Jacek Matulewski
http://www.phys.uni.torun.pl/~jacek/

JBuilder X: Grafika w Javie

(dodatek do książki
„JBuilder X: Efektywne programowanie w Javie”,
Wydawnictwo Helion, Gliwice 2004)

Toruń – Marburg, 24 lutego 2004

Najnowsza wersja tego dokumentu oraz źródła opisanych apletów znajdują się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/helion/jbuilderx/index.html

Spis treści

Spis treści **2**

Wstęp **2**

Klasa Graphics **2**

Rysowanie linii **3**

Buforowanie rysowania grafiki **10**

Klasa Strzalka **14**

Zegar analogowy **22**

Wstęp

Zasadniczym celem tego rozdziału jest zobrazowanie w jaki sposób, korzystając z możliwości graficznych Javy, wzbogacić interfejs apletu. Zaczniemy od bardzo prostego zadania narysowania serii linii, następnie będziemy je modyfikować i wzbogacać. Na tym przykładzie pokażemy też w jakim celu i w jaki sposób zastosować technikę podwójnego buforowania. Kolejny paragraf pokaże jak tworzyć własne klasy reprezentujące proste konstrukcje graficzne. Przykładem będzie klasa implementująca wektor, w którym jako parametry będziemy podawać punkt zaczepienia, kierunek i długość zamiast standardowych punktów początkowego i końcowego. W niektórych przypadkach znacznie upraszcza to tworzenie grafiki. Uwzględnimy również możliwość narysowania grota na jego końcu.

Zakładam, że czytając ten tekst Czytelnik zaznajomił się książką „JBuilder X. Efektywne programowanie w Javie" pominę więc szczegółowe opisy podanych tam typowych czynności w JBuilderze (tworzenie projektów, klas, itp.) zastępując je odniesieniami do odpowiedniego fragmentu książki.

Klasa Graphics

Wszystkie metody służące do rysowania w Javie umieszczone są w klasie `Graphics`. Metodą tej klasy jest na przykład `drawLine` pozwalająca rysować linie. W tym rozdziale będziemy ją wykorzystywać non stop. Klasa `Graphics` zawiera również metody pozwalające na rysowanie prostych figur geometrycznych (zob. poniższa tabela), metody `drawPolygon` i `drawPolyline` rysujące wielokąty i linie łamane, metody sterujące kolorami, wyświetlające łańcuchy (`drawString`) oraz opisaną w książce metodę `drawImage` służącą do wyświetlania plików graficznych a także wiele innych.

Z apletem zawsze jest związany obiekt typu `Graphics`, którego referencję można pobrać metodą apletu `getGraphics`. Obiekt, do którego się ona odnosi pozwala na rysowanie bezpośrednio na „płótnie" apletu.

Warunkiem trwałości rysunków przygotowanych przez programistę jest ich odświeżanie (np. wówczas, gdy aplet został przesłonięty przez inne okno i ponownie odsłonięty)[1]. Wobec tego dobrą zasadą jest, żeby wszelkie polecenia rysowania na aplecie umieszczać w metodzie `paint`. Mamy wówczas gwarancję, że zostaną wywołane przy każdym odświeżeniu apletu. Sami możemy wymusić odświeżenie wykorzystując bezargumentową metodę `repaint`. Ta metoda wywołuje asynchronicznie metodę apletu `update(Graphics)`, o której więcej w kolejnych paragrafach, a ona dopiero metodę `paint(Graphics)`. Nie należy jawnie wywoływać ani `update`, ani `paint`, a jedynie poprzez `repaint`.

Nazwa metody rysującej kontur	Nazwa metody (z wypełnieniem)	Opis
drawLine(x1,y1,x2,y2)		rysowanie linii od (x1,y1) do (x2,y2)
drawRect(left,top,width,height)	fillRect	prostokąt
drawRoundRect(left,top,width,height,aw,ah)	fillRoundRect	prostokąt o zaokrąglonych rogach
draw3DRect(left,top,width,height,raised)	fill3DRect	prostokąt z brzegiem imitującym 3D
drawOval(left,top,width,height)	fillOval	elipsa
drawArc(left,top,width,height,start,kat)	fillArc	łuk

W metodach `draw...` i `fill...` klasy `Graphics`, poza `drawLine`, pierwsze cztery argumenty oznaczają lewy górny wierzchołek pola zajmowanego przez figurę (left, top) oraz szerokość i wysokość tego pola. Dolny prawy wierzchołek znajduje się więc w punkcie (left+width, top+height).

W metodzie `drawLine`, na której się teraz skupimy, cztery argumenty to współrzędne punktów początku i końca linii.

Rysowanie linii

Ćwiczenie: Zacznijmy od narysowania serii leżących poziomo linii o różnych barwach: od czerni stopniowo będziemy przechodzić do wybranego losowo koloru. W ten sposób uzyskamy bardzo ładny efekt gradientu.

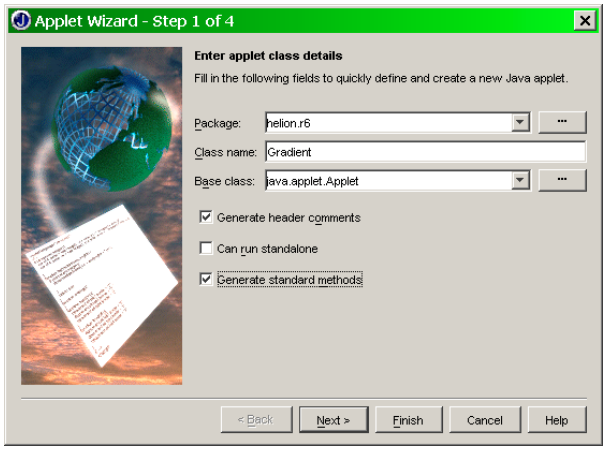
Pakiet `java.awt` zawiera klasę `Color`, która udostępnia obsługę dwóch układów współrzędnych opisujących kolory: RGB i HSB. W RGB kolor określany jest przez składniki czerwony (R od ang. *red*), zielony (G od *green*) i niebieski (B – *blue*), podczas gdy w drugim składnikami są barwa (H od *hue*), nasycenie (S od *saturation*) i jasność (B od *brightness*). Pierwszy lepiej odwzorowuje rzeczywisty sposób tworzenia koloru w kineskopie lub w drukarce, ale drugi jest o wiele bardziej intuicyjny w kontakcie z użytkownikiem. Okna dialogowe wyboru koloru zazwyczaj korzystają z układu HSB lub jednej z jego licznych odmian. Warto zauważyć, że to właśnie z takiego układu korzystamy również, gdy słownie chcemy opisać jakiś kolor.

Do wybrania barwy (składnik H) posłużymy się generatorem liczb pseudolosowych udostępnianym przez metodę `java.lang.Math.random`. Nasycenie koloru będzie stałe, równe wartości maksymalnej (S = 1), a jasność będziemy zmieniać, aby uzyskać płynne przejście między czernią, a wybranym kolorem.

Zabierzmy się zatem za pisanie kodu. Najpierw stwórzmy nowy aplet postępując w sposób opisany w pierwszym rozdziale książki. Najlepiej utworzyć osobny pakiet `helion.xc` (zob. poniższy rysunek). Niech

[1] Sprawa jest nieco bardziej złożona, ale zostanie wyjaśniona w paragrafie dotyczącym buforowania.

klasa apletu nazywa się `Gradient`[2]. Dobrze jest zaznaczyć pole *Generate standard methods*, co spowoduje, że do apletu dodane zostaną opisane w pierwszym rozdziale metody standardowe `start`, `stop` i `destroy`.



Po ustaleniu nazwy pakietu i klasy kończymy działanie kreatora apletu klikając *Finish*.

