystaniem elementów pakietu Swing istotna jest znajomość struktury *JRootPane*, pokazanej poniżej.



Rysunek 5.2 Konstrukcja płyty rootPane

JRootPane jest tworzona przez *glassPane* (szyba, płyta szklana) i *layeredPane* (warstwę płyt), na którą składają się: opcjonalna *menuBar* (pasek menu) oraz *contentPane* (płyta robocza, zawartość). Płyta *glassPane* jest umieszczona zawsze na wierzchu wszystkich elementów (stąd szyba) i jest związana z poruszaniem myszy. Ponieważ *glassPane* może stanowić dowolny komponent możliwe jest więc rysowanie w tym komponencie. Domyślnie *glassPane* jest niewidoczna. Płyta *contentPane* stanowi główną roboczą część kontenera gdzie rozmieszczamy komponenty i musi być zawsze nadrzędna względem każdego "dziecka" płyty *JRootPane*. Oznacza to, że zamiast bezpośredniego dodania komponentu do kontenera musimy dodać element do płyty *contentPane*:

```
- zamiast dodania:
        rootPane.add(child);
    czy jak to było w AWT np.:
        Frame frame;
        frame.add(child);

- wykonujemy:
        rootPane.getContentPane().add(child);
     np.:
        JFrame frame;
        frame.getContentPane().add(child);
```

Jest to niezwykle ważna różnica pomiędzy klasycznym dodawaniem elementów do kontenerów w AWT a w SWING. Oznacza to, że również ustawienia

rozkładów muszą odwoływać się do płyt zamiast bezpośrednio do kontenerów np. frame.getContentPane().setLayout(new GridLayout(3,3)).

Dodatkowo w pakiecie Swing wprowadzono inne rozkłady takie jak:

- BOX LAYOUT,
- OVERLAY LAYOUT,
- SCROLLPANE LAYOUT,
- VIEWPORT LAYOUT.

Przykładowo można porównać działanie rozkładu *BorderLayout* dla implementacji appletu w AWT i SWING:

Przykład 5.10 :

```
// AWT:
// ButtonTest4.java

import java.awt.*;
import java.applet.Applet;

public class ButtonTest4 extends Applet {
      public void init() {
        setLayout(new BorderLayout());
        add(new Button("North"), BorderLayout.NORTH);
        add(new Button("South"), BorderLayout.SOUTH);
        add(new Button("East"), BorderLayout.EAST);
        add(new Button("West"), BorderLayout.WEST);
        add(new Button("Center"), BorderLayout.CENTER);
      }
} // koniec public class ButtonTest4
```

Przykład 5.11 :

```
// SWING:
// ButtonTest5.java:

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

public class ButtonTest5 extends JApplet {
      public void init() {
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(new Button("North"), BorderLayout.NORTH);
        getContentPane().add(new Button("South"), BorderLayout.SOUTH);
        getContentPane().add(new Button("East"), BorderLayout.EAST);
        getContentPane().add(new Button("West"), BorderLayout.WEST);
        getContentPane().add(new Button("Center"), BorderLayout.CENTER);
      }
} // koniec public class ButtonTest5
```

W celu zrozumienia różnicy warto skompilować i uruchomić aplet buttonTest5 z usunięciem części " *getContentPane*()" z *linii getContentPane().setLayout(new BorderLayout())*; w kodzie programu.

5.2.4 Zdarzenia

Oprócz wyświetlenia komponentu konieczne jest stworzenie odpowiedniego sterowania związanego z danym komponentem. W przykładowym kodzie apletu ButtonTest.java ukazano implementację metody action():

Przykład 5.12:

```
//ButtonTest.java:

import java.awt.*;
import java.applet.*;

public class ButtonTest extends Applet {

                public void init() {
                   Button przyciskTest = new Button("przycisnij mnie...");
                   add(przyciskTest);
                }

                public boolean action(Event e, Object test) {
                   if(test.equals("przycisnij mnie..."))
                     System.out.println("Przycisnieto " + test);
                  else
                     return false;
                  return true;
                }
}// koniec public class ButtonTest

//---------------------------------------------------

ButtonTest.html :
...
<APPLET  CODE="ButtonTest.class" WIDTH=70 HEIGHT=40 ALIGN=CENTER></APPLET>
```

Uwaga! Kompilacja tego kodu w środowisku JDK późniejszym niż 1.0 zgłosi ostrzeżenie o użyciu metody (*action()*) o obniżonej wartości (*deprecated*). Nie powoduje to jednak w tym przypadku przerwania procesu kompilacji.

