

Klasy abstrakcyjne

Klasa abstrakcyjna jest to klasa której obiekty nie mogą być tworzone, może być natomiast dziedziczona. Może posiadać konstruktor, może on być jednak wywołany tylko przez klasy pochodne.

Klasę abstrakcyjną tworzymy przy pomocy modyfikatora `abstract`

Dodatkowo klasa abstrakcyjna może posiadać metody abstrakcyjne, metoda taka posiada listę argumentów, jednak nie posiada ciała.

Metody abstrakcyjne muszą zostać nadpisane w klasach pochodnych. (również poprzedzone są modyfikatorem `abstract`)

```
public abstract class Zwierze {  
    protected String imie;  
  
    public Zwierze (String nazwij){  
        imie = nazwij;  
    }  
  
    String podajImie(){  
        return imie;  
    }  
    abstract String wydajGlos();  
}
```

```
public class Kot extends Zwierze {  
    public Kot(String nazwij) {  
        super(nazwij);  
    }  
    @Override  
    String wydajGlos() {  
        return "Miau";  
    }  
}
```

Interfejsy

Interfejs w Javie to deklarowany za pomocą słowa kluczowego **interface** nazwany zbiór deklaracji zawierający:

- publiczne abstrakcyjne metody (bez implementacji),
- publiczne statyczne zmienne finalne (stałe) o ustalonych typach i wartościach.

Implementacja interfejsu w klasie polega na zdefiniowaniu w tej klasie wszystkich metod zadeklarowanych w implementowanym interfejsie.

Interfejsy

Ogólna postać definicji interfejsu w języku Java:

```
public interface NazwaInterfejsu
{
    typ nazwaZmiennej = wartosc;
    ...
    typ nazwaMetody(lista_parametrów);
    ...
}
```

Uwagi:

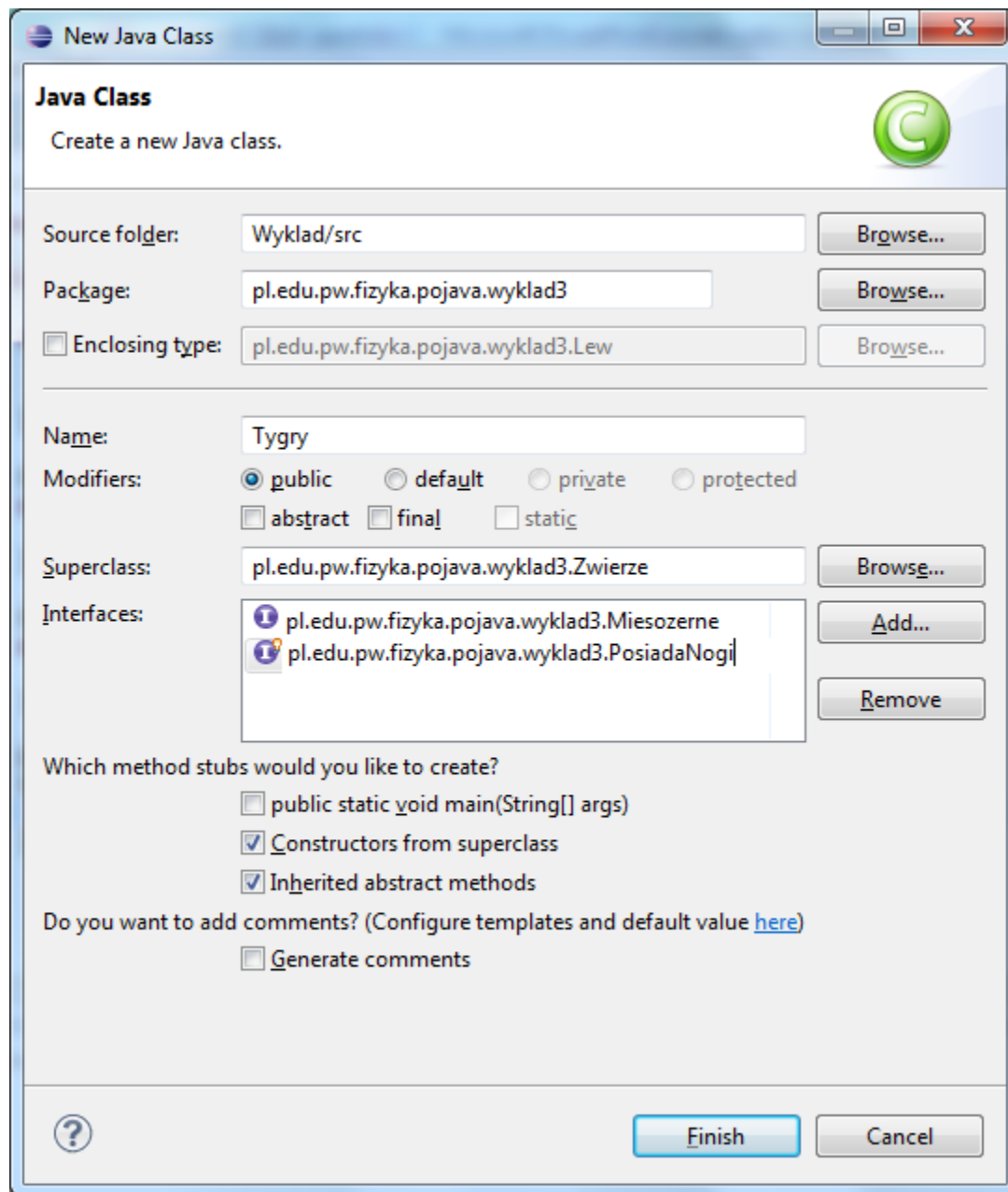
- modyfikator dostępu **public** przed słowem **interface** może nie występować (wówczas interfejs jest dostępny tylko w bieżącym pakiecie,
- ewentualne zmienne są zawsze typu **static final** i mają przypisaną wartość stałą,
- metody są zawsze abstrakcyjne (bez implementacji).

```
public interface Miesozerne {  
  
    String zjedzMieso();  
  
}
```

```
public interface PosiadaNogi {  
  
    int podajIloscNog();  
  
}
```

```
public class Lew extends Zwierze implements  
Miesozerne, PosiadaNogi {  
    public Lew(String nazwij) {  
        super(nazwij);  
    }  
    @Override  
    public int podajIloscNog() {  
        return 4;  
    }  
    @Override  
    public String zjedzMieso() {  
        return "Lew zjadł mięso";  
    }  
    @Override  
    String wydajGlos() {  
        return "Lew ryczy";  
    }  
}
```

Szybkie dodawanie interfejsów w Eclipse



The screenshot shows the 'New Java Class' dialog box in the Eclipse IDE. The dialog is titled 'New Java Class' and has a green 'C' icon in the top right corner. The main text says 'Create a new Java class.' Below this, there are several input fields and buttons:

- Source folder:** A text field containing 'Wyklad/src' and a 'Browse...' button.
- Package:** A text field containing 'pl.edu.pw.fizyka.pojava.wyklad3' and a 'Browse...' button.
- Enclosing type:** A checkbox labeled 'Enclosing type:' followed by a text field containing 'pl.edu.pw.fizyka.pojava.wyklad3.Lew' and a 'Browse...' button.
- Name:** A text field containing 'Tygry'.
- Modifiers:** A group of radio buttons for 'public' (selected), 'default', 'private', and 'protected'. Below them are checkboxes for 'abstract', 'final', and 'static'.
- Superclass:** A text field containing 'pl.edu.pw.fizyka.pojava.wyklad3.Zwierze' and a 'Browse...' button.
- Interfaces:** A list box containing two entries: 'pl.edu.pw.fizyka.pojava.wyklad3.Miesozerne' and 'pl.edu.pw.fizyka.pojava.wyklad3.PosiadaNogi'. To the right of the list box are 'Add...' and 'Remove' buttons.
- Which method stubs would you like to create?:** A group of checkboxes: 'public static void main(String[] args)' (unchecked), 'Constructors from superclass' (checked), and 'Inherited abstract methods' (checked).
- Do you want to add comments?:** A checkbox labeled 'Generate comments' (unchecked). A link 'here' is provided for configuring templates and default values.

At the bottom of the dialog, there is a question mark icon on the left and two buttons: 'Finish' and 'Cancel'.

Wygenerowany automatycznie kod:

```
package pl.edu.pw.fizyka.pojojava.wyklad3;
public class Tygrys extends Zwierze implements Miesozerne, PosiadaNogi {
    public Tygrys(String nazwij) {
        super(nazwij);
        // TODO Auto-generated constructor stub
    }
    @Override
    public int podajIloscNog() {
        // TODO Auto-generated method stub
        return 0;
    }
    @Override
    public String zjedzMieso() {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    String wydajGlos() {
        // TODO Auto-generated method stub
        return null;
    }
}
```

Implementacja interfejsu

Ogólna postać definicji klasy implementującej interfejs::

```
[public] class NazwaKlasy extends KlasaBazowa  
    implements NazwaInterfejsu_1, ...,  
    NazwaInterjejsu_n  
{  
    ...  
}
```

- klasa może implementować wiele interfejsów,
- klasa może dziedziczyć tylko po jednej klasie bazowej
- klasa musi definiować WSZYSTKIE metody implementowanych interfejsów (albo pozostać klasa abstrakcyjną)
- klasa może zawierać własne (nie będące częścią interfejsu) atrybuty i metody.

Różnice między klasami abstrakcyjnymi a interfejsami

- W interfejsach wszystkie metody są abstrakcyjne, natomiast w klasie abstrakcyjnej można stworzyć metody posiadające ciało, jak i abstrakcyjne.
- W języku java można dziedziczyć tylko po jednej klasie, natomiast interfejsów, można implementować wiele. Ponadto interfejsy mogą dziedziczyć wiele interfejsów
- Klasa abstrakcyjna zazwyczaj jest mocno związana z klasami dziedziczącymi w sensie logicznym, interfejs natomiast nie musi być już tak mocno związany z daną klasą (określa jej cechy).

Wprowadzenie do SWING

JFC (Java Foundation Classes) – zbiór klas do budowy interfejsu graficznego użytkownika i interaktywności aplikacji Javy

- Komponenty Swing – etykiety, przyciski, listy itd.
- Przełączany wygląd Look-and-Feel – dostosowanie GUI do platformy systemowej, wiele różnych wygląków,
- Java 2D – biblioteka do tworzenia grafiki

...

Tworzenie prostego okna

```
public class ProsteOkno extends JFrame {  
    private static final long serialVersionUID =  
2639319250522477417L;  
    public ProsteOkno() throws HeadlessException {  
        super();  
        setSize(400,400);  
  
    }  
    public static void main(String[] args) {  
        JFrame f = new ProsteOkno();  
        f.setVisible(true);  
    }  
}
```

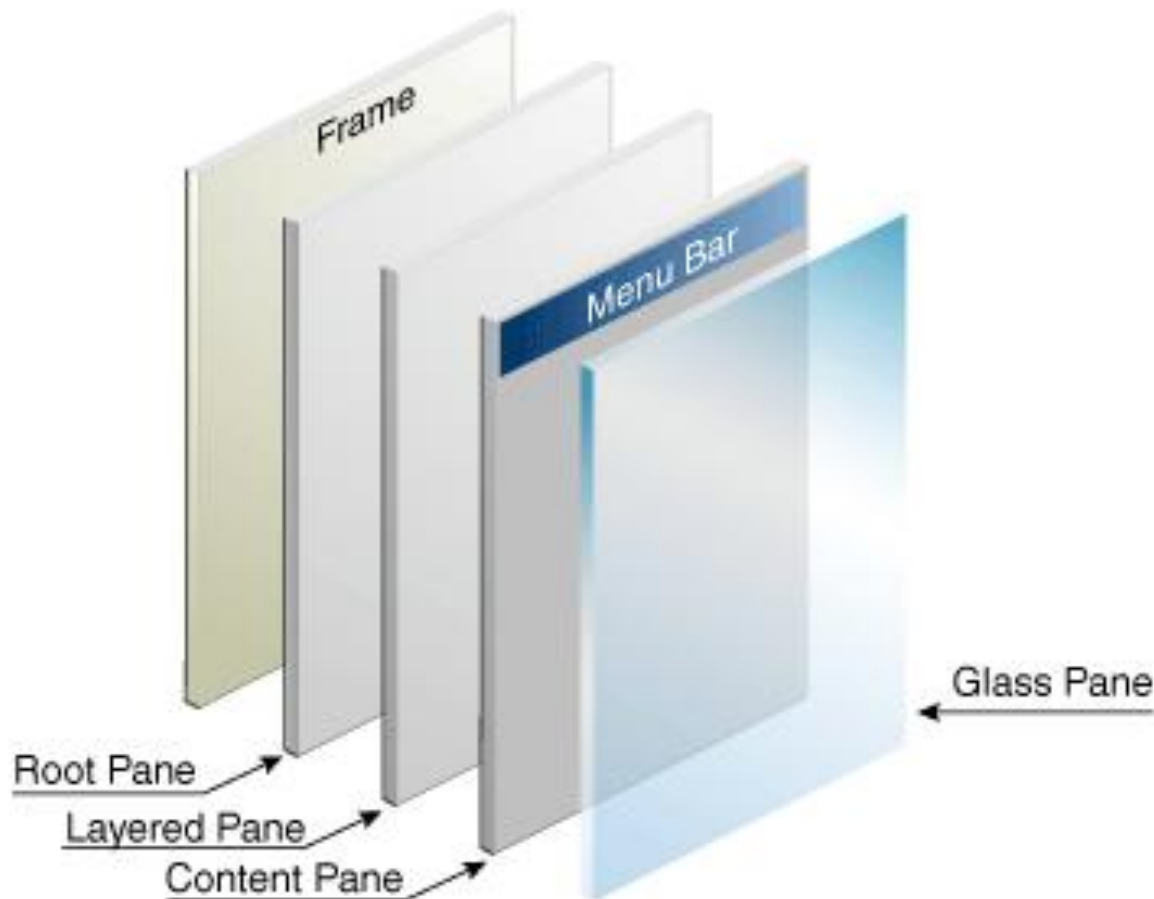
- Dla tak utworzonego okna zamknięcie [X] nie powoduje zamknięcia aplikacji. Aby to zmienić, należy wywołać metodę `setDefaultCloseOperation()`