Dostęp do „płótna” apletu umożliwia metoda o następującej sygnatutrze `public void paint(Graphics g)`. Jak widać argumentem tej metody jest obiektem typu `Graphics` opisywanym na początku tego rozdziału, co umożliwia umieszczanie w niej poleceń rysujących na aplecie. Metoda ta jest wywoływana przez wirtualną maszynę Javy zawsze, gdy konieczne jest odświeżenie apletu. Programista może również wymusić jej wywołanie uruchamiając metodę `repaint`. Jak pisałem wcześniej, nie należy wywoływać metody `paint` bezpośrednio.

Tworzymy szkielet metody, w której umieścimy pozostałe polecenia:

```
public void paint(Graphics g)
  {
  }
```

Następne polecenia umieścimy wewnątrz tej metody.

Naszym zadaniem jest narysowanie zbioru poziomych linii o jednej barwie, ale różnej jasności (od 0 do 1). Należy więc wykorzystać pętlę, która będzie indeksowała piksele w pionie obszaru dostępnego dla użytkownika na powierzchni apletu. Przed pętlą umieścimy instrukcję wybierającą losowo jedną z barw (liczba rzeczywista w zakresie od 0 do 1):

```
float barwa=(float)Math.random();
```

Wygodnie jest też pobrać do zmiennej typu całkowitego wysokość apletu:

```
int wysokosc=this.getBounds().height;
```

[2] Jak wspominałem wcześniej, zgodnie ze standardem twórców Javy, nazwa klasy powinna zaczynać się od dużej litery, co ma odróżniać ją od pakietów oraz metod i własności.

Uwaga! W JBuilderX, który używa JDK 1.4, dostępna jest w klasie `Applet` funkcja `getHeight` zwracająca wysokość apletu w pikselach. Problem polega na tym, że ta, wydawałoby się bardziej naturalna od użytej w powyższym kodzie funkcja, dostępna jest dopiero od JDK 1.2. Nie będzie wobec tego obsługiwana przez wbudowane do przeglądarek wirtualne maszyny Java. Należy bardzo uważać, bo takich funkcji jest sporo i warto często sprawdzać, czy nasz aplet działa w zwykłej przeglądarce.

Pętla będzie wówczas miała następujący szkielet:

```
for(int indeks=0;indeks<wysokosc;indeks++)
      {
      }
```

Należy zwrócić uwagę na to, że początkowa wartość indeksu pętli równa jest 0, a więc ostatnia wartość to `wysokosc-1`. Tak samo numerowane są piksele w obszarze apletu.

Do ustalenia koloru skorzystamy ze statycznej metody `getHSBColor` klasy `Color` przyjmującej jako argumenty kolejno barwę, nasycenie i jasność. Metoda ta tworzy obiekt typu kolor i zwraca jako wartość jego referencję. Metoda `getHSBColor` przyjmuje trzy argumenty typu `float` o wartościach od 0.0 do 1.0. Kolejno rysowane w pętli linie będą różnić się jedynie jasnością, wewnątrz pętli musimy więc wykonać dwa zadania: pierwsze to zmiana bieżącego koloru „pióra", a drugie to wywołanie metody rysującej linię, czyli `Graphics.drawLine`. Wobec tego pętla powinna wyglądać następująco:

```
for(int indeks=0;indeks<wysokosc;indeks++)
  {
    float jasnosc=indeks/(float)(wysokosc-1);
    Color kolor = Color.getHSBColor(barwa, 1, jasnosc);
    g.setColor(kolor);
    g.drawLine(0,indeks,this.getBounds().width,indeks);
  }
```

Pierwsze polecenie w pętli określa jasność na podstawie zmiennej indeksującej pętlę w taki sposób, żeby uzyskać wartość 0 w pierwszej iteracji pętli i wartość maksymalną, równą 1 dla ostatniej. W następnej linii barwa ustalona przed pętlą i zmienna `jasnosc` wykorzystywane są do stworzenia obiektu klasy `Color`. Drugi argument metody `getHSBColor`, czyli nasycenie jest stałe i równe 1. Następnie metodą `Graphics.setColor` (w naszym przypadku odpowiedni obiekt podany przez głowę metody nazywa się `g`) ustalamy kolor, w jakim rysowane są linie (a także wszystkie inne prymitywy graficzne korzystające z tego samego obiektu `g`). Jej argumentem jest oczywiście obiekt klasy `Color`, a więc korzystamy z obiektu, który stworzyliśmy w poprzednim poleceniu.

Powyżej opisane polecenia można „skompresować" do jednej linii:

```
g.setColor(Color.getHSBColor(barwa,1,indeks/(float)(wysokosc-1)));
```

Do samego rysowania linii służy metodą `drawLine` z klasy `Graphics`. Przyjmuje ona cztery argumenty: dwa pierwsze określają współrzędne początku, dwa kolejne współrzędne końca linii. Układ współrzędnych jest nieco inny niż ten, do którego jesteśmy przyzwyczajeni na lekcjach matematyki, ale typowy dla grafiki komputerowej. Punkt (0,0) znajduje się bowiem w lewym górnym rogu apletu, a współrzędne y rosną w miarę przesuwania się w dół.

Cała metoda powinna wyglądać następująco:

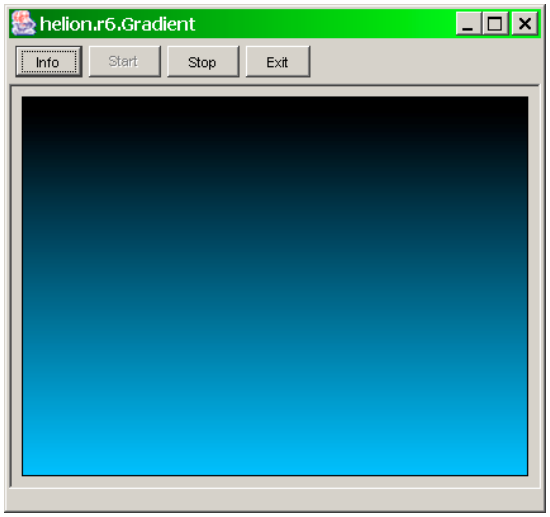
```
public void paint(Graphics g)
  {
    float barwa = (float) Math.random();
    int wysokosc = this.getBounds().height;

    for (int indeks = 0; indeks < wysokosc; indeks++)
      {
      float jasnosc = indeks / (float) (wysokosc - 1); //ustalenie jasnosci
      Color kolor = Color.getHSBColor(barwa, 1, jasnosc); //stworzenie koloru
      g.setColor(kolor); //ustalenie koloru w jakim rysujemy
      //g.setColor(Color.getHSBColor(barwa,1,indeks/(float)(wysokosc-1)));
      g.drawLine(0, indeks, this.getBounds().width, indeks); //narysowanie linii
      }
  }
```

W kreatorze apletu zaznaczyliśmy opcję *Generate standard methods*, co spowodowało dodanie do kodu klasy bezargumentowych metod publicznych `start`, `stop` i `destroy`. Ostatnią można usunąć, a w `start` i `stop` umieśćmy polecenie wymuszające odmalowanie apletu `repaint()`, a więc:

```
public void start()
  {
    repaint();
  }
```

W efekcie w wewnętrznej przeglądarce, po każdym zatrzymaniu i uruchomieniu apletu za pomocą przycisków *Start* i *Stop*, lub w Internet Explorerze po naciśnięciu przycisku *Odśwież*, nasz gradient jasności zmieni barwę.



Ćwiczenie: Zmodyfikować aplet z poprzedniego ćwiczenia w taki sposób, aby uzyskać efekt tęczy.

Zadanie okazuje się być niezwykle prostym ponieważ wystarczy jedynie drobna modyfikacja w kodzie napisanym w poprzednim ćwiczeniu. Zamiast jasności należy teraz uzależnić od indeksu pętli pierwszy argument w metodzie `getHSBColor`, a więc barwę. Powinniśmy w ten sposób uzyskać gamę wszystkich barw przypominającą tęczę.

