Rozdział 6 Grafika i multimedia w Javie

6.1 Grafika (rysunki)
6.2 Czcionki
6.3 Kolor
6.4 Obrazy
6.5 Dźwięki
6.6 Java Media API

6.1 Grafika (rysunki)

Pakiet AWT zarówno w wersjach wcześniejszych jak i w wersji 2 wyposażony jest w klasę Graphics, a od wersji 2 dodatkowo w klasę Graphics2D. Klasy te zawierają liczne metody umożliwiające tworzenie i zarządzanie grafiką w Javie. Podstawą pracy z grafiką jest tzw. kontekst graficzny, który jako obiekt posiada właściwości konkretnego systemu prezentacji np. panelu. W AWT kontekst graficzny jest dostarczany do komponentu poprzez następujące metody:

- paint
- paintAll
- update
- print
- printAll
- getGraphics

Obiekt graficzny (kontekst) zawiera informacje o stanie grafiki potrzebne dla podstawowych operacji wykonywanych przez metody Javy. Zaliczyć tu należy następujące informacje:

- obiekt komponentu, który będzie obsługiwany,
- współrzędne obszaru rysowania oraz obcinania,
- aktualny kolor,
- aktualne czcionki,
- aktualna funkcja operacji na pikselach logicznych (XOR lub Paint),
- aktualny kolor dla operacji XOR.

Posiadając obiekt graficzny można wykonać szereg operacji rysowania np.: Graphics g;

g.drawLine(int x1, int y1, int x2, int y2) - rysuje linię pomiędzy współrzędnymi (x1,y1) a (x2,y2), używając aktualnego koloru,

g.drawRect(int x, int y, int width, int height) - rysuje prostokąt o wysokości height i szerokości width począwszy od punktu (x,y), używając aktualnego koloru,

g.drawString(String str, int x, int y) - rysuje tekst str począwszy od punktu (x,y), używając aktualnego koloru i czcionki,

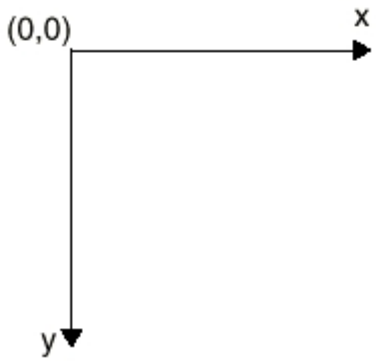
g.drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer) - wyświetla aktualnie dostępny zestaw pikseli obrazu img na tle o kolorze bgcolor,

począwszy od punktu (x,y)

g.setColor(Color c) - ustawia aktualny kolor c np. Color.red

g.setFont(Font font) - ustawia aktualny zestaw czcionek

W celu rysowania elementów grafiki konieczna jest znajomość układu współrzędnych w ramach, którego wyznacza się współrzędne rysowania. Podstawowym układem współrzędnych w Javie jest układ użytkownika, będący pewną abstrakcją układów współrzędnych dla wszystkich możliwych urządzeń. Układ użytkownika definiowany jest w sposób następujący.



Rysunek 6.1 Układ współrzędnych użytkownika

Układ współrzędnych konkretnego urządzenia może pokrywać się z układem użytkownika lub nie. W razie potrzeby współrzędne z układu użytkownika są automatycznie transformowane do właściwego układu współrzędnych danego urządzenia.
Jako ciekawostkę można podać, że JAVA nie definiuje wprost metody umożliwiającej rysowanie piksela, która w innych językach programowania służy często do tworzenia obrazów. W Javie nie jest to potrzebne. Do tych celów służą liczne metody drawImage(). Niemniej łatwo skonstruować metodę rysującą piksel np.

```
drawPixel(Color c, int x, int y){
   setColor(c);
   drawLine(x,y,x,y);
}
```

Pierwotne wersje AWT definiują kilka obiektów geometrii jak np. Point, Rectangle. Elementy te są bardzo przydatne dlatego, że, co jest właściwe dla języków obiektowych, nie definiujemy za każdym razem prostokąta za pomocą atrybutów opisowych (współrzędnych) lecz przez gotowy obiekt - prostokąt, dla którego znane są (różne metody) jego liczne właściwości.

Poniższa aplikacja umożliwia sprawdzenie działania prostych metod graficznych:

Przykład 6.1:

```
//Rysunki.java:

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