setDefaultCloseOperation()

- EXIT_ON_CLOSE – zamyka aplikacje po zamknięciu ramki
- DISPOSE_ON_CLOSE – zamyka ramke, usuwa obiekt reprezentujący ramke, ale pozostawia pracującą aplikację,
- DO_NOTHING_ON_CLOSE – pozostawia ramkę otwartą i kontynuuje pracę aplikacji
- HIDE_ON_CLOSE – zamyka ramkę pozostawiając pracującą aplikację (domyślne)

Kontenery

Wszystkie komponenty Swing muszą być częścią drzewa, którego korzeniem jest JFrame (ew. JInternalFrame, JDialog lub JApplet)



Dodawanie komponentu

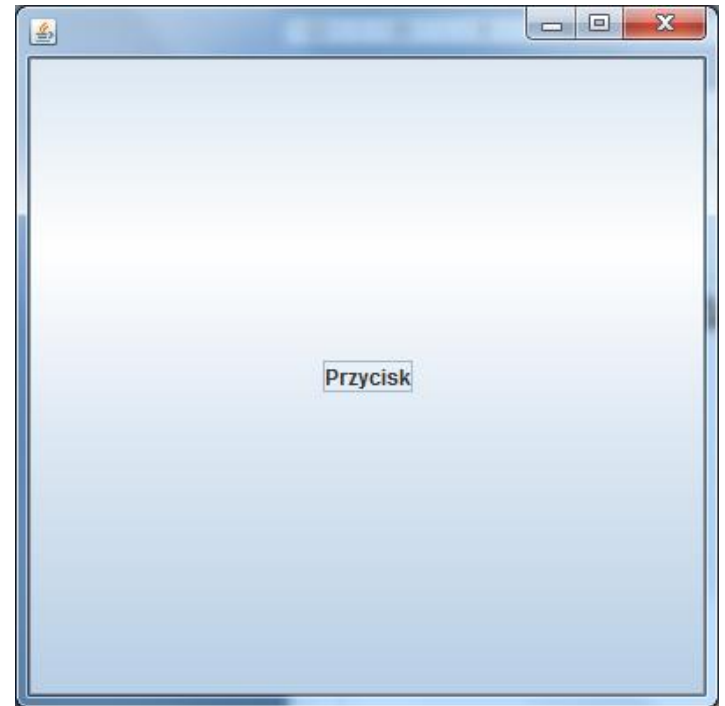
- utworzenie komponentu (obiektu pewnej klasy) oraz ustawienie jego właściwości
- określenie kontenera pośredniego – panelu z zawartością (ang. *content pane*) - *niekonieczne*
- wywołanie metody *add()* kontenera lub kontenera pośredniego

add(komponent)

Kontener (np. JPanel) to także rodzaj komponentu, i może być dodany jako element innego kontenera.

Okno z przyciskiem

```
public class ProsteOkno2 extends JFrame {  
    public ProsteOkno2() throws HeadlessException {  
        super();  
        setSize(400,400);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
  
        JButton b = new JButton("Przycisk");  
        add(b);  
    }  
    public static void main(String[] args) {  
        JFrame f = new ProsteOkno2();  
        f.setVisible(true);  
    }  
}
```



Rozmieszczanie komponentów

- Sposób rozmieszczenia komponentów zależy od zarządcy układu (Layout Manager)
- Zastosowanie zarządcy układu w kontenerze – metoda *setLayout()*

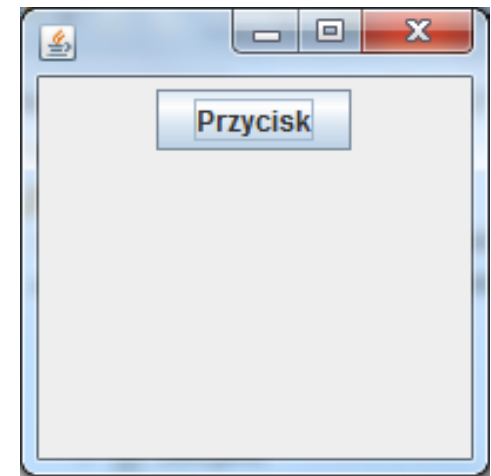
Predefiniowani zarządcy układu

- ciągły – *FlowLayout*
- siatkowy - *GridLayout*
- brzegowy – *BorderLayout*
- kartowy – *CardLayout*
- torebkowy - *GridBagLayout*
- pudełkowy – *BoxLayout*
- wiosenny – *SpringLayout*
- grupowy - *GroupLayout*


```

public ProsteOkno2() throws HeadlessException {
    super();
    setSize(200,200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new FlowLayout());
    JButton b = new JButton("Przycisk");
    add(b);
}

```

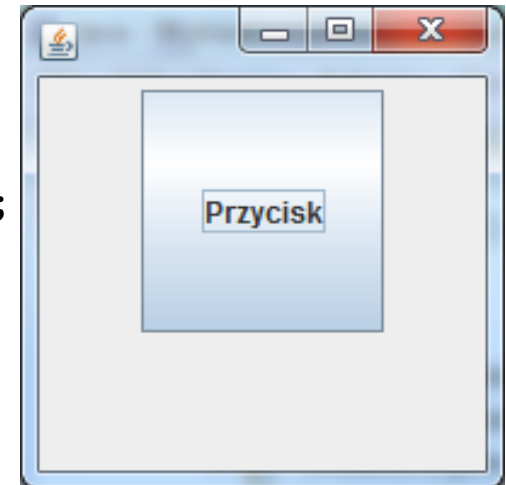


FlowLayout nadaje komponentom najmniejsze możliwe rozmiary

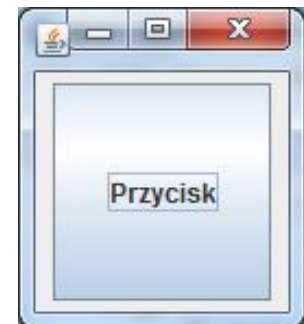
```

public ProsteOkno2() throws HeadlessException {
    super();
    setSize(200,200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new FlowLayout());
    JButton b = new JButton("Przycisk");
    b.setPreferredSize(new
Dimension(100,100));
    add(b);
}