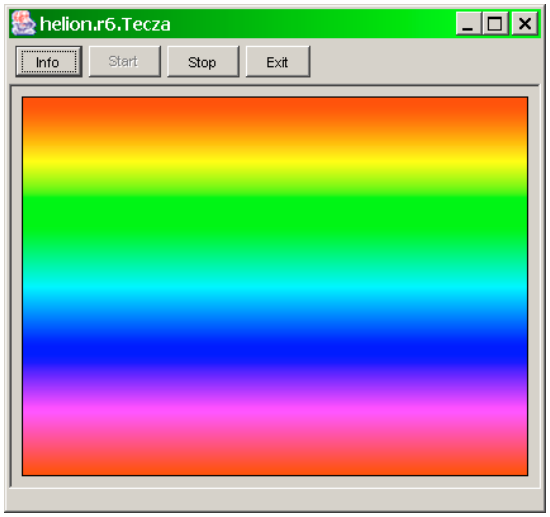
Ponieważ barwa nie będzie już ustalana losowo usuwamy pierwszą linię metody `paint`. Zmieniamy także inne polecenia wewnątrz pętli (zobacz wyróżnione linie w poniższym listingu):

```
public void paint(Graphics g)
  {
  int wysokosc = this.getBounds().height;
  for(int indeks=0;indeks<wysokosc;indeks++)
    {
      float barwa=indeks/(float)(wysokosc-1);
      Color kolor=Color.getHSBColor(barwa,1,1);
      g.setColor(kolor);
      g.drawLine(0,indeks,this.getBounds().width,indeks);
    }
  }
```

Zaznaczone linie można ponownie zapisać zwięźlej

```
g.setColor(Color.getHSBColor(indeks/(float)(wysokosc-1),1,1));
```

Podobnie jak wcześniej jasność, teraz barwę ustalamy na podstawie indeksu pętli. W tworzeniu obiektu `kolor` wykorzystujemy zmienną `barwa`, a nasycenie i jasność ustalamy równe 1. Nie zmienia się polecenie rysowania linii.



Ćwiczenie: Przygotować aplet analogiczny do opisywanego w pierwszym ćwiczeniu, z tą różnicą, że zamiast losowania barwy należy pozwolić na ustalenie jej za pomocą parametru przekazywanego z kodu HTML.

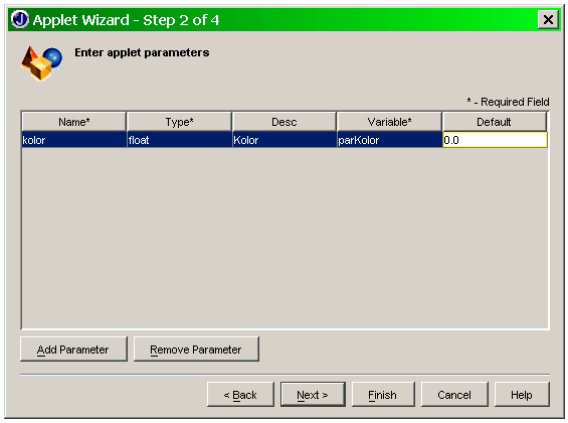
Bardzo często wykorzystywanym mechanizmem sterowania działaniem apletu jest opisane w pierwszym rozdziale książki przekazywanie parametrów z kodu HTML do apletu. Teraz użyjemy tego mechanizmu jako sposobu na poinformowanie apletu jaką barwę ma użyć do tworzenia gradientu.

Stwórzmy nowy aplet o nazwie `GradientParametr` za pomocą kreatora apletu. Tym razem nie kończmy działania kreatora na pierwszym kroku, a przejdźmy do drugiego i dodajmy do listy pobieranych parametrów nowy o nazwie (kolumna *Name* w kreatorze) `kolor` typu (*Type*) `float` z dowolnym opisem (*Desc*). Wartość odczytanego parametru niech będzie przekazana do zmiennej (kolumna *Variable*) `parKolor`. Wartość domyślna (*Default*) musi być ustalona w zakresie od 0.0 do 1.0. Na zrzucie ekranu widoczny jest przykład, w którym wartość domyślna ustalona jest na 0.0, czyli na barwę czerwoną.

JBuilder doda do klasy apletu deklarację zmiennej `parKolor` typu `float`, a w metodzie `init` umieści następującą linię

```
parKolor = Float.valueOf(this.getParameter("kolor", "0.0")).floatValue();
```

która odczyta wartość parametru i zapisze jego wartość w zmiennej `parKolor`.



W trzecim kroku kreatora ustalmy nazwę pliku HTML dodawanego do projektu, który pozwoli na podgląd apletu w dowolnej przeglądarce (najlepiej nadawać nazwę taką samą jak nazwa apletu tj. `GradientParametr`).

Z kodu umieszczonego w pliku *Gradient.java*, apletu z pierwszego ćwiczenia (powinno być nadal dostępne w bieżącym projekcie) kopiujemy metodę `paint` i wklejamy do nowej klasy.

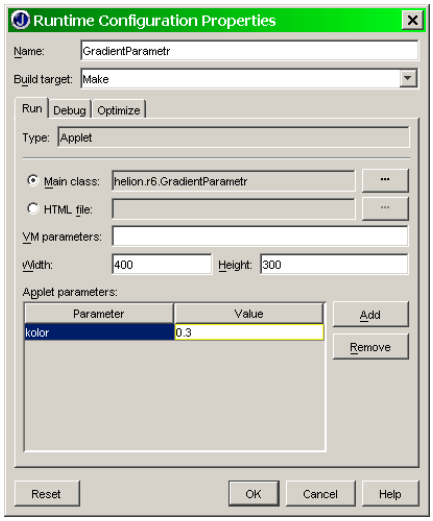
Należy uważać, żeby nie wkleić jej za definicją klasy lub do wnętrza którejś z metod. Najlepiej umieścić ją przed ostatnim zamykającym nawiasem klamrowym.

Teraz należy ją nieco zmodyfikować, zastępując polecenie ustalające barwę losowo poleceniem przepisującym wartość ze zmiennej `parKolor`.

```
public void paint(Graphics g)
 {
```

```
float barwa=parKolor;
int wysokosc = this.getBounds().height;
for(int indeks=0;indeks<wysokosc;indeks++)
    {
    float jasnosc=indeks/(float)(wysokosc-1);
    Color kolor=Color.getHSBColor(barwa,1,jasnosc);
    g.setColor(kolor);
    g.drawLine(0,indeks,this.getBounds().width,indeks);
    }
}
```

W ten sposób nasz kod stał się czuły na parametr `kolor`. Można go przetestować albo z wbudowanej przeglądarki apletów (osobne okno, które pojawia się domyślnie po kompilacji), albo z dokumentu HTML przygotowanego przez kreator apletu (z wnętrza JBuildera lub np. z Interner Explorera). Najpierw zajmijmy się pierwszym sposobem. Jeżeli chcemy, aby JBuilder wymuszał przekazywanie parametru przez wbudowaną przeglądarkę przejdźmy do *Project*, *Project Properties*, *Run* lub *Run*, *Configurations* (obie drogi zaprowadzą nas w to samo miejsce), a następnie przycisk *Edit*, *Add* w sekcji *Applet Parameters*. Pojawi się nowa linia, w której możemy podać nazwę parametru (w naszym przypadku musi to być „kolor") i jego wartość (np. 0.3, co odpowiada zieleni) tak, jak na poniższym rysunku.



Jeżeli chcemy parametr podać przez kod HTML należy między znacznikiem `<applet ...>`, który zazwyczaj zajmuje kilka linii, a znacznikiem domykającym `</applet>` dodać linię definiującą parametr:

```
<param name = "kolor" value = "0.15">
```

Zapewne linia podobna do tej jest już obecna w kodzie dokumentu HTML – zadba o to kreator apletu.