Metoda ta obsługuje zdarzenia związane z obiektem typu Button, i w rozpatrywanym przykładzie wyświetla tekst na standardowym urządzeniu wyjścia. Warto zwrócić uwagę na instrukcję warunkową if(test.equals()). Otóż metoda action może obsługiwać kilka komponentów i w zależności od obiektu test wykonywać to działanie, które programista umieścił w odpowiedniej części warunkowej. W przypadku obsługi wielu różnych komponentów bardzo często pojawiają się błędy wynikające z nazewnictwa etykiety komponentu (przestawione litery, wielkie litery, itp.). Ogólnie biorąc każdy komponent może wywołać określone zdarzenie (Event),

którego obsługa umożliwia interaktywną pracę z interfejsem programu. W tradycyjnym modelu GUI program generuje interfejs graficzny, a następnie w nieskończonej pętli czeka na pojawienie się zdarzeń, które należy w celowy sposób obsłużyć. W Javie 1.0 obsługę zdarzeń wykonuje się poprzez odpowiednie instrukcje warunkowe określające typ zdarzenia w ciele metod: action() oraz handleEvent() (od JDK 1.1 metody te są wycofywane – „deprecated"). Takie rozwiązanie jest mało efektywne, a na pewno nie jest obiektowe. Od wersji Javy 1.1 wprowadza się nowy system przekazywania i obsługi zdarzeń określający obiekty jako nasłuchujące zdarzeń (listeners) i jako generujące zdarzenia (sources). W tym nowym modelu obsługi zdarzeń komponent "odpala" ("fire") zdarzenie. Każdy typ zdarzenia jest reprezentowany przez określoną klasę. W nowym systemie można nawet stworzyć własne typy zdarzeń. Wygenerowane przez komponent zdarzenie jest odbierane przez właściwy element nasłuchu związany z komponentem generującym zdarzenie. Jeżeli w ciele klasy nasłuchu danych zdarzeń istnieje metoda obsługi tego zdarzenia, to zostanie ona wykonana. Generalnie więc można rozdzielić źródło zdarzeń i obsługę zdarzeń. Najczęstszym modelem programu wykorzystywanym do tworzenia obsługi zdarzeń jest implementacja obsługi zdarzeń jako klasa wewnętrzna (inner class) klasy komponentu, którego zdarzenia mają
być obsługiwane. Przykładowo:

Przykład 5.13:

```
//ButtonTest2.java:

import java.awt.*;
import java.awt.event.*; // Koniecznie należy pamiętać o importowaniu pakietu event
import java.applet.*;

public class ButtonTest2 extends Applet {
  Button b1, b2;
  public void init() {
    b1 = new Button("Przycisk 1");
    b2 = new Button("Przycisk 2");
    b1.addActionListener(new B1());  //dodajemy obiekt nasłuchujący zdarzenia dla komponentu b1
    b2.addActionListener(new B2());  //dodajemy obiekt nasłuchujący zdarzenia dla komponentu b1
    add(b1);
    add(b2);
  } // koniec public void init()

  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      getAppletContext().showStatus("Przycisk 1");
    }
  } // koniec class B1

  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      getAppletContext().showStatus("Przycisk 2");
    }
  } // koniec class B2
  /* Dla porównania stary model obsługi zdarzeń:
  public boolean action(Event e, Object test) {
    if(e.target.equals(b1))  //lub test.equals("Przycisk 1"); gorsze rozwiazanie
```

```
    getAppletContext().showStatus("Przycisk 1");
  else if(e.target.equals(b2))
          getAppletContext().showStatus("Przycisk 2");
        else
          return super.action(e, test);
  return true;
}
 */

} // koniec public class ButtonTest2 extends Applet

//--------------------------------------------------
```

ButtonTest2.html :

```
...
<APPLET  CODE="ButtonTest2.class" WIDTH=70 HEIGHT=40 ALIGN=CENTER></APPLET>
```

Jak widać na prezentowanym przykładzie element nasłuchu zdarzeń jest obiektem klasy implementującej interfejs nasłuchu. To, co musi zrobić programista to stworzyć odpowiedni obiekt nasłuchu dla komponentu odpalającego zdarzenie. Przesłanie obiektu nasłuchu polega na wykonaniu metody addXXXXXListener() dla danego komponentu, gdzie XXXXX oznacza typ zdarzenia jakie ma być nasłuchiwane (np. addMouseListener, addActionListener, addFocusListener, ...).
Sposób rejestracji i nazwa metody obsługującej tę rejestrację daje programiście informację jaki typ zdarzeń jest obsługiwany w danym fragmencie kodu.