```



Metoda `setPreferredSize()` pozwala w pewnym stopniu wpływać na rozmiar komponentów przy rozmieszczaniu;
Metoda `pack()` powoduje minimalizację rozmiarów komponentów i okna (z zachowaniem preferowanych rozmiarów)



Dodawanie obsługi zdarzeń do komponentu

Programowanie GUI jest programowaniem zdarzeniowym. Zdarzenia są generowane np. w momencie naciśnięcia klawisza lub kliknięcia myszą. W celu obsługi zdarzenia wykorzystywane są obiekty-Słuchacze (ang. Listeners). Aby móc generować obiekty-Słuchacze klasa musi implementować interfejs nasłuchu, który zawiera abstrakcyjne metody do obsługi zdarzeń.

Java dostarcza bogaty zestaw interfejsów nasłuchujących, metody każdego z tych interfejsów umożliwiają reakcję na zdarzenie określonego typu. Klasy słuchacze mogą implementować jeden lub kilka z tych interfejsów, zyskując w ten sposób zdolność do obsługi wybranych zestawów zdarzeń.

Dodawanie obsługi zdarzeń do komponentu

Do obsługi zdarzeń komponentu konieczna jest rejestracja dla danego komponentu obiektu klasy nasłuchującej:

```
źródłoZdarzeń.addRodzajListener (obiektKlasyNasłuchującej) ;
```

```
źródłoZdarzeń.addRodzajListener (obiekt  
KlasyNasłuchującej) ;
```

oznacza, że dla obsługi zdarzeń generowanych przez komponent **źródłoZdarzeń**, zarejestrowano obiekt **obiektKlasyNasłuchującej** implementujący interfejs nasłuchujący **RodzajListener**.

Obsługa interfejsu w klasie wewnętrznej

```
public class ObslugaZdarzen1 extends JFrame {

    //Zmienna wewnetrzna klasy
    int i = 10;
    // definicja klasy wewnętrznej - zwróć uwagę na "widoczność" zmiennych
    class MojInterfejs implements ActionListener{
        public void actionPerformed(ActionEvent arg0) {
            setTitle("Klasa wewnetrzna - wartość zmiennej i: " + i );
        }
    }

    public ObslugaZdarzen1() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        MojInterfejs mI = new MojInterfejs();
        b.addActionListener(mI);
        add(b);
    }

    public static void main(String[] args) {
        JFrame f = new ObslugaZdarzen1();
        f.setVisible(true);
    }
}
```

Obsługa interfejsu w klasie wewnętrznej

```
public class ObslugaZdarzen1 extends JFrame {

    //Zmienna wewnętrzna klasy
    int i = 10;

    // definicja klasy wewnętrznej - zwróć uwagę na "widoczność" zmiennych
    class MojInterfejs implements ActionListener{
        public void actionPerformed(ActionEvent arg0) {
            setTitle("Klasa wewnętrzna - wartość zmiennej i: " + i );
        }
    }

    public ObslugaZdarzen1() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        b.addActionListener(new MojInterfejs());
        add(b);
    }

    public static void main(String[] args) {
        JFrame f = new ObslugaZdarzen1();
        f.setVisible(true);
    }
}
```

Obsługa interfejsu w wewnętrznej klasie anonimowej

```
public class ObslugaZdarzen2 extends JFrame {

    public ObslugaZdarzen2() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        b.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                setTitle("Anonimowa klasa wewnetrzna");
            }
        });
        add(b);
    }

    public static void main(String[] args) {
        JFrame f = new ObslugaZdarzen2();
        f.setVisible(true);
    }
}
```

Interfejs implementowany w innej klasie publicznej zdefiniowanej w oddzielnym pliku - zmienna przekazana przez konstruktor.

```
public class ObslugaZdarzen3 extends JFrame {  
  
    public ObslugaZdarzen3() throws HeadlessException {  
        super();  
        setSize(600,200);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        setLayout(new FlowLayout());  
        JButton b = new JButton("Przycisk");  
        MojInterfejsPubliczny mIP = new MojInterfejsPubliczny(this);  
        b.addActionListener(mIP);  
        add(b);  
    }  
    public static void main(String[] args) {  
        JFrame f = new ObslugaZdarzen3();  
        f.setVisible(true);  
    }  
}
```

Klasa MojInterfejsPubliczny zdefiniowana w oddzielnym pliku...

Interfejs implementowany w innej klasie publicznej
zdefiniowanej w oddzielnym pliku - zmienna
przekazana przez konstruktor – c.d.

```
public class MojInterfejsPubliczny implements ActionListener {
    JFrame referencjaDoOkna;

    MojInterfejsPubliczny(JFrame zmiennaPrzekazanaWKonstruktorze){
        referencjaDoOkna = zmiennaPrzekazanaWKonstruktorze;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        referencjaDoOkna.setTitle("Interfejs w innej klasie publicznej");
    }
}
```


Interfejs implementowany w innej klasie publicznej zdefiniowanej w oddzielnym pliku – wymiana danych przez metody set... i get...

```
public class ObslugaZdarzen4 extends JFrame {  
  
    public ObslugaZdarzen4() throws HeadlessException {  
        super();  
        setSize(600,200);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        setLayout(new FlowLayout());  
        JButton b = new JButton("Przycisk");  
        MojInterfejsPubliczny2 mIP2 = new MojInterfejsPubliczny2();  
        mIP2.setReferencjaDoOkna(this);  
        b.addActionListener(mIP2);  
        add(b);  
    }  
    public static void main(String[] args) {  
        JFrame f = new ObslugaZdarzen4();  
        f.setVisible(true);  
    }  
}
```

Klasa **MojInterfejsPubliczny2** również zdefiniowana w oddzielnym pliku...