Teraz wystarczy kliknąć na plik *GradientParametr.html* w oknie projektu i obejrzeć efekt. Równie dobrze można wczytać go do zewnętrznej przeglądarki.

Buforowanie rysowania grafiki

Zagadnienie podwójnego buforowania jest omówione niemal w każdej książce poświęconej programowaniu w języku Java, więc tutaj jedynie je zasygnalizujemy jako konieczny element wiedzy dotyczącej grafiki każdego programisty Java.

Mówiąc w skrócie: buforowanie polega na zastąpieniu rysowania „bezpośrednio na ekranie”[3] przez przygotowanie rysunku poza ekranem i następnie szybkiego skopiowania go na ekran. Dzięki temu eliminuje się efekt migotania związany z czyszczeniem „płótna” apletu i ponownym wypełnianiem[4].

Ćwiczenie: Kolejne ćwiczenie ma uzmysłowić konieczność buforowania: do apletu stworzonego w pierwszym ćwiczeniu dodamy pasek przewijania zmieniający barwę rysowanych linii.

Ta modyfikacja jest niezwykle prosta. Należy na aplecie umieścić komponent `java.awt.Scrollbar` z palety *AWT*[5]. Następnie w metodzie `paint` losowy wybór barwy zastępujemy przez wykorzystanie pozycji odczytanej z nowo utworzonego obiektu `scrollbar1`:

```
public void paint(Graphics g)
  {
  float barwa=(float)scrollbar1.getValue()/90;
  int wysokosc = this.getBounds().height;
  for(int indeks=0;indeks<wysokosc;indeks++)
      {
      float jasnosc=indeks/(float)(wysokosc-1);
      Color kolor=Color.getHSBColor(barwa,1,jasnosc);
      g.setColor(kolor);
      g.drawLine(0,indeks,this.getBounds().width,indeks);
      }
  }
```

W wyróżnionej linii określamy barwę na podstawie pozycji paska przewijania, dla własności `scrollbar1.minimum` (dostęp przez okno własności) ustalonej na 0, a `maximum` na 100. Dziwną własnością tego komponentu jest to, że maksymalna wartość pobierana przez `getValue` jest mniejsza od określonej jako maksymalna o `BlockIncrement` (tzn. maksymalna wartość przy przesuwaniu paska myszką wynosi 90 a nie 100).

Poruszanie paskiem jednak nadal nie będzie przynosić zmian koloru narysowanych linii; należy wymusić ich ponowne narysowanie po każdej zmianie pozycji paska regulującego ich barwę. Wobec tego konieczne jest obsłużenie zdarzenia `adjustmentValueChanged` obiektu `scrollbar1`. W tym celu przechodzimy do

---

[3] Jest to oczywiście jedynie skrót myślowy, bo chodzi o pamięć ekranu.

[4] Nie zawsze czyszczenie całego obszaru apletu jest rozwiązaniem optymalnym, np. w przypadku gdy animujemy tylko fragment rysunku (nieruchomy Donald poruszający tylko rękoma).

[5] Należy pamiętać o zmianie własności `layout` apletu na `null` jeżeli chcemy dowolnie ustalić jego rozmiar i pozycję.

zakładki *Design* i w oknie inspektora wybieramy zakładkę *Events*. Teraz wystarczy kliknąć na odpowiednią pozycję[6] i w powstałej w ten sposób metodzie wywołać metodę `repaint` naszego apletu.

Po tych modyfikacjach zmiana pozycji suwaka powoduje wywołanie metody `repaint`, która z kolei wywołuje metodę `paint`, a ta ustala kolor na podstawie pozycji paska przewijania.

W końcu uzyskaliśmy pożądany efekt zmiany barw, a przy okazji jasna stała się potrzeba wykorzystania techniki buforowania – częste odświeżanie powierzchni apletu powoduje uciążliwe dla wzroku migotanie.

Cała prawda o wywoływaniu paint

Wywołanie metody `repaint` przez użytkownika lub wirtualną maszynę Javy wywołuje `paint`, ale nie bezpośrednio, a poprzez metodę `update`. Metoda `update` przyjmuje w argumencie obiekt typu `Graphics`, który zostanie przesłany do `paint`. To daje możliwość, żeby zoptymalizować wywołanie `paint` lub wprowadzić podwójne buforowanie bez konieczności modyfikacji rysowania w samej metodzie `paint`.

Pierwotna postać metody `update` w JDK 1.4 jest następująca:

```
public void update(Graphics g)
{
  if (isShowing())
  {
    if (! (peer instanceof LightweightPeer))
    {
      g.clearRect(0, 0, width, height);
    }
    paint(g);
  }
}
```

Pierwszy warunek `if` sprawdza, czy okno apletu jest widoczne na ekranie (czy znajduje się w obszarze pokazywanym na monitorze i czy nie jest przesłonięty przez inne okna). Kolejny warunek sprawdza, czy mamy do czynienia z tzw. lekkim (z ang. *lightweight*), czy z ciężkim (z ang. *heavyweight*) komponentem[7]. Ciężkie komponenty wykorzystują komponenty wizualne ze środowiska graficznego platformy, w której uruchomiony został aplet. Taki komponent, w przeciwieństwie do komponentów lekkich, musi zostać „zmazany” przy odświeżaniu. Do tego służy polecenie `clearRect` w powyższym listingu. Ostatnie polecenie wywołuje metodę `paint` z podanym w argumencie metody `update` obiektem `g` typu `Graphics`.

---

[6] Procedura tworzenia metod zdarzeniowych opisana jest szczegółowo w rozdziale trzecim.

[7] Zobacz informacje na ten temat w opisie projektu `JPictureView` w książce.

Ćwiczenie: Stworzyć samodzielnie metodę update, która odtwarza działanie oryginalnej metody java.applet.Applet.update

Ponieważ skupiamy się na wykorzystaniu biblioteki *AWT* w naszej metodzie pominiemy warunek sprawdzenia „lekkości" komponentów. Dla wygody dalszej rozbudowy tej metody zadeklaruję zmienne typu `int`, w których będę przechowywał wysokość i szerokość apletu. Następnie dodajemy polecenia czyszczenia całej powierzchni apletu i ponownego rysowania jego zawartości:

```
public void update(Graphics g)
  {
    if (this.isShowing()) {
      Rectangle dim=this.getBounds();
      int w=dim.width, h=dim.height;
      g.clearRect(0, 0, w, h);
      paint(g);
    }
  }
```

Ćwiczenie: Umieścić mechanizm buforowania w metodzie update.

Potrzebny będzie nam dodatkowy obiekt typu `Graphics` na którym będziemy rysować przed skopiowaniem jego zawartości na ekran (czyt. do odpowiedniego obiektu `Graphics`). Problem polega na tym, że `Graphics` jest klasą abstrakcyjną, a więc nie można stworzyć obiektu tego typu za pomocą operatora `new`. Najprostsze rozwiązanie to stworzyć obiekt typu `Image` i z niego pobrać odpowiednią metodą referencję typu `Graphics`.

Do klasy apletu dodajemy więc następującą deklarację

```
private Image bufor=null;
```

Być może bardziej elegancko ze względu na zakres dostępności byłoby deklarować `bufor` wewnątrz metody `update`, ale jego wielokrotne tworzenie przy częstym wywoływaniu metody `paint` zauważalnie zwalnia działanie apletu.

Powyższe polecenie deklaruje jedynie referencję do typu `Image`. Niestety nie możemy w tej samej linii stworzyć odpowiedniego obiektu, gdyż konieczna do tego jest znajomość rozmiaru apletu. Umieścimy więc polecenie stworzenia obiektu w metodzie `update` z zastrzeżeniem, żeby tworzony był tylko raz, przy pierwszym jej uruchomieniu. Jak to zrobić? To proste, wystarczy tylko sprawdzić, czy `bufor` ma jeszcze wartość `null`. Po stworzeniu obiektu zmienna `bufor` będzie przechowywać jego adres, a więc wystarczy wykorzystać ten fakt pisząc:

```
if (bufor==null) bufor=this.createImage(w,h);
```

W efekcie polecenie `createImage` będzie wywołane tylko raz, przy pierwszym wykonaniu metody `paint`.