```
import java.awt.*;

public class Rysunki extends Frame {
   Rysunki () {
      super ("Rysunki");
      setSize(200, 220);
   }
   public void paint (Graphics g) {
       Insets insets = getInsets();
       g.translate (insets.left, insets.top);
       g.drawLine (5, 5, 195, 5);
       g.drawLine (5, 75, 5, 75);

      g.drawRect (25, 10, 50, 75);
       g.fillRect (25, 110, 50, 75);
       g.drawRoundRect (100, 10, 50, 75, 60, 50);
       g.fillRoundRect (100, 110, 50, 75, 60, 50);

       g.setColor(Color.red);
       g.drawString ("Test grafiki",50, 100);
       g.setColor(Color.black);
   }
   public static void main (String [] args) {
      Frame f = new Rysunki ();
       f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.out.println("Dziekujemy za prace z programem...");
            System.exit(0);
        }
      });
      f.setVisible(true);
   }
}// koniec public class Rysunki extends Frame
```

Wykorzystana w powyższym kodzie metoda translate() zmienia początek układu współrzędnych przesuwając go do aktywnego pola graficznego (bez ramek).

Java2D API w sposób znaczny rozszerza możliwości graficzne AWT. Po pierwsze umożliwia zarządzanie i rysowanie elementów graficznych o współrzędnych zmiennoprzecinkowych (float i double). Własność ta jest niezwykle przydatna dla różnych aplikacji m.in. dla aplikacji w medycynie (wymiarowanie, planowanie terapii, projektowanie implantów, itp.). Ta podstawowa zmiana podejścia do rysowania obiektów graficznych i geometrycznych powoduje powstanie, nowych, licznych klas i metod. W sposób szczególny należy wyróżnić tutaj sposób rysowania nowych elementów. Odbywa się to poprzez zastosowanie jednej metody:

```
Graphics2D g2;
g2.draw(Shape s);
```

Metoda draw umożliwia narysowanie dowolnego obiektu implementującego interfejs Shape (kształt). Przykładowo narysowanie linii o współrzędnych typu float można wykonać w następujący sposób:

```
Line2D linia = new Line2D.Float(20.0f, 10.0f, 100.0f, 10.0f);
g2.draw(linia);
```

Oczywiście klasa Line2D implementuje interfejs Shape. Java2D wprowadza liczne klasy w ramach pakietu java.awt.geom, np:

Arc2D.Double
Arc2D.Float
CubicCurve2D.Double
CubicCurve2D.Float
Dimension2D
Ellipse2D.Double
Ellipse2D.Float
GeneralPath
Line2D
Line2D.Double
Line2D.Float
Point2D
Point2D.Double
Point2D.Float
QuadCurve2D.Double
QuadCurve2D.Float
Rectangle2D
Rectangle2D.Double
Rectangle2D.Float
RoundRectangle2D.Double
RoundRectangle2D.Float

W celu skorzystania z tych oraz innych dobrodziejstw jakie wprowadza Java2D należy skonstruować obiekt graficzny typu Graphics2D. Ponieważ Graphics2D rozszerza klasę Graphics, to konstrukcja obiektu typu Graphics2D polega na:

```
Graphics2D g2 = (Graphics2D) g;
```

gdzie g jest obiektem graficznym otrzymywanym jak omówiono wyżej.

Uwaga! Argumentem metody paint komponentów jest obiekt klasy Graphics a nie Graphics2D.

Dodatkowe klasy w AWT wspomagające grafikę to BasicStroke oraz TexturePaint. Pierwsza z nich umożliwia stworzenie właściwości rysowanego obiektu takich jak np.: szerokość linii, typ linii. Przykładowo ustawienie szerokości linii na 12 punktów odbywać się może poprzez zastosowanie następującego kodu:

```
grubaLinia = new BasicStroke(12.0f);
g2.setStroke(grubaLinia);
```

Klasa TexturePoint umożliwia wypełnienie danego kształtu (Shape) określoną teksturą. Do dodatkowych zalet grafiki w Java2D należy zaliczyć:
- sterowanie jakością grafiki (np. antyaliasing, interpolacje)

- sterowanie przekształceniami geometrycznymi (przekształcenia sztywne - affiniczne
- klasa AffineTransform),
- sterowanie przeźroczystością elementów graficznych,
- bogate narzędzia do zarządzania czcionkami i rysowania tekstu,
- narzędzia do drukowania grafiki,
- inne.

Przykładowa aplikacja ukazująca proste elementy grafiki w Java2D

Przykład 6.2:

```
//Rysunki2.java:

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

public class Rysunki2 extends Frame {
   Rysunki2 () {
      super ("Rysunki2");
      setSize(200, 220);
   }
   public void paint (Graphics g) {

       Graphics2D g2 = (Graphics2D) g;
       Insets insets = getInsets();
       g2.translate (insets.left, insets.top);

       Line2D linia = new Line2D.Float(20.0f, 20.0f, 180.0f, 20.0f);
       g2.draw(linia);