Interfejs implementowany w innej klasie publicznej zdefiniowanej w oddzielnym pliku – wymiana danych przez metody set... i get...

```
public class MojInterfejsPubliczny2 implements ActionListener {
    JFrame referencjaDoOkna;
    String nowyTytul = "Inna klasa publiczna - przekazywanie zmiennych przez metody
set... i get...";

    @Override
    public void actionPerformed(ActionEvent e) {
        referencjaDoOkna.setTitle(nowyTytul);
    }
// przekazanie referencji do okna, ktorego nazwa ma byc zmieniona, metoda set...
    void setReferencjaDoOkna(JFrame zmiennaPrzekazana){
        referencjaDoOkna = zmiennaPrzekazana;
    }
//dodatkowe metody get... i set... definiuje się w razie potrzeby w podobny sposób:
    void setTextInterfejsu(String przekazanyTekst){
        nowyTytul = przekazanyTekst;
    }
    String getTextInterfejsu(){
        return nowyTytul;
    }
}
```

Interfejs implementowany w klasie, do której dodawany jest komponent

```
public class ObsługaZdarzen5 extends JFrame implements ActionListener {

    public ObsługaZdarzen5() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        b.addActionListener(this);
        add(b);
    }

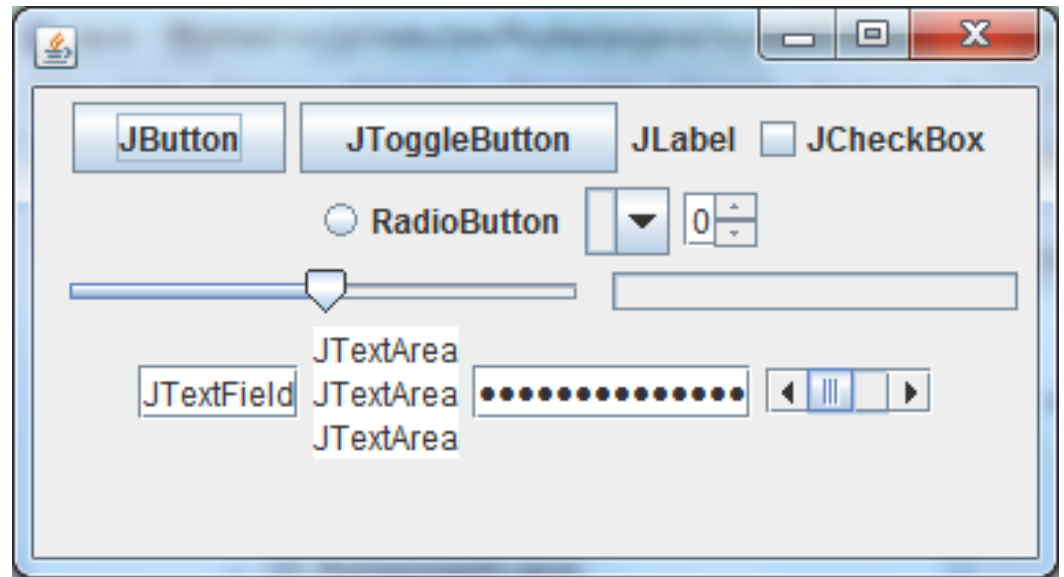
    public static void main(String[] args) {
        JFrame f = new ObsługaZdarzen5();
        f.setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        setTitle("Interfejs implementowany w tej samej klasie...");
    }

}
```

Kontrolki w Swing

- JButton
- JCheckBox
- JComboBox
- JList
- JMenu
- JRadioButton
- JSlider
- JSpinner
- JTextField
- JPasswordField



Komponenty.java

Kontrolki do wyświetlania informacji oraz kontrolki najwyższego poziomu

- JLabel
- JProgressBar
- JSeparator
- JToolTip
- JPanel
- JScrollPane
- JSplitPane
- JTabbedPane
- JToolBar
- JFrame
- JLayeredPane
- RootPane

Kontrolki z formatowaniem

- JColorChooser
- JEditorPane
- JTextPane
- JFileChooser
- JTable
- JTextArea
- JTree

JComponent

Wszystkie komponenty Swing których nazwy zaczynają się od „J” są pochodnymi klasy **JComponent**

Wybrane metody:

- **setToolTipText()** - ustawienie dymku podpowiedzi,
- **setBorder()** - ustawienie obramowania,
- **paintComponent()** - rysowanie na komponencie,
 - metody używane przez „zarządców układu” (layout managers) – metody `getPreferredSize`, `getAlignmentX`, `getMinimumSize`, `getMaximumSize`, `set....` (implementacja w klasach pochodnych)

Każdy komponent ma odpowiadający obiekt **ComponentUI**, który **przeprowadza rysowanie**, przechwytywanie zdarzeń, ustalanie rozmiaru, itd. jest on zależny od bieżącego wyglądu interfejsu, który ustawiamy poleceniem **UIManager.setLookAndFeel(...)**

Ogólny schemat dodawania i obsługi zdarzeń komponentów:

- deklaracja zmiennej (`Klasa obiekt;`)
- tworzenie nowego obiektu
`(obiekt = new Klasa(parametry_kons);)`
- ustawianie właściwości komponentu (preferowany rozmiar, tekst, kolor...)
- dodawanie interfejsu do obiektu:
`obiekt.addNazwaInterfejsu(obiekt_implementujący_interfejs);`
- dodawanie obiektu do okna lub np. panelu :
`add(obiekt), panel.add(obiekt), ...`
- odpowiednia modyfikacja metody obsługującej zdarzenia generowane przez obiekt w **obiekcie_implementującym_interfejs**

Wybrane interfejsy-zarządcy obsługi zdarzeń

- **ActionListener** – obsługuje zdarzenia generowane przez użytkownika na rzecz danego składnika interfejsu (np. kliknięcie przycisku)
- **AdjustmentListener** – obsługuje zdarzenie jako zmianę stanu składnika (np. przesuwanie suwaka w polu tekstowym)
- **FocusListener** – obsługuje zdarzenie od przejścia składnika w stan aktywny/nieaktywny
- **ItemListener** - obsługuje zdarzenie od np. zaznaczenia pola wyboru
- **KeyListener** - obsługuje zdarzenie np. od wpisywania tekstu z klawiatury
- **MouseListener** - obsługuje zdarzenie od naciśnięcia klawiszy myszy
- **MouseMotionListener** - obsługuje zdarzenie od przesuwania wskaźnika myszy nad danym składnikiem
- **WindowListener** - obsługuje zdarzenie od okna np. minimalizacja, maksymalizacja, przesunięcie, zamknięcie

Wiązanie wybranych składników z obsługą zdarzeń

- **addActionListener()** dla JButton, JCheckBox, JComboBox, JTextField, JRadioButton
- **addAdjustmentListene()** dla JScrollbar
- **addFocusListener()** dla wszystkich składników Swing
- **addItemListener()** dla JButton, JCheckBox, JComboBox, JTextField, JRadioButton
- **addKeyListener()** dla wszystkich składników Swing
- **addMouseListener()** dla wszystkich składników Swing
- **addMouseMotionListener()** wszystkich składników Swing
- **addWindowListener()** wszystkich obiektów typu JFrame oraz JWindow

Metody te muszą być zastosowane przed wstawieniem składnika do kontenera (JFrame, JPanel, ...)