Cała metoda powinna wyglądać następująco:

```
public void update(Graphics g)
{
  Rectangle dim=this.getBounds();
```

```
    int w=dim.width, h=dim.height;

    if (!this.isShowing()) return;

    if (bufor==null) bufor=this.createImage(w,h);

    Graphics bg=bufor.getGraphics(); //pobieram Graphics zwiazane z buforem
    bg.clearRect(0,0,w,h); //czyszczenie bufora
    bg.setClip(0,0,w,h); //Ustalenie polozenia i rozmiaru (u nas calosc)
    this.paint(bg); //tu maluje do bufora
    g.drawImage(bufor,0,0,null); //maluje bufor na aplecie

    bg.dispose(); //zwalniam bufor
  }
```

Najistotniejsza w powyższym kodzie jest wyróżniona grupa poleceń. Pierwsze z nich przypisuje zmiennej `bg` obiekt typu `Graphics` uzyskany z obiektu `bufor`. Obiekt ten umożliwi malowanie na obiekcie `bufor` (klasy `Image`). Następnie bufora jest czyszczony metodą `Graphics.clearRect`. W kolejnej linii ustalamy fragment bufora, który może być zamalowany. Przy użytych tu argumentach nie powoduje to żadnej zmiany (wykorzystany jest cały jego obszar), ale polecenie to umieszczone zostało w kodzie, aby wskazać właściwe miejsce na ewentualną modyfikację odświeżanego obszaru.

Wreszcie najważniejsze jest wywołanie metody `paint` naszego apletu. Inaczej niż w oryginalnej wersji jako argumentu nie używamy obiektu `g` związanego z apletem, ale nasz `bufor`, a dokładnie związany z nim obiekt `bg`. Następnie zawartość bufora kopiowana jest na „płótno" apletu. Korzyść z tej metody bierze się z różnicy szybkości wykonania metody `paint` i `drawImage`. W pierwszym przypadku obraz jest budowany polecenie po poleceniu na oczach użytkownika, w drugim budowany jest poza ekranem, a następnie szybko kopiowany do pamięci ekranu.

Ostatnie polecenie (`Graphics.dispose`) nie usuwa obiektu bufora, a jedynie zwalnia wykorzystywane czasowo przez niego zasoby systemu, co pozwala na lepsze gospodarowanie pamięcią przez wirtualną maszynę Javy.

Ćwiczenie: Zbuduj klasę BufferedApplet rozszerzającą klasę java.applet.Applet, w której znajdzie się obsługa buforowania.

Należy postępować według następującego algorytmu:

1) Za pomocą odpowiedniego kreatora dodajemy do projektu i pakietu `helion.xc` nową klasę o nazwie `BufferedApplet` rozszerzającą klasę `java.applet.Applet`.
2) Umieszczamy w niej przygotowaną w poprzednim paragrafie metodę `update`. Należy pamiętać, aby w klasach, które będą dziedziczyły z klasy `BufferedApplet` nie było metody `update`, gdyż przesłoniłaby ona metodę umieszczoną w `BufferedApplet` lub żeby tą metodę wywoływały.
3) Wystarczy teraz w aplecie, w którym chcemy zastosować technikę buforowania zastąpić klasę bazową `Applet` przez `BufferedApplet`. W przypadku ostatniego przykładu należy również usunąć napisaną przez nas metodę `update` (identyczną z umieszczoną w `BufferedApplet`).

Uwaga! Nową klasę możemy używać swobodnie jako zamiennik standardowej klasy java.applet.Applet. Można również zdefiniować tę klasę jako rozszerzenie klasy javax.swing.JApplet, jeżeli chcemy pisać aplety z interfejsem korzystającym z pakietu Swing.

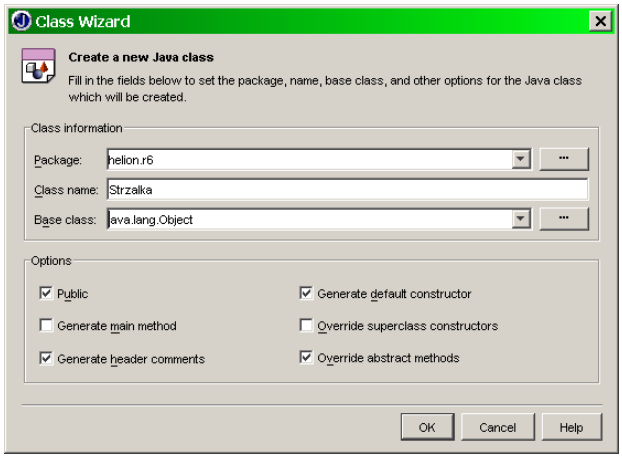
Klasa Strzalka

Bardzo często wygodniej jest rysować linie podając punkt początkowy oraz długość i kierunek zamiast współrzędnych punktu końcowego. Czasem wygodnie jest również przechowywać informacje o takiej linii, żeby np. w kolejnym kroku animacji jedynie zmienić jej kierunek, bez konieczności ustalania pozostałych parametrów. I to właśnie zrobimy w kolejnym ćwiczeniu dodając również możliwość dorysowania grota., aby stworzyć elegancką klasę implementującą rysowanie strzałki.

Napisanie klasy, która będzie odpowiadała za przygotowanie i rysowanie strzałki to zadanie tego rozdziału, ale jego prawdziwym celem jest pokazanie w jaki sposób posługiwać się klasami wspomagającymi tworzenie grafiki.

Ćwiczenie: Zbudujmy publiczną klasę należącą do pakietu helion.xc o nazwie Strzalka bez konstruktora domyślnego. Niech klasa zawiera własności prywatne określające punkt zaczepienia fxpocz i fypocz, długość fdlg i kierunek fkat oraz wartość logiczną fgrot decydującą o tym, czy należy rysować grot wektora czy jedynie główną linię. Dostęp do tych własności możliwy będzie jedynie poprzez grupę funkcji ustalających wartość set... tych własności oraz funkcje pobierające odpowiednie informacje get.... Niech klasa posiada konstruktor umożliwiający ustalenie wszystkich własności.

Do stworzenia nowej klasy najwygodniej wykorzystać jak zwykle kreator klas (menu *File*, *New Class...*). W polu *Package* wybieramy nazwę pakietu (w naszym przypadku `helion.xc`), pole *Class name* wypełniamy pisząc nazwę klasy, która będzie jednocześnie rdzeniem nazwy pliku (`Strzalka`). Ani nazwa klasy, ani nazwa pakietu nie może zawierać polskich liter. Przypominam, że klasa pisana w Javie zawsze rozszerza inną klasę. Jeżeli w jej sygnaturze nie ma jawnego użycia słowa `extends` – niejawnie rozszerza klasę `java.lang.Object`. Zatem możemy do pola *Base class* wpisać tę właśnie klasę. Ponadto należy zaznaczyć pola *Public*, żeby zagwarantować dostęp do klasy zarówno z pakietu jak i z poza niego. Można również zaznaczyć pole *Generate default constructor*, nie po to, żeby klasę obdarować domyślnym konstruktorem, ale dlatego, że łatwo będzie przerobić go na konstruktor, którego potrzebujemy – wystarczy tylko uzupełnić go o argumenty.