Często rejestracja obiektu obsługi zdarzeń zawiera definicję klasy i metody obsługi zdarzenia. W przeciwieństwie do wyżej omawianej metody definiowane klasy są anonimowe. To alternatywne rozwiązanie obsługi zdarzeń jest często wykorzystywane szczególnie tam, gdzie kod obsługi zdarzenia jest krótki. Najprostszym przykładem może być stworzenie ramki - FRAME (okna), która będzie wyświetlana na ekranie i prostego kodu obsługującego zdarzenie zamknięcia ramki (np. metodą System.exit(0);). Standardowo wykorzystuje się następujący kod:

```
…
addWindowListener(new WindowAdapter(){                    //dodajemy obsługę
zdarzeń okna

  public void windowClosing(WindowEvent e){
  //implementacja metody zamykania okna przy wystąpieniu danego zdarzenia
    System.out.println("Dziekujemy za prace z programem...");
 //akcja podejmowana przy zamykaniu okna
     System.exit(0);                                      //koniec programu
  }

});
…
```

Możemy więc teraz stworzyć przykładowy program (może być wywołany zarówno jako aplet jak i aplikacja) demonstrujący omówione zagadnienia:

Przykład 5.14:

```
//ButtonTest3.java:

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

public class ButtonTest3 extends Applet {
  Button b1, b2;
  TextField t = new TextField(20);

  public void init() {
    b1 = new Button("Przycisk 1");
    b2 = new Button("Przycisk 2");
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    add(b1);
    add(b2);
    add(t);
  }

  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Przycisk 1");
    }
  }// koniec class B1

  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Przycisk 2");
    }
  }// koniec class B2

  // statyczna metoda  main() wymagana dla aplikacji:

  public static void main(String args[]) {
    ButtonTest3 applet = new ButtonTest3();
    Frame okno = new Frame("ButtonTest3");
    okno.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e){
                        System.out.println("Dziekujemy za prace z programem...");
                        System.exit(0);
         }
   });
    okno.add(applet, BorderLayout.CENTER);
    okno.setSize(300,200);
    applet.init();
    applet.start();
    okno.setVisible(true);
  }
} // koniec public class ButtonTest3 extends Applet
```

W celu właściwej obsługi zdarzeń konieczna jest wiedza dotycząca typu zdarzeń . Związane jest z tym poznanie podstawowych klas zdarzeń. W wersji JDK 1.0 podstawową klasą przechowującą informacje związaną z danym zdarzeniem była klasa java.awt.Event. W celu kompatybilności z JDK1.0 klasa ta występuje również w późniejszych wersjach JDK. Począwszy od JDK 1.1 podstawową klasą zdarzeń jest klasa java.awt.AWTEvent będąca korzeniem wszystkich klas zdarzeń w AWT. Klasa ta dziedziczy z klasy java.util.EventObject będącą korzeniem wszystkich klas zdarzeń w Javie (nie tylko AWT ale np. dla SWING, itp.). Klasy zdarzeń dla AWT są zdefiniowane w pakiecie java.awt.event.* i dlatego w celu obsługi zdarzeń konieczne jest importowanie tego pakietu:
(java.awt.event.*):

AdjustmentEvent
ContainerEvent,
FocusEvent,
KeyEvent,
MouseEvent,
ComponentEvent
Text Event
WindowEvent
ItemEvent
ActionEvent
InputMethodEvent
InvocationEvent
PaintEvent

Dodatkowe zdarzenia zdefiniowane dla biblioteki SWING to:
(javax.swing.event.*):

AncestorEvent
CaretEvent
ChangeEvent
HyperlinkEvent
InternalFrameEvent
ListDataEvent
ListSelectionEvent
MenuDragMouseEvent
MenuEvent
MenuKeyEvent
PopupMenuEvent
TableColumnModelEvent
TableModelEvent
TreeExpansionEvent
TreeModelEvent
TreeSelectionEvent
UndoableEditEvent

Zdarzenia są powiązane z komponentami i tak:

Komponent	Zdarzenie
Adjustable	AdjustmentEvent
Applet	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Button	ActionEvent, FocusEvent, KeyEvent,

Komponent	Zdarzenie
	MouseEvent, ComponentEvent
Canvas	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Checkbox	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
CheckboxMenuItem	ActionEvent, ItemEvent
Choice	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Component	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Container	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Dialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
FileDialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Frame	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Label	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
List	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ItemEvent, ComponentEvent
Menu	ActionEvent
MenuItem	ActionEvent
Panel	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
PopupMenu	ActionEvent
Scrollbar	AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
ScrollPane	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextArea	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextComponent	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextField	ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Window	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

Każda klasa zdarzeń definiuje zbiór pól przechowujących stan zdarzenia. Stan zdarzenia jest istotny ze względu na obsługę zdarzenia, a więc na wykonanie odpowiedniego działania związanego z wartościami pól obiektu zdarzenia. Referencja do obiektu odpowiedniego zdarzenia jest podawana jako argument przy wykonywaniu metody obsługującej to zdarzenie. Metody związane ze zdarzeniami są pogrupowane jako zbiory abstrakcyjnych metod w interfejsach lub jako zbiory pustych metod w klasach (adaptery). Zadaniem programisty jest definicja metod wykorzystywanych do obsługi zdarzeń. Następujące interfejsy i adaptery są zdefiniowane w Javie:

Interfejs lub adapter	Metody
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener	componentHidden(ComponentEvent)
ComponentAdapter	componentShown(ComponentEvent)
	componentMoved(ComponentEvent)
	componentResized(ComponentEvent)
ContainerListener	componentAdded(ContainerEvent)
ContainerAdapter	componentRemoved(ContainerEvent)
FocusListener	focusGained(FocusEvent)
FocusAdapter	focusLost(FocusEvent)
KeyListener	keyPressed(KeyEvent)
KeyAdapter	keyReleased(KeyEvent)
	keyTyped(KeyEvent)
MouseListener	mouseClicked(MouseEvent)
MouseAdapter	mouseEntered(MouseEvent)
	mouseExited(MouseEvent)
	mousePressed(MouseEvent)
	mouseReleased(MouseEvent)
MouseMotionListener	mouseDragged(MouseEvent)
MouseMotionAdapter	mouseMoved(MouseEvent)
WindowListener	windowOpened(WindowEvent)
WindowAdapter	windowClosing(WindowEvent)
	windowClosed(WindowEvent)
	windowActivated(WindowEvent)
	windowDeactivated(WindowEvent)
	windowIconified(WindowEvent)
	windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)
TextListener	textValueChanged(TextEvent)

Wykonując metodę obsługującą zdarzenie można wykorzystać pola obiektu danego zdarzenia. Możliwość ta pozwala określić okoliczności wystąpienia zdarzenia np.: współrzędne kursora podczas generacji zdarzenia, typ klawisza klawiatury jaki był wciśnięty, nazwę komponentu źródłowego dla danego zdarzenia, stan lewego przycisku myszki podczas zdarzenia, itp. Analiza pól obiektu jest zatem niezwykle istotna z punktu widzenia działania tworzonego interfejsu graficznego.

Poniższy program ukazuje obsługę zdarzeń związanych z klawiaturą i myszką:

Przykład 5.15:

```
//Komunikator.java:


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

class Ekran extends Canvas{

        public String s="Witam";
        private Font f;

        Ekran (){
                super();
```

```
                f = new Font("Times New Roman",Font.BOLD,20);
                setFont(f);
                addKeyListener(new KeyAdapter(){
                        public void keyPressed(KeyEvent ke){
                                s=new String(ke.paramString());
                                repaint();
                        }
                });
                addMouseListener(new MouseAdapter(){
                        public void mousePressed(MouseEvent me){
                                s=new String(me.paramString());
                                repaint();
                        }
                });

        }

        public void paint(Graphics g){
                g.drawString(s,20,220);
        }


}// koniec class Ekran

public class Komunikator extends Frame  {


        Komunikator (String nazwa){
                super(nazwa);
        }

        public static void main(String args[]){
                Komunikator okno = new Komunikator("Komunikator");
                okno.setSize(600,500);
                Ekran e = new Ekran();
                okno.add(e);
                okno.addWindowListener(new WindowAdapter(){
                         public void windowClosing(WindowEvent e){
                                System.out.println("Dziekujemy za prace z programem...");
                                System.exit(0);
                        }
                 });
                okno.setVisible(true);


        }
}// koniec public class Komunikator
```

W przykładzie tym zastosowano komponent Canvas stanowiący pole graficzne, w obrębie którego można rysować używając metod zdefiniowanych w klasie Graphics. Z polem graficznym związano obsługę dwóch typów zdarzeń: wciśnięcie klawisza klawiatury (keyPressed()) oraz wciśnięcie klawisza myszki (mousePressed()). Zdarzenia te są opisane w obiektach klas KeyEvent oraz MouseEvent. Każdy obiekt przechowuje informacje związane z powstałym zdarzeniem i dlatego można te pola

odczytać celem wyświetlenia ich wartości. W powyższym przykładzie zasotosowano metodę paramString() zwracającą wartości pól obiektu w postaci łańcucha znaków (String). W przypadku wystąpienia zdarzenia i jego obsługi za pomocą omawianych metod generowana jest metoda repaint() powodująca odświeżenie pola graficznego. Warto zauważyć, że w definiowaniu klas anonimowych obsługujących zdarzenia zastosowano adaptery. Jak wspomniano wcześniej jest to na tyle wygodne w przeciwieństwie od implementacji interfejsów, że nie trzeba definiować wszystkich metod w nich zawartych (zadeklarowanych).