       BasicStroke grubaLinia = new BasicStroke(6.0f);
       g2.setStroke(grubaLinia);
       g2.setColor(Color.red);
       Line2D linia2 = new Line2D.Float(20.0f, 180.0f, 180.0f, 180.0f);
       g2.draw(linia2);
       g2.drawString ("Test grafiki",50, 100);
       g2.setColor(Color.black);
   }
   public static void main (String [] args) {
      Frame f = new Rysunki2 ();
      f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.out.println("Dziekujemy za prace z programem...");
            System.exit(0);
        }
      });
      f.setVisible(true);
   }
} // koniec public class Rysunki2 extends Frame
```

6.2 Czcionki

W Javie można korzystać z różnych typów czcionek, które związane są z daną platformą na jakiej pracuje Maszyna Wirtualna. Dostęp do czcionek odbywa się poprzez trzy typy nazw: nazwy logiczne czcionek, nazwy czcionek, nazwy rodziny czcionek. Nazwy logiczne czcionek to nazwy zdefiniowane dla Javy. Możliwe są następujące nazwy logiczne czcionek w Javie: Dialog, DialogInput, Monospaced, Serif, SansSerif, oraz Symbol. Nazwy logiczne są odwzorowywane na nazwy czcionek powiązane z czcionkami dla danego systemu. Odwzorowanie to występuje w pliku font.properties znajdującego się w katalogu lib podkatalogu jre (Java Runtime Engine). W pliku tym zdefiniowano również sposób kodowania znaków. Z tego powodu w zależności od sposobu kodowania i typów wykorzystywanych czcionek definiowane są różne pliki font.properties z odpowiednim rozszerzeniem np. font.properties.pl. Przykładowe fragmenty plików opisujący właściwości czcionek to np.:

```
//Windows NT font.properties:
(...)
serif.0=Times New Roman,ANSI_CHARSET
serif.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
(...)

//Solaris font.properties.pl:
(..)
serif.plain.0=-linotype-times-medium-r-normal--*-%d-*-*-p-*-iso8859-1
serif.1=-monotype-timesnewroman-regular-r-normal--*-%d-*-*-p-*-iso8859-2
(...)
```

Różnica pomiędzy nazwą logiczną czcionek w Javie a ich nazwami jest niewielka, np. nazwa logiczna Serif, nazwa serif. Różnice nazw ukazuje poniższy program:

Przykład 6.3:

```
//Nazwy.java

import java.awt.*;

public class Nazwy {

        private Font f;

        Nazwy (){

                f = new Font("Serif",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Serif: "+f.getName());
                System.out.println("Oto nazwa  czcionki Serif: "+f.getFontName());
                f = new Font("SansSerif",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki SansSerif: " + f.getName());
                System.out.println("Oto nazwa  czcionki SansSerif: " +f.getFontName());
                f = new Font("Dialog",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Dialog: "+f.getName());
                System.out.println("Oto nazwa  czcionki Dialog: "+f.getFontName());
```

```
                f = new Font("DialogInput",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki DialogInput: " + f.getName());
                System.out.println("Oto nazwa  czcionki DialogInput: "+ f.getFontName());
                f = new Font("Monospaced",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Monospaced: "+ f.getName());
                System.out.println("Oto nazwa  czcionki Monospaced: "+ f.getFontName());
                f = new Font("Symbol",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Symbol: "+ f.getName());
                System.out.println("Oto nazwa  czcionki Symbol: "+f.getFontName());
        }

        public static void main(String args[]){
                Nazwy czcionki = new Nazwy();

        }
}// koniec public class Nazwy
```

W pracy z Javą w celu stworzenia uniwersalnego kodu należy korzystać z nazw logicznych. Dla danej platformy można oczywiście używać nazw fontów tam zainstalowanych. W celu uzyskania informacji o zainstalowanych czcionkach na danej platformie należy posłużyć się metodami klasy GraphicsEnvironment (w poprzednich wersjach JDK należało korzystać z metody getFontList() klasy Toolkit). Poniższy program ukazuje zainstalowane czcionki w systemie poprzez podanie nazw rodzin czcionek oraz poszczególnych nazw.

Przykład 6.4:

```
//Czcionki.java:


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


public class Czcionki extends Frame  {

        public String s[];
        private Font czcionki[];
        private Font f;

        Czcionki (String nazwa){
                super(nazwa);
                f = new Font("Verdana",Font.BOLD,12);
                s=GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
                czcionki=GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts();


        }

        public static void main(String args[]){
                Czcionki okno = new Czcionki("Lista czcionek");
                okno.setSize(600,500);
                okno.setLayout(new GridLayout(2,1));
```

```
                List lista1 = new List(1, false);
                for (int i=0; i<okno.s.length; i++){
                        lista1.add(okno.s[i]);
                }
                lista1.setFont(okno.f);
                okno.add(lista1);

                List lista2 = new List(1, false);
                for (int i=0; i<okno.czcionki.length; i++){
                        lista2.add(okno.czcionki[i].toString());
                }
                lista2.setFont(okno.f);
                okno.add(lista2);

                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 Czcionki
```

Pełny opis czcionek można uzyskać posługując się metodami zdefiniowanymi w klasie FontMetrics. Podstawowe metody umożliwiają między innymi uzyskanie informacji o szerokości znaku lub znaków dla wybranej czcionki (charWithd(), charsWidth()), o wysokości czcionki (getHight()), o odległości między linią bazową czcionki a wierzchołkiem znaku (getAscent()), o odległości między linią bazową czcionki a podstawą znaku (getDescent()), itd.
Ustawianie czcionek dla danego komponentu polega na stworzeniu obiektu klasy Font a następnie na wywołaniu metody komponentu setFont(), np.:

```
(...)
Component c;
(...)
Font f = new Font („Arial”, Font.PLAIN, 12);
c.setFont(f);
(...)
```

Wywołanie konstruktora klasy Font w powyższym przykładzie wymaga określenia nazwy lub nazwy logicznej czcionki, stylu, oraz rozmiaru czcionki. Styl czcionki może być określony przez następujące pola klasy Font:

Font.PLAIN – tekst zwykły,
Font.ITALIC – tekst pochyły (kursywa),
Font.BOLD – tekst pogrubiony,
Font.ITALIC + Font.BOLD – tekst pogrubiony i pochyły.

Do tej pory omówione zostały podstawowe własności grafiki dostarczanej przez Java2D z pominięciem obsługi obrazów i kolorów. Zarządzanie kolorami i metody definiowania, wyświetlania i przetwarzania obrazów są niezwykle istotne z punktu widzenia aplikacji medycznych. Dlatego teraz te zagadnienia zostaną osobno omówione.

6.3 Kolor

Kolor jest własnością postrzegania promieniowania przez człowieka. Średni obserwator (definiowany przez różne organizacje normalizacyjne) postrzega konkretne kolory jako wynik rejestracji promieniowania elektromagnetycznego o konkretnych długościach fal. Ponieważ system wzrokowy człowieka wykorzystuje trzy typy receptorów do rejestracji różnych zakresów promieniowania (będziemy dalej nazywać te zakresy zakresami postrzeganych kolorów) również systemy sztuczne tworzą podobne modele reprezentacji kolorów. Najbliższym spektralnie systemem wykorzystywanym dla celów reprezentacji kolorów jest system RGB (posiadający różne wady np. brak reprezentacji wszystkich barw, różnice kolorów często inne od tych postrzeganych przez człowieka (nieliniowa zależność postrzegania zmian intensywności), itp.). System RGB jest zależny od urządzenia, stąd istnieją różne jego wersje np. CIE RGB, PAL RGB, NTSC RGB, sRGB.

Standardowo przyjmuje się, że kolor na podstawie przestrzenni RGB definiowany jest jako liniowa kombinacja barw podstawowych:

kolor = r*R+g*G+b*B,

gdzie RBG to kolory (długości fal) podstawowe, a rgb to wartości stymulujące (tristimulus values) wprowadzające odpowiednie wagi z zakresu (0,1). Zakres wag w systemach komputerowych jest przeważnie modyfikowany do postaci zakresu dyskretnych wartości całkowitych o dynamice 8 bitowej czyli 0..255. Stąd też programista definiuje kolor jako zbiór trzech wag z zakresu 0..255, np., w Javie Color(0,0,255) daje barwę niebieską. W Javie istnieją predefiniowane stałe statyczne w klasie Color reprezentujące najczęściej wykorzystywane kolory np. Color.blue. Domyślną przestrzenią kolorów w Javie jest sRGB (wprowadzana jako standard dla WWW - http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html), niemniej można korzystać z innych przestrzenni kolorów np. niezależnej sprzętowo CIE XYZ (CIE - Commission Internationale de L'Eclairage). Wprowadzono specjalny pakiet jawa.awt.color obsługujący przestrzenie kolorów oraz standardowe profile urządzeń bazując na specyfikacji ICC Profile Format Specification, Version 3.4, August 15, 1997, International Color Consortium. Podsumowując należy wskazać najbardziej istotne klasy w Javie związane z kolorem:

- java.awt.color.ColorSpace
- java.awt.Color
- java.awt.image.ColorModel

Warto zwrócić uwagę, że każda z wymienionych klas występuje w innym miejscu. Do tej pory omówiona została krótko rola klasy ColorSpace oraz Color. Ostatnia z prezentowanych klas ColorModel jest związana (jak nazwa pakietu sugeruje) z tworzeniem i wyświetlaniem obrazów. Klasa ColorModel opisuje bowiem konkretną metodę odwzorowania wartości piksela na dany kolor. W celu określenia koloru piksela zgodnie z przyjętą przestrzenią kolorów definiuje się w wybranym modelu komponenty kolorów i przeźroczystości (np. Red Green Blue Alpha). Metodę definiowania tych komponentów określa właśnie ColorModel.

Najbardziej znane formy zapisu komponentów to tablice komponentów lub jedna wartość 32 bitowa, gdzie każde 8 bitów wykorzystywane jest do przechowywania informacji o danych komponencie. Wykorzystywane podklasy klasy abstrakcyjnej ColorModel to: ComponentColorModel, IndexColorModel oraz PackedColorModel.

Pierwszy model - ComponentColorModel jest uniwersalną wersją przechowywania współrzędnych w wybranej przestrzeni kolorów, stąd nadaje się do reprezentacji dowolnej przestrzeni kolorów. Każda próbka (komponent) w tym modelu jest przechowywana oddzielnie. Dla każdego więc piksela tworzony jest zbiór odseparowanych próbek. Ilość próbek na piksel musi być taka sama jak ilość współrzędnych w przyjętej przestrzeni kolorów. Kolejność występowania próbek jest definiowana przez przestrzeń kolorów, przykładowo dla przestrzeni RGB - TYPE_RGB, index 0 oznacza próbkę dla komponentu RED, index 1 dla GREEN, index 2 dla BLUE. Typ danych dla przechowywania próbek może być 8-bitowy, 16-bitowy lub 32-bitowy: DataBuffer.TYPE_BYTE, DataBuffer.TYPE_USHORT, DataBuffer.TYPE_INT. Wywołując konstruktor tej klasy podaje się przede wszystkim przestrzeń kolorów, oraz tablicę o długości odpowiadającej liczbie komponentów w danej przestrzeni przechowującą dla każdego komponentu liczbę znaczących bitów.