Na początku klasy zadeklarujmy wszystkie potrzebne własności (przyjęliśmy tu konwencję, w której prywatne zmienne wewnętrzne poprzedzamy literką „f”):

```
private int fxpocz,fypocz;
private int fxkonc=0,fykonc=0;
private int fdlg,fkat=0; //kat w stopniach
private double fkat_rad=0; //kat w radianach
private boolean fgrot;
```

Są to kolejno: współrzędne punktów początkowego i końcowego, długość i kierunek. Użytkownik będzie podawał kierunek w stopniach ograniczonych do liczb całkowitych. Tracimy w ten sposób dokładność, funkcje trygonometryczne w klasie `java.lang.Math` posługują się kątami w radianach przechowywanych w zmiennych `double`, jednak nawet dla długiego wektora różnica położenia punktu końcowego dla różnicy kierunków wynoszącej jeden stopień jest zaniedbywalna.

Zanim przejdziemy do pisania konstruktora zdefiniujmy serię publicznych metod pozwalających na zmianę powyższych własności. Pierwsza z nich będzie określała punkt zaczepienia:

```
public void setPunktZaczepienia(int xpocz,int ypocz)
{
  fxpocz=xpocz;
  fypocz=ypocz;
}
```

Typowym zagadnieniem w tym kontekście jest automatyczna aktualizacja wartości niektórych własności klasy – w naszym przypadku zmiennych przechowujących położenie końca wektora. Najlepiej jest napisać metodę, która będzie obliczać te wartości, co pozwoli na uniknięcie powtarzania kodu wielokrotnie w każdej metodzie `set...` . Oczywiście powinna to być metoda prywatna, np.

```
private void obliczPozycjeKonca()
{
  fxkonc=fxpocz+(int)(fdlg*Math.cos(fkat_rad));
  fykonc=fypocz+(int)(fdlg*Math.sin(fkat_rad));
}
```

Należy jej wywołanie dodać do `setPunktZaczepienia`.

```
public void setPunktZaczepienia(int xpocz,int ypocz)
{
  fxpocz=xpocz;
  fypocz=ypocz;
  obliczPozycjeKonca();
}
```

Analogicznie piszemy dwie kolejne metody:

```
public void setDlugosc(int dlg)
{
```

```
  fdlg=dlg;
  obliczPozycjeKonca();
}
```

oraz

```
public void setKat(int kat)
{
  fkat=kat-90;
  fkat_rad=toRadians(fkat); //kat w radianach od pionu
  obliczPozycjeKonca();
}
```

Jeszcze jedna rzecz wymaga komentarza: we współrzędnych biegunowych kąt oblicza się od osi *x* leżącej poziomo. Tutaj przyjmujemy bardziej intuicyjną w grafice konwencję, w której kąt 0 oznacza strzałkę skierowaną pionowo w górę. Oznacza to konieczność odjęcia 90 stopni od kąta określającego kierunek wektora w metodzie `setKat`.

Pozostaje jeszcze metoda ustalająca wartość zmiennej `fgrot`, która nie wymaga obliczania pozycji punktu końcowego:

```
public void setGrot(boolean grot)
{
  fgrot=grot;
}
```

Warto zauważyć, że w metodzie `setKat` nie korzystamy z metody `Math.toRadians`, która została wprowadzona do klasy `Math` dopiero od wersji JDK 1.2. Ponieważ wykorzystanie tej funkcji wyklucza wykorzystanie wbudowanej do przeglądarek wirtualnej maszyny Javy i zmusza do ściągnięcia plug-inu, lepiej jest napisać ją samodzielnie. Szczególnie, że nie jest zbyt skomplikowana (podana poniżej wartość jest dokładną kopią definicji z klasy `Math`, którą można obejrzeć korzystając z polecenia *Find Definition* z menu kontekstowego) przy zaznaczonej oryginalnej metodzie `Math.toRadians`. Do naszej klasy dodajemy więc:

```
public static double toRadians(int angdeg)
{
  return angdeg / 180.0 * Math.PI;
}
```

Teraz możemy napisać konstruktor

```
public Strzalka(int xpocz,int ypocz,int dlg,int kat, boolean agrot)
{
  setPunktZaczepienia(xpocz,ypocz);
  setDlugosc(dlg);
```

```
    setKat(kat);
    setGrot(agrot);
  }
```

Należy zauważyć pewną wadę takiego rozwiązania. Otóż każda, poza ostatnią, z metod wywołanych w konstruktorze wywołuje metodę obliczającą punkt końcowy. To powoduje jej zbędne uruchomienia, ale jest konieczne jeżeli chcemy, żeby metody z grupy `set...` były publiczne. Na szczęście w naszym przypadku koszt wykonania metody `obliczPozycjeKonca` jest znikomy.

Zauważmy, że nasza klasa nie ma domyślnego konstruktora. I to jest bardzo dobre rozwiązanie, ponieważ nie ma w ten sposób ryzyka, że obiekt zostanie zainicjowany przez programistę bez świadomego zainicjowania własności. Jeżeli jednak chcemy taki konstruktor dodać, wówczas należy pamiętać, żeby z niego wywołać metody `set...` z domyślnymi wartościami w argumentach.

Warto dodać do klasy metody pozwalające na pobranie informacji o obliczonym końcu strzałki. Np.:

```
public int getXKonc()
{
  return fxkonc;
}

public int getYKonc()
{
  return fykonc;
}
```

Teraz pozostaje przygotować metodę rysującą wektor.

Ćwiczenie: Przygotować metodę public void rysuj(Graphics) rysującą na podanym w argumencie obiekcie typu Graphics strzałkę zgodnie z przechowywanymi w obiekcie parametrami[8]

Narysowanie głównej linii nie jest oczywiście problemem.

```
public void rysuj(Graphics g)
{
  g.drawLine(fxpocz,fypocz,fxkonc,fykonc);
}
```

Należy pamiętać, żeby zaimportować klasę `java.awt.Graphics` poleceniem `import java.awt.Graphics`.

Trochę kłopotu może sprawić rysowanie grota wektora. Jego wygląd będzie kontrolowany przez kilka stałych określających kąt rozchylenia, to, czy grot ma być proporcjonalny do długości wektora czy też ma mieć stałą

[8] Ten i następny paragraf można rozwijać równolegle, ponieważ wygodne jest budowanie klasy rysującej z możliwością zobaczenia efektu.

długość oraz współczynnik skrócenia w pierwszym przypadku i długość w drugim. Nawet jeżeli nie chcemy udostępnić kontroli nad tymi stałymi użytkownikowi klasy, dobrze jest tworzyć stałe zamiast umieszczać liczby bezpośrednio w poleceniach `drawLine`. Ułatwia to łatwiejszą modyfikacje i kontrolę kodu. Jeżeli nie będą one udostępniane – można je zdefiniować wewnątrz metody `rysuj`, a nawet w obrębie zakresu określanego przez sprawdzenie warunku, czy grot ma być rysowany. My udostępnimy te stałe poprzez funkcję `setGrotWlasnosci`, więc musimy je zdefiniować jako własności prywatne klasy. Dodajmy zatem do klasy następujące własności:

```
private int fgrot_rozchylenie=20; //w stopniach
private boolean fgrot_proporcjonalny=false;
private double fgrot_skrocenie=5;
private int fgrot_dlugosc=15;
```

Kierunek dwóch linii grota jest obliczany na podstawie kierunku wektora za pomocą wzoru: $\varphi = \phi + 180^\circ \pm 20^\circ$, gdzie ϕ jest kierunkiem głównej linii, a dodanie kąta półpełnego powoduje zmianę kierunku na przeciwny. Rozchylenie linii grota równe jest 20 stopni, co określane jest przez stałą `grot_rozchylenie`. Wewnątrz metody `rysuj` dodajmy deklaracje zmiennych:

```
double grot_roz_rad=toRadians(grot_rozchylenie);
double grotkat_rad=fkat_rad+Math.PI+grot_roz_rad; //prawa linia grota
```

Długość grota obliczana jest następującym poleceniem:

```
int grotdlg=grot_proporcjonalny?fdlg/grot_skrocenie:grot_dlugosc;
```

Skorzystaliśmy tutaj z poznanego w drugim rozdziale książki operatora *warunek*?*prawda*:*fałsz*. Wynika z tego, że jeżeli stała `grot_proporcjonalny` ustawiona jest na `true`, długość grota będzie pięć razy krótsza od długości wektora (`grot_skrocenie` wynosi 5), a w przeciwnym przypadku będzie równa 15, czyli wartości stałej `grot_dlugosc`.