JComponent

Wszystkie komponenty Swing których nazwy zaczynają się od „J” są pochodnymi klasy **JComponent**

Wybrane metody:

- **setToolTipText()** - ustawienie dymku podpowiedzi,
- **setBorder()** - ustawienie obramowania,
- **paintComponent()** - rysowanie na komponencie,
 - metody używane przez „zarządców układu” (layout managers) – metody `getPreferredSize`, `getAlignmentX`, `getMinimumSize`, `getMaximumSize`, `set....` (implementacja w klasach pochodnych)

Każdy komponent ma odpowiadający obiekt **ComponentUI**, który **przeprowadza rysowanie**, przechwytywanie zdarzeń, ustalanie rozmiaru, itd. jest on zależny od bieżącego wyglądu interfejsu, który ustawiamy poleceniem **UIManager.setLookAndFeel(...)**

– **setToolTipText()** – przykład (przy okazji także **GridLayout()**)

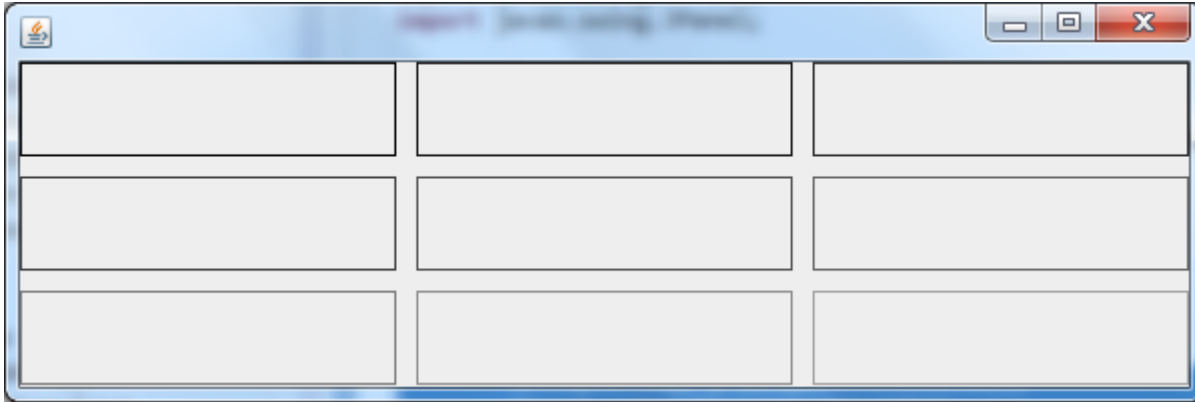


```
setLayout(new GridLayout(3,3,10,10));
JButton guziki[] = new JButton[9];
for (int i = 0; i<9; i++){
    guziki[i] = new JButton ("Przycisk" + i);
    guziki[i].setToolTipText("Podpowiedz przycisku nr " + i);
    add(guziki[i]);
}
```

- ToolTipTextDemo.java

setBorder()

- BorderDemo.java

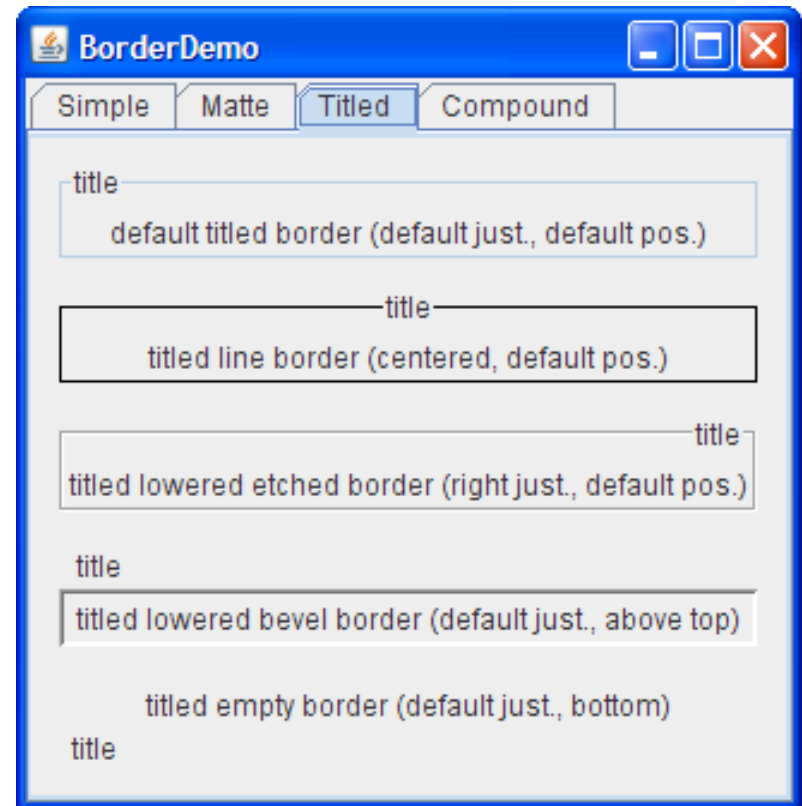
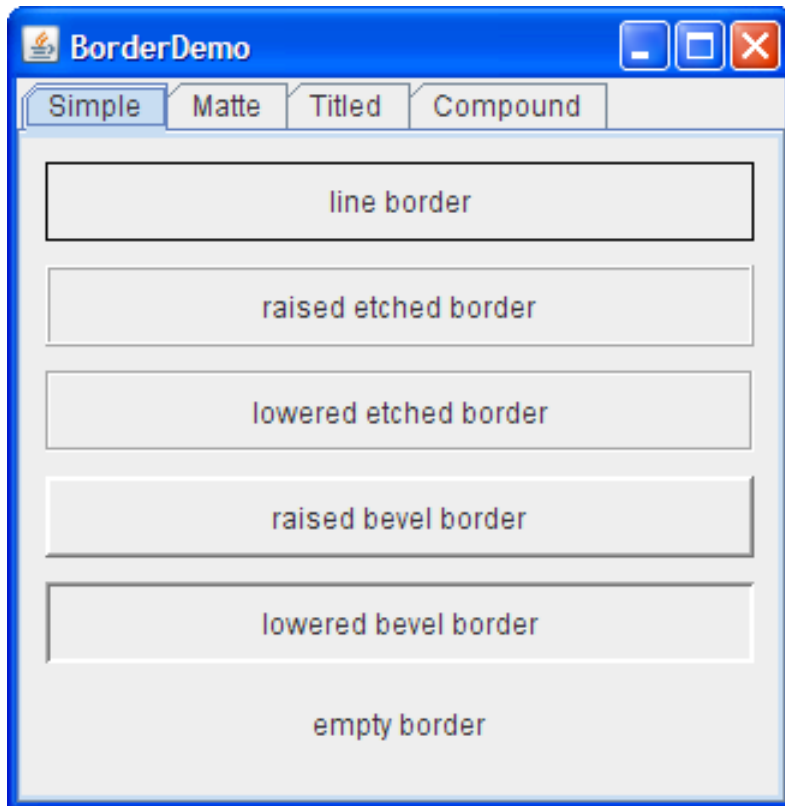


```
setLayout(new GridLayout(3,3,10,10));
JPanel panele[] = new JPanel[9];
for (int i = 0; i<9; i++){
    panele[i] = new JPanel ();
    panele[i].setBorder(BorderFactory.createLineBorder(new
                                                             Color(i*20,i*20,i*20)));
    add(panele[i]);
}
```

setBorder()