Kolejny model - IndexColorModel - jest szczególnie wykorzystywany w aplikacjach medycznych, ponieważ odwołuje się on do koloru nie poprzez zbiór próbek lecz poprzez wskaźnik do tablicy kolorów. Definiując model kolorów zgodnie z tą klasą tworzymy tablicę kolorów, do której później odwołujemy się podając wskaźnik. Model taki jest wykorzystywany w różnych aplikacjach tam, gdzie tworzy się tak zwane palety kolorów, tablice kolorów (pseudokolorów). Jest to szczególnie ważne tam, gdzie posiadane wartości do tworzenia obrazu (wartości pikseli) nie reprezentują promieniowania widzialnego. Mówi się wówczas o tak zwanych sztucznych obrazach i tablicach pseudo-kolorów (np. obrazy w podczerwieni, obrazy ultradźwiękowe, obrazy Rtg, itp.). W medycynie prawie wszystkie metody obrazowania wykorzystują tablice pseudokolorów, a właściwie jedną tablicę - odcieni szarości (najczęściej R=G=B). Wywołując jeden z konstruktorów klasy IndexColorModel podaje się liczbę bitów na piksel, rozmiar tablicy oraz konkretne definicje kolorów poprzez zbiór trzech podstawowych próbek dla RGB.
Ostatni z przedstawianych modeli - PackedColorModel - jest abstrakcyjną klasą w ramach której kolor jest oznaczany dla danej przestrzeni kolorów poprzez definicję wszystkich komponentów oddzielnie, spakowanych w jednej liczbie 8, 16 lub 32 bitowej. Podklasą tej abstrakcyjnej klasy jest DirectColorModel. Przy wywołaniu konstruktora podaje się liczbę bitów na piksel oraz wartości masek dla każdego komponentu celem wskazania wartości tych komponentów. Obecnie model ten wykorzystuje się jedynie dla przestrzeni sRGB.

6.4 Obrazy

Możliwości pozyskiwania, tworzenia i przetwarzania obrazów w Javie są bardzo duże. Począwszy od pierwszych wersji w ramach AWT poprzez Java2D a skończywszy na tworzonej JAI (Java Advanced Imaging API - bardziej elastyczne definicje obrazów, bogate biblioteki narzędzi np. DFT) język JAVA dostarcza wiele narzędzi do tworzenia profesjonalnych systemów syntezy, przetwarzania, analizy i prezentacji obrazów.

W początkowych wersjach bibliotek graficznych Javy podstawą pracy z obrazami była klasa java.awt.Image. Obiekty tej abstrakcyjnej klasy nadrzędnej uzyskiwane są w sposób zależny od urządzeń. Podstawowa metoda zwracająca obiekt typu Image (klasa Image jest abstrakcyjna), często wykorzystywana w programach tworzonych w Javie to getImage(). Metoda ta dla aplikacji jest związana z klasą Toolkit, natomiast dla appletów z klasą Applet. Wywołanie metody getImage polega albo na podaniu ścieżki dostępu (jako String) lub lokalizatora URL do obrazu przechowywanego w formacie GIF lub JPEG. Przykładowo:

Image obraz = Toolkit.getDefaultToolkit.getImage("jacek.gif");
- zwraca obiekt obraz na podstawie obrazu przechowywanego w pliku jacek.gif w bieżącej ścieżce dostępu

Image obraz = Toolkit.getDefaultToolkit.getImage(new URL("http://www-med.eti.pg.gda.pl/~jwr/icons/jwrs4.gif");
- zwraca obiekt obraz na podstawie obrazu przechowywanego w pliku jwrs4.gif na serwerze www-med w danych podkatalogach.

Inną metodą uzyskania obiektu Image jest stworzenie obrazu poprzez wykorzystanie metody createImage(). Aby uzyskać obiekt typu Image za pomocą tej metody jako argument należy podać element implementujący interfejs ImageProducer. Obiekt będący wystąpieniem klasy implementującej ten interfejs jest odpowiedzialny za stworzenie (produkcję) obrazu jaki jest związany z obiektem typu Image. Przykładowo w poprzednich metodach getImage() obiekt typu Image jest często zwracany wcześniej niż stworzony zostanie (np. załadowany z sieci) obraz. Wówczas metody odwołujące się do obiektu Image zwrócą błąd. Dlatego obiekt typu ImageProducer informuje klientów (obserwatorów) o zakończonym procesie tworzenia obrazu związanego z obiektem typu Image. Klientów stanowią obiekty klas implementujących interfejs ImageObserver, których zachowanie jest ukazane poprzez jedyną metodę imageUpdate(). Metoda ta zgodnie z informacją od obiektu ImageProducer żąda odrysowania elementu (np. komponentu graficznego jak np. panel, applet, itd. - java.awt.Component implementuje interfejs ImageObserver). W czasie gdy ImageProducer wysyła informację o stanie do obiektu ImageObserver wysyła również dane do konsumenta czyli obiektu będącego wystąpieniem klasy implementującej interfejs ImageConsumer. Przykładowe klasy implementujące interfejs ImageConsumer to java.awt.image.ImageFilter oraz java.awt.image.PixelGrabber. Ponieważ do obiektu ImageConsumer dostarczane są wszystkie informacje związane z tworzonym obrazem (wymiary, piksele, model koloru) obiekt ten może za pomocą swoich metod wykonać wiele operacji na danych. Przykładowo obiekty klasy ImageFilter umożliwiają wycinanie próbek, filtrację kolorów, itp. natomiast obiekty klasy PixelGrabber umożliwiają pozyskać część lub wszystkie próbki z obrazu. Powracając do metody createImage(), której argumentem jest obiekt ImageProducer, umożliwia ona stworzenie (generację) obrazu na podstawie posiadanych danych. Konieczne jest jednak stworzenie obiektu typu ImageProducer, będącego argumentem tej metody. W tym celu wykorzystuje się klasę implementującą interfejs ImageProducer - MemoryImageSource. Klasa ta dostarcza szereg konstruktorów, którym podaje się: typ modelu kolorów (ColorModel) lub wykorzystuje się domyślny RGB, rozmiar tworzonego obrazu, wartości pikseli. Przykładowo w następujący sposób można wygenerować własny obiekt typu Image:

Przykład 6.5:

```
//Obraz.java:

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

public class Obraz extends Frame {

Image ob;

   Obraz () {
       super ("Obraz");
       setSize(200, 220);
       ob=stworzObraz();
   }

   public Image stworzObraz(){
      int w = 100;  //szerokość obrazu
      int h = 100;   //wysokość obrazu
      int pix[] = new int[w * h]; //tablica wartości próbek
      int index = 0;
      //generacja przykładowego obrazu
      for (int y = 0; y < h; y++) {
         int red = (y * 255) / (h - 1);
         for (int x = 0; x < w; x++) {
            int blue = (x * 255) / (w - 1);
            pix[index++] = (255 << 24) | (red << 16) | blue;
         }
      }
      Image img = createImage(new MemoryImageSource(w, h, pix, 0, w));
      //tworzony jest obraz w RGB o szerokości w, wysokości h,
      //na podstawie tablicy próbek pix, bez przesunięcia w tej tablicy z w elementami w linii
      return img;
   }

   public void paint (Graphics g) {
     Insets insets = getInsets();
     g.translate (insets.left, insets.top);
     g.drawImage(ob,50,50,this);
   }

   public static void main (String [] args) {
      Frame f = new Obraz ();
       f.addWindowListener(new WindowAdapter(){
           public void windowClosing(WindowEvent e){
               System.out.println("Dziekujemy za prace z programem...");
               System.exit(0);
           }
      });
      f.setVisible(true);
   }
}//koniec public class Obraz extends Frame
```

Mając obiekt typu Image można obraz z nim związany wyświetlić (narysować). W tym celu należy wykorzystać jedną z metod drawImage(). W najprostszej metodzie drawImage() podając jako argument obiekt typu Image, współrzędne x,y oraz obserwatora (ImageObserver, często w ciele klasy komponentu, w którym się rysuje odwołanie do obserwatora jest wskazaniem aktualnego obiektu komponentu - this) możemy wyświetlić obrazu w dozwolonym do tego elemencie. Inna wersja metody drawImage() umożliwia skalowanie wyświetlanego obrazu poprzez podanie dodatkowych argumentów: szerokość (width) i wysokość (height).
Oprócz poznanych do tej pory operacji związanych z obrazami (stworzenie obiektu Image, filtracja danych, wyświetlenie i skalowanie) często wykorzystuje się dwie z kilku istniejących metod klasy Image, a mianowicie Image.getWidth() oraz Image.getHeight(). Metody te umożliwiają poznanie wymiarów obrazu, co jest często niezwykle istotne z punktu widzenia ich wyświetlania i przetwarzania. Argumentem obu metod jest ImageObserver (powiadomienie o dostępnych danych) po to aby można było uzyskać dane o stworzonym obrazie dla istniejącego już obiektu Image.

Opisane dotąd metody pracy z obrazami są historycznie pierwsze, a ich znaczne rozszerzenie daje Java2D. Java2D API rozszerza a zarazem zmienia koncepcję pracy z obrazami w Javie. Podstawową klasą jest w tej koncepcji klasa BufferedImage, będącą rozszerzeniem klasy Image z dostępnym buforem danych. Obiekt BufferedImage może być stworzony bezpośrednio w pamięci i użyty do przechowywania i przetwarzania danych obrazu uzyskanego z pliku lub poprzez URL.
Obraz BufferedImage może być wyświetlony poprzez użycie obiektów klasy Graphics2D. Obiekt BufferedImage zawiera dwa istotne obiekty: obiekt danych - Raster oraz model kolorów ColorModel. Klasa Raster umożliwia zarządzanie danymi obrazu. Na obiekt tej klasy składają się obiekty DataBuffer oraz SampleModel. DataBuffer stanowi macierz wartości próbek obrazu, natomiast SampleModel określa sposób interpretacji tych próbek. Przykładowo dla danego piksela próbki (RGB) mogą być przechowywane w trzech różnych macierzach (banded interleved) lub w jednej macierzy w formie przeplatanych próbek (pixel interleved) dla różnych komponentów (R1G1B1R2G2B2...). Rolą SampleModel jest określenie jakiej formy użyto do zapisu danych w macierzach. Najczęściej nie tworzy się bezpośrednio obiektu Raster lecz wykorzystuje się efekt działania obiektu BufferedImage, który rozbija Image na Raster oraz ColorModel. Niemniej istnieje możliwość stworzenia obiektu Raster poprzez stworzenie obiektu WritableRaster i podaniu go jako argumentu w jednym z konstruktorów klasy BufferedImage. Najbardziej popularne rozwiązanie tworzące obiekt klasy BufferedImage przedstawia następujący przykład:

```
URL url = ...

Image img = getToolkit().getImage(url);
try {
    //pętla w której czekamy na skonczenie produkcji obrazu dla obiektu img
    MediaTracker mt = new MediaTracker(this);
    mt.addImage(img, 0);
    mt.waitForID(0);
} catch (Exception e) {}
int iw = img.getWidth(this);
int ih = img.getHeight(this);
```

```
BufferedImage bi = new BufferedImage(iw, ih, BufferedImage.TYPE_INT_RGB);
//tworzymy obiekt BufferedImage
Graphics2D g2 = bi.createGraphics(); //określamy kontekst graficzny dla obiektu
g2.drawImage(img, 0, 0, this);
//wrysowujemy obraz do bufora - wypełniamy DataBuffer obiektu BufferedImage
```

Dla tak stworzonego obiektu BufferedImage można następnie przeprowadzić liczne korekcje graficzne, dodać elementy graficzne, zastosować metody przetwarzania obrazu oraz wyświetlić obraz. Wyświetlenie obrazu obiektu BufferedImage odbywa się poprzez wykorzystanie metod drawImage() zdefiniowanych dla klasy Graphics2D (np. g2.drawImage(bi,null,null);). Korekcje graficzne i dodawanie elementów graficznych polega na rysowaniu w stworzonym kontekście graficznym dla obiektu BufferedImage. Przetwarzanie obrazu odbywa się głównie poprzez wykorzystanie klas implementujących interfejs BufferedImageOp a mianowicie:
AffineTransformOp, BandCombineOp, ColorConvertOp, ConvolveOp, LookupOp, oraz RescaleOp.

Do najczęściej wykorzystywanych operacji należą przekształcenia geometryczne sztywne (affiniczne) - aplikacja AffineTransformOp - oraz operacje splotu. Te ostatnie wykorzystuje się do tworzenia filtrów cyfrowych, za pomocą których można zmieniać jakość obrazu oraz wykrywać elementy obrazu jak np. linie, punkty, itp. Poniższy przykład ukazuje możliwość implementacji filtru cyfrowego oraz przykładową operację skalowania.

```
float[] SHARPEN3x3_3 = { 0.f, -1.f,  0.f,              //maska filtru cyfrowego
                        -1.f,  5.f, -1.f,
                         0.f, -1.f,  0.f};
BufferedImage bi=...

AffineTransform at = new AffineTransform();
at.scale(2.0, 2.0);
//określenie operacji geometrycznej - skalowanie wsp. 2
BufferedImageOp biop = null;
BufferedImage bimg = new BufferedImage (w,h,BufferedImage. TYPE_INT_RGB);
                                          // tworzymy nowy bufor
Kernel kernel = new Kernel(3,3,SHARPEN3x3_3);  //tworzymy kernel - jądro splotu
ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP, null);
  //definiujemy operację splotu z przepisywaniem wartości krawędzi
cop.filter(bi,bimg); //wykonujemy operację splotu
biop = new AffineTransformOp(at, AffineTransformOp.TYPE_NEAREST_NEIGHBOR);
   //definiujemy operację skalowania i interpolacji obrazu
g2.drawImage(bimg,biop,x,y); //rysujemy skalowany obraz
```

Poniższy przykład ukazuje możliwość implementacji filtru dolnoprzepustowego:

Przykład 6.6:

```
//Obraz2.java:

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

```
import java.awt.*;

public class Obraz2 extends Frame {

BufferedImage bi;
boolean stan=false;
Button b1;

   Obraz2 () {
      super ("Obraz2");
      setSize(200, 220);
      gUI();
      bi=stworzObraz();
   }

   public void gUI(){
       setLayout(new BorderLayout());
       Panel kontrola = new Panel();
       b1 = new Button("Filtracja");
       b1.addActionListener(new B1());
       kontrola.add(b1);
       add(kontrola,BorderLayout.SOUTH);
   }

   public BufferedImage stworzObraz(){

       int w = 100;  //szerokość obrazu
      int h = 100;   //wysokość obrazu
      int pix[] = new int[w * h]; //tablica wartości próbek
      int index = 0;
      for (int y = 0; y < h; y++) {
         int red = (y * 255) / (h - 1);
         for (int x = 0; x < w; x++) {
            int blue = (x * 255) / (w - 1);
            pix[index++] = (255 << 24) | (red << 16) | blue;
         }
      }
       Image img = createImage(new MemoryImageSource(w, h, pix, 0, w));
           BufferedImage bimg = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
       Graphics2D g2 = bimg.createGraphics();
       g2.drawImage(img, 0, 0, this);
       return bimg;
   }


   public void paint (Graphics g) {
      Insets insets = getInsets();
     g.translate (insets.left, insets.top);
     Graphics2D g2d = (Graphics2D) g;
     g2d.drawImage(bi,null,50,50);
   }

   class B1 implements ActionListener {
       public void actionPerformed(ActionEvent e) {
           float lp1=1.0f/9.0f;
```

```
            float l=4.0f*lp1;
            float[] lp = { lp1, lp1,  lp1,
                           lp1,  l, lp1,
                           lp1, lp1,  lp1};

            if (!stan){
              Kernel kernel = new Kernel(3,3,lp);
              ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP, null);
               bi = cop.filter(bi,null);
               stan=true;
               b1.setLabel("Powrot");
            }else
            {
               bi=stworzObraz();
               stan=false;
               b1.setLabel("Filtracja");
            }
            validate();
            repaint();
     }
    }

    public static void main (String [] args) {
        Frame f = new Obraz2 ();
         f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
               System.out.println("Dziekujemy za prace z programem...");
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}// koniec  public class Obraz2 extends Frame
```

6.5 Dźwięki

Do wersji JDK 1.2 można korzystać z obsługi plików dźwiękowych w formacie au (8-bit, mono), tylko w obrębie apletu. W celu odtworzenia dźwięku zapisanego w pliku o formacie au należy wykorzystać jedną z dwóch metod zdefiniowaną w klasie Applet:

public void play(URL url); gdzie „url” to adres pliku wraz z nazwą pliku,
lub
public void play(URL url, String nazwa); gdzie „url” to adres pliku, a „nazwa” to nazwa pliku.

Przykładowo można zastosować następujący kod:

Przykład 6.7:

//Graj.java:

```
import java.applet.*;

public class Graj extends Applet{

        public void init(){
                String nazwa = getParameter(„muzyka”);
                if(nazwa != null) play(getDocumentBase(),nazwa);
        }

}// koniec public class Graj

//-------------------------------------------
//Graj.html:

(...)
<param name=muzyka value=witam.au>
(...)
```

Powyższy przykład rozpocznie odtwarzanie pliku dźwiękowego o nazwie pobranej z pola parametru w pliku html, przechowywanego w tym samym katalogu na serwerze co dokument html. Ponieważ kod zawarto w metodzie init() odtwarzanie dźwięku będzie jednorazowe. W celu odtwarzania dźwięku w pętli można wykorzystać interfejs AudioClip. Zdefiniowane w klasie Applet metody getAudioClip() wywoływane tak samo jak metody play(), zwracają obiekt typu AudioClip (nie jawna definicja metod interfejsu). Dostępne metody umożliwiają zarówno jednorazowe odtworzenie dźwięku (play()) jak i odtwarzanie w pętli loop(()) oraz zatrzymanie odtwarzania dźwięku (stop()).

6.6 Java Media API

Pracę z wieloma mediami w Javie w znaczny sposób rozszerzają dodatkowe bibliteki Javy (Extensions) oznaczane przez javax. W ramach projektu Java Media API stworzono różne biblioteki ułatwiające pracę z mediami. Do podstawowych bibliotek należy zaliczyć m.in:

- Java 2D – dwu-wymiarowa grafika i obrazowanie
- Java 3D – trój-wymiarowa grafika i obrazowanie
- Java Advanced Imaging – rozszerzenie Java2D pod względem reprezentacji i przetwarzania obrazów
- Java Media Framework – przechwytywanie i odtwarzanie mediów (audio-wideo) ,
- Java Speech – synteza i rozpoznawanie dźwięku,
- Java Sound – przechwytywanie i odtwarzanie dźwięków.

W wersji JDK 1.3 dostępna jest standardowo biblioteka Java Sound, umożliwiająca nagrywanie i odtwarzanie dźwięków w różnych formatach.