Teraz możemy obliczyć współrzędne końca prawej linii grota:

```
int xgrotkonc=fxkonc+(int)(grotdlg*Math.cos(grotkat_rad));
int ygrotkonc=fykonc+(int)(grotdlg*Math.sin(grotkat_rad));
```

i narysować ją:

```
g.drawLine(fxkonc,fykonc,xgrotkonc,ygrotkonc);
```

Analogicznie postępujemy dla lewej linii:

```
grotkat_rad=fkat_rad+Math.PI-grot_roz_rad; //lewa linia grota
xgrotkonc=fxkonc+(int)(grotdlg*Math.cos(grotkat_rad));
ygrotkonc=fykonc+(int)(grotdlg*Math.sin(grotkat_rad));
```

```
g.drawLine(fxkonc,fykonc,xgrotkonc,ygrotkonc);
```

Cała metoda `rysuj` powinna wyglądać następująco:

```
public void rysuj(Graphics g)
{
  //glowna linia
  g.drawLine(fxpocz,fypocz,fxkonc,fykonc);

  if (fgrot) //grot
  {
    double grot_roz_rad=toRadians(grot_rozchylenie);
    double grotkat_rad=fkat_rad+Math.PI+grot_roz_rad;
    //prawa linia
    int grotdlg=grot_proporcjonalny?fdlg/grot_skrocenie:grot_dlugosc;
    int xgrotkonc=fxkonc+(int)(grotdlg*Math.cos(grotkat_rad));
    int ygrotkonc=fykonc+(int)(grotdlg*Math.sin(grotkat_rad));
    g.drawLine(fxkonc,fykonc,xgrotkonc,ygrotkonc);
    //lewa linia
    grotkat_rad=fkat_rad+Math.PI-grot_roz_rad;
    xgrotkonc=fxkonc+(int)(grotdlg*Math.cos(grotkat_rad));
    ygrotkonc=fykonc+(int)(grotdlg*Math.sin(grotkat_rad));
    g.drawLine(fxkonc,fykonc,xgrotkonc,ygrotkonc);
  }
}
```

Aby umożliwić programiście korzystającemu z klasy `Strzalka` zmianę własności grota możemy dodać do klasy następującą metodę publiczną:

```
public void setGrotWlasnosci(int rozchylenie,boolean proporcjonalny,
                             float skrocenie,int dlugosc)
{
  fgrot_rozchylenie=rozchylenie;
  fgrot_proporcjonalny=proporcjonalny;
  fgrot_skrocenie=skrocenie;
  fgrot_dlugosc=dlugosc;
}
```

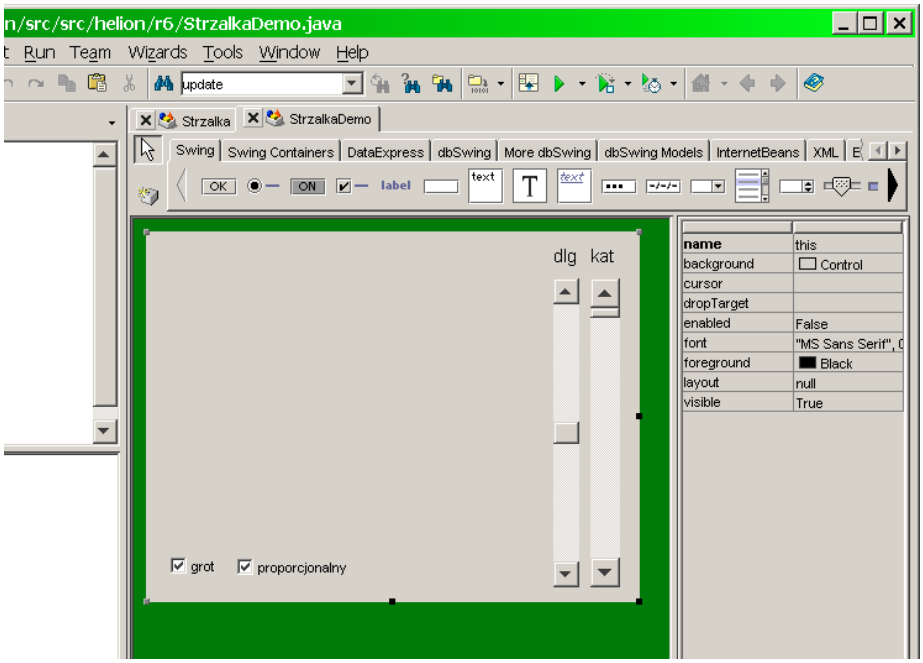
Można rozważyć dodanie jej wywołania do konstruktora.

Ćwiczenie: Stwórz aplet demonstrujący działanie klasy Strzalka (obrót, zmiana długości)

Aby obejrzeć efekt metody `Strzalka.rysuj` stworzymy prosty aplet testujący postępując według następującego planu.

1) Tworzymy nowy aplet (w sposób opisany w rozdziale pierwszym książki). Jako klasę bazową można podać zwykły `java.applet.Applet` lub `BufferedApplet` z naszego pakietu.

2) W widoku projektowania umieszczamy na aplecie dwa paski przewijania z zakładki AWT (`java.awt.Scrollbars`) oraz dwa pola opcji (`java.awt.Checkbox`). Za pomocą pasków przewijania będziemy modyfikować długość i kierunek wektora, a pola opcji będą określać czy rysujemy grot na końcu wektora i czy jest on proporcjonalny do długości wektora.



3) Dodajemy do klasy apletu własność typu `Strzalka` o nazwie `s` poleceniem `Strzalka s = null;`.

4) Tworzymy metodę `paint` i przy pierwszym jej uruchomieniu tworzymy obiekt `s`. Następnie rysujemy go na aplecie. Na aplecie drukujemy również długość i kierunek strzałki korzystając z metody `Graphics.drawString`. Jej pierwszym argumentem jest wyświetlany łańcuch, a dwa kolejne określają pozycje w pikselach:

```
public void paint(Graphics g)
{
  super.paint(g);

  //opisy
  g.drawString("dlg: "+scrollbar1.getValue(),10,15);
  g.drawString("kat: "+scrollbar2.getValue(),10,30);

  //strzalka
  if (s==null) //pierwszy raz
  {
    s = new Strzalka(this.getBounds().width/2, this.getBounds().height/2,
                     scrollbar1.getValue(),scrollbar2.getValue(),
                     checkbox1.getState());
    s.setGrotWlasnosci(20,checkbox2.getState(),5,15);
  }

  s.rysuj(g);
}
```

5) Teraz pozostaje zareagować na zmiany pozycji pasków przewijania i stanu pól opcji. Tworzymy metody zdarzeniowe w sposób opisany w pierwszym rozdziale, w których modyfikujemy odpowiednie własności obiektu `s` i odświeżamy aplet metodą `repaint`:

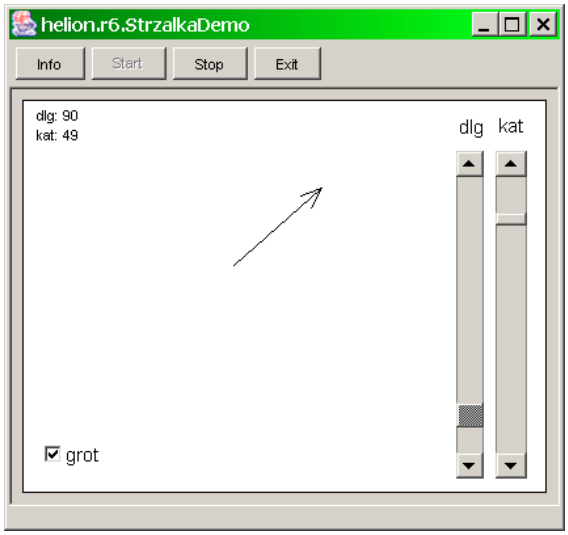
```
void scrollbar1_adjustmentValueChanged(AdjustmentEvent e)
{
  s.setDlugosc(scrollbar1.getValue());
  repaint();
}

void scrollbar2_adjustmentValueChanged(AdjustmentEvent e)
{
  s.setKat(scrollbar2.getValue());
  repaint();
}

void checkbox1_itemStateChanged(ItemEvent e)
{
  s.setGrot(checkbox1.getState());
  repaint();
}

void checkbox2_itemStateChanged(ItemEvent e)
{
  s.setGrotWlasnosci(20,checkbox2.getState(),5,15);
  repaint();
}
```

I gotowe.