- Więcej typów ramek w dokumentacji:

<http://docs.oracle.com/javase/tutorial/uiswing/components/border.html>



Lista nasłuchiwaczy wspieranych przez wszystkie komponenty

- **Component Listener** – nasłuchuje zdarzeń związanych ze zmianą wielkości, widoczności oraz pozycji komponentu
- **Focus Listener** – sprawdza, czy komponent jest ustawiony fokus
- **Key Listener** – zdarzenia generowane są tylko przez komponent, na którym jest fokus
- **Mouse Listener** – nasłuchuje kliknięcia, przytrzymania klawisza, ruchu myszki wewnątrz lub poza komponent
- **Mouse-Motion Listener** – nasłuchuje zmian pozycji kursora
- **Mouse-Wheel Listener** – ruch kółka myszki ponad komponentem
- **Hierarchy Bound Listener** – zmiany w hierarchii zawierania komponentów dotyczący pozycji oraz skalowania

Przykład wykorzystania FocusListener

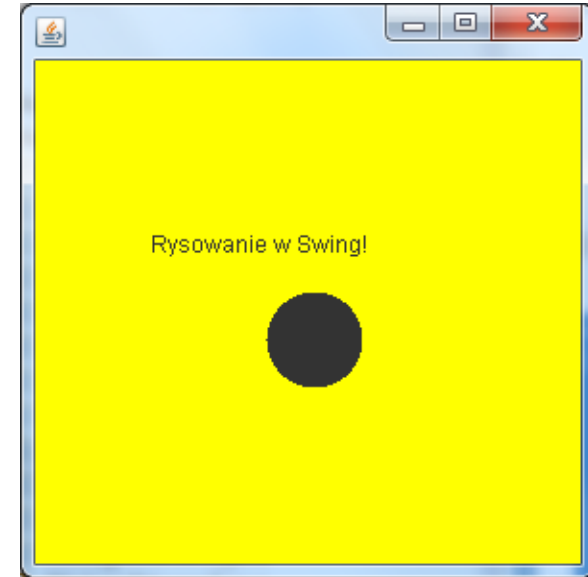
- FocusDemo.java

```
@Override
public void focusGained(FocusEvent arg0) {
    System.out.println("Focus gained on: " +
        arg0.getSource().getClass());
}
@Override
public void focusLost(FocusEvent arg0) {
    System.out.println("Focus lost on: " +
        arg0.getSource().getClass());
}
```


Proste rysowanie z wykorzystaniem paintComponent()

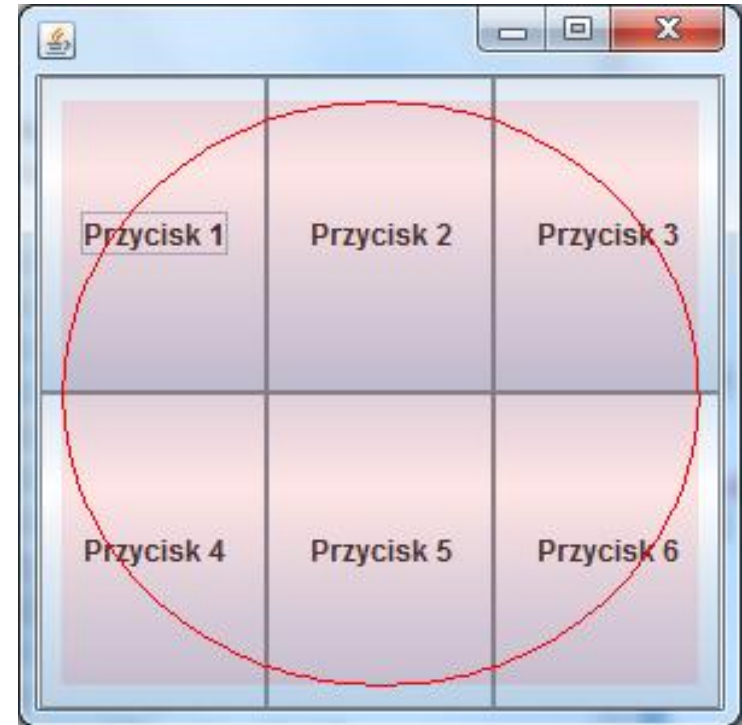
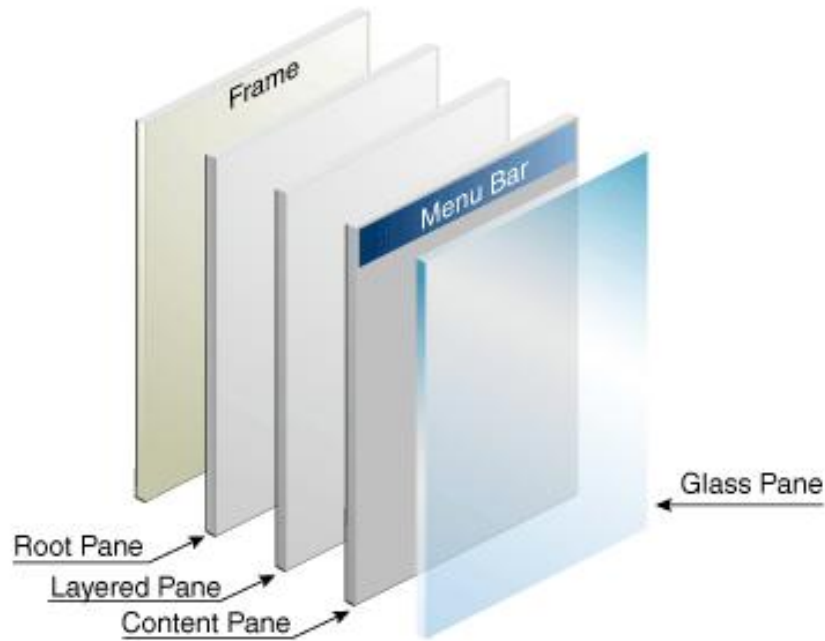
```
class PanelRysowania extends JPanel
{
    PanelRysowania()
    {
        setBackground(Color.yellow);
    }

    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);
        g.drawString("Rysowanie w Swing!", 60, 100);
        g.fillOval(120, 120, 50, 50);
    }
}
```



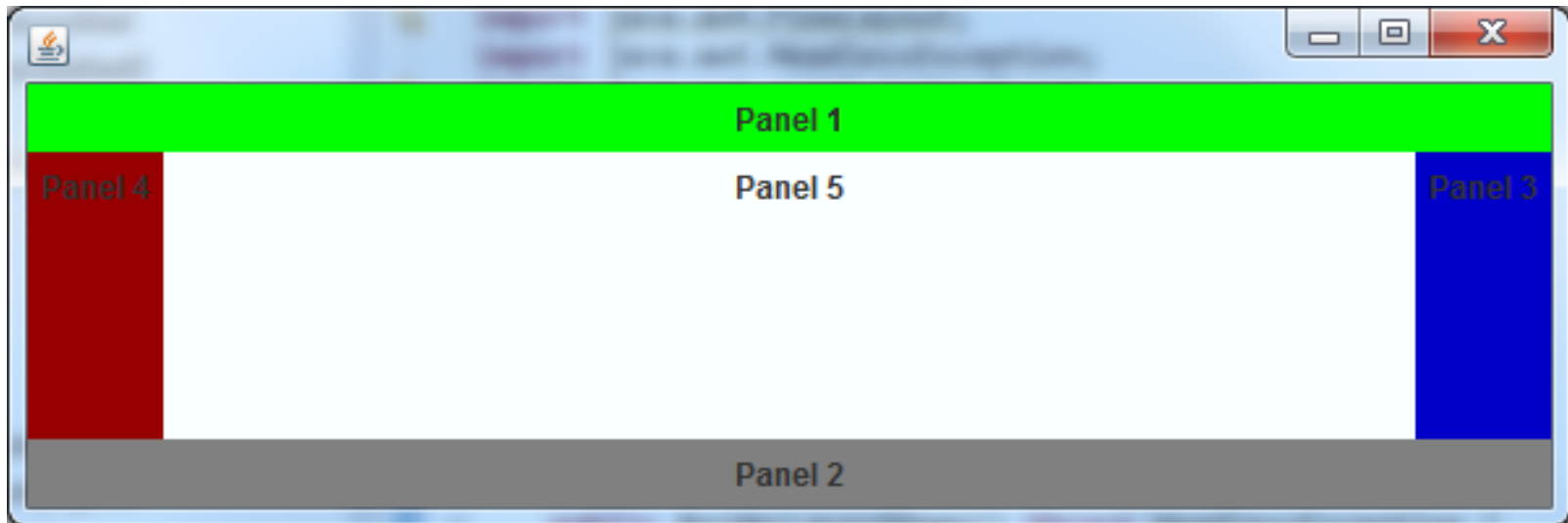
- Przykład: ProsteRysowanie.java

GlassPane można wykorzystać do uzyskiwania ciekawych efektów



- Przykład: `ProsteRysowanieGlass.java`

BorderLayout – wygodny do podziału okna na „funkcjonalne” obszary

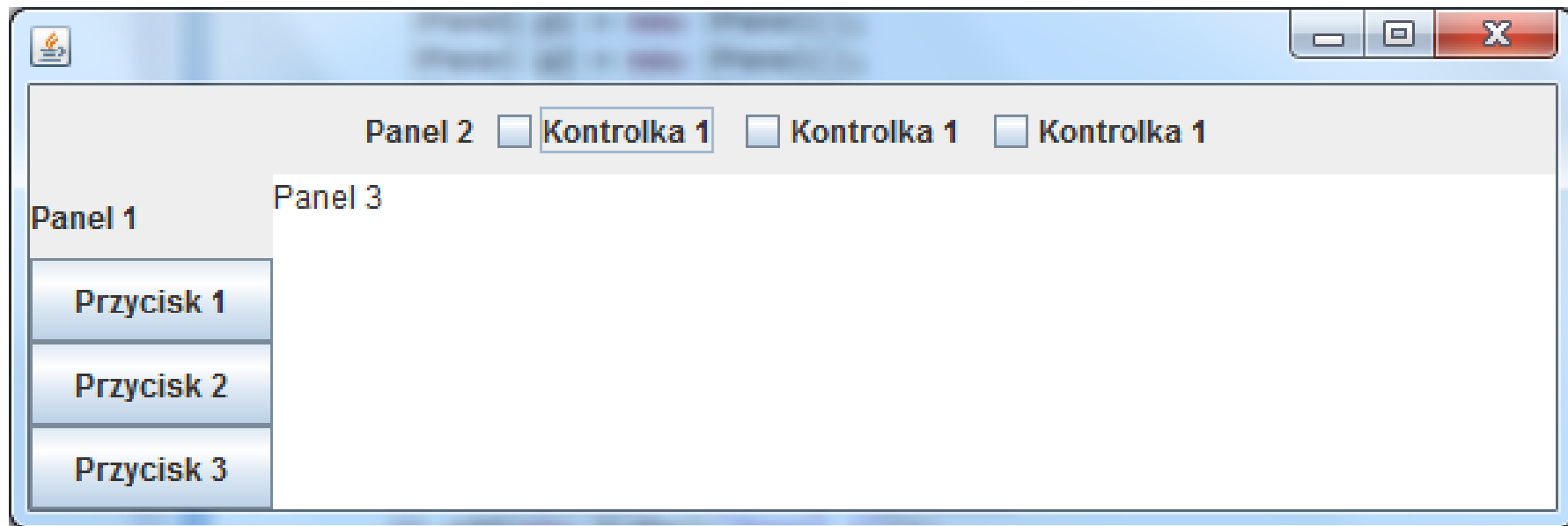


```
setLayout(new BorderLayout());  
JPanel p1 = new JPanel();  
JPanel p2 = new JPanel();  
JPanel p3 = new JPanel();  
JPanel p4 = new JPanel();  
JPanel p5 = new JPanel();  
add(BorderLayout.EAST, p3);  
add(BorderLayout.WEST, p4);  
add(BorderLayout.NORTH, p1);  
add(BorderLayout.SOUTH, p2);  
add(BorderLayout.CENTER, p5);
```

- BorderLayoutDemo.java

BorderLayout()

Nie wszystkie składowiki (NORTH, SOUTH, EAST, WEST, CENTER) muszą być dodane, a w ramach paneli można zastosować inny typ zarządzania rozmieszczeniem komponentów



- BorderLayoutDemo2.java