Ćwiczenie: Pakiet AWT zawiera klasę Point, która służy do przechowywania informacji o punkcie. Wykorzystajmy ją jako alternatywny argument metody setPunktZaczepienia oraz alternatywną wartość zwracaną przez getPunktKonca.

Przeciążymy teraz metody ustalające parametry w taki sposób, żeby wykorzystać klasę `java.awt.Point` (należy ją oczywiście zaimportować w nagłówku pliku *Strzalka.java*).

```
public void setPunktZaczepienia(Point pocz)
{
  fxpocz=pocz.x;
  fypocz=pocz.y;
  obliczPozycjeKonca();
};
```

Stwórzmy również metodę pobierającą punkt położenia końca wektora.

```
public Point getPunktKonca()
{
  return new Point(fxkonc,fykonc);
}
```

Zegar analogowy

Aby zaprezentować działanie klasy Strzalka w bardziej praktycznym zadaniu wykorzystamy ją do rysowania wskazówek zegara. W tym paragrafie wyjątkowo odstąpimy od ogólnego założenia unikania Java 2. Aplet, który napiszemy, nie będzie więc mógł być oglądany przez dowolną przeglądarkę WWW, a wymagał będzie odpowiedniego plug-inu (co najmniej JDK 1.2).

Przy programowaniu zegara analogowego trzeba rozwiązać trzy problemy. Pierwszym jest cykliczne wywoływanie metody aktualizującej zegar (zagadnienie typowe także dla animacji), drugim pobranie aktualnego czasu i trzecim rysowanie wskazówek zegara. Ostatni problem rozwiąże wykorzystanie klasy `Strzalka` dla każdej wskazówki. Pozostałe problemy też są do pokonania korzystając z zalet Java 2 (kosztem niekompatybilności z wbudowanymi VJM z przeglądarek).

Ćwiczenie: Korzystając z klasy javax.swing.Timer zbudować mechanizm zegara, który co sekundę będzie odświeżał aplet metodą repaint.

Problem periodycznego wywołania żądanej metody rozwiązujemy za pomocą klasy `Timer`[9]. Mamy dwie klasy o tej nazwie do wyboru: z pakietu AWT i z pakietu Swing, ale należy pamiętać, że żadna z nich nie jest obecna w JDK 1.1. Tu wykorzystamy `javax.swing.Timer`.

```
private Timer timer=new Timer(1000,
                              new ActionListener()
                              {
                                public void actionPerformed(ActionEvent evt)
                                    {
                                    repaint();
                                    }
                              });
```

Rozszyfrujmy ten kod. Tworzymy obiekt o nazwie `timer` typu `Timer`. Konstruktor tej klasy przyjmuje dwa argumenty: pierwszy to odstęp czasu między kolejnymi „impulsami” wysyłanymi do zdefiniowanego w drugim argumencie obiektu nasłuchującego typu `ActionListener`. Odstęp ustalamy na 1000 milisekund, czyli jedną sekundę. Klasa `ActionListener` jest abstrakcyjna; wymaga zdefiniowania metody `actionPerformed`. Należy więc zdefiniować nową klasę rozszerzającą klasę `ActionListener`, w której zdefiniowana jest ta

[9] Można ten problem także rozwiązać nie wykorzystując Java 2, a korzystając z osobnego wątku do odmierzania czasu jak w omówionym w książce aplecie `PuzzleNaCzas`.

metoda, a następnie stworzyć obiekt tej klasy i podać jako drugi argument konstruktora klasy `Timer`. Java dopuszcza składnię upraszczającą te czynności przez ułatwienia w tworzeniu rozszerzeń klas. Wystarczy za nazwą klasy bazowej napisać, oczywiście w nawiasach klamrowych, definicję nowych metod. W powyższym kodzie definiujemy w taki sposób klasę zawierającą publiczną metodę `actionPerformed` w miejscu tworzenia obiektu (klasa anonimowa, zagadnienie to zostało szerzej omówione w drugim rozdziale książki), której zadaniem jest wywołanie metody `repaint` apletu.

Końcowy efekt jest taki, że nasz aplet odświeżany jest co sekundę. Wystarczy teraz, żebyśmy w metodzie `paint` umieścili polecenia rysujące zegar, a będzie miarowo „tykał" i zmieniał położenie wskazówek co jedną sekundę.

Ćwiczenie: Zdefiniuj metodę paint, w której należy narysować tarczę zegara

Zdefiniujmy metodę `paint` (jak zwykle musi być publiczna i nie może zwracać wartości). Najpierw przygotowujemy informacje o możliwym rozmiarze tarczy zegara (dopasowujemy go do rozmiaru apletu) oraz tworzymy obiekty kolorów zegara:

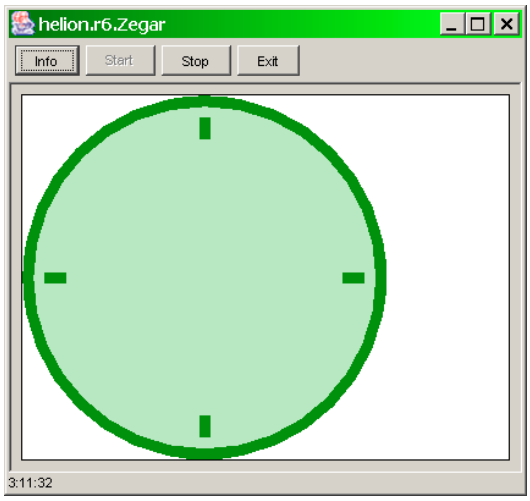
```
public void paint(Graphics g)
{
  int rozmiar=Math.min(this.getBounds().width,this.getBounds().height);
  int szer=rozmiar/30;
  int wysgodz=2*szer;
  Color kolorBrzegu=new Color(0,150,0);
  Color kolorTla=new Color(192,235,192);
}
```

Następnie rysujemy tarczę zegara. Ze względu na prostotę kodu zrezygnowałem z wszelkiej optymalizacji, m.in. rysowana co sekundę jest cała tarcza:

```
g.setColor(kolorBrzegu);
g.fillOval(0,0,rozmiar,rozmiar);
g.setColor(kolorTla);
g.fillOval(szer,szer,rozmiar-2*szer,rozmiar-2*szer);
```

(Te i poniższe polecenia umieszczamy oczywiście także w metodzie `paint`.) Na tarczy rysujemy cztery prostokąty odpowiadające godzinom 3, 6, 9 i 12.

```
g.setColor(kolorBrzegu);
g.fillRect(rozmiar/2-szer/2,2*szer,szer,wysgodz); //12
g.fillRect(rozmiar/2-szer/2,rozmiar-4*szer,szer,wysgodz); //6
g.fillRect(2*szer,rozmiar/2-szer/2,wysgodz,szer); //9
g.fillRect(rozmiar-4*szer,rozmiar/2-szer/2,wysgodz,szer); //3
```



Ćwiczenie: Dodaj do metody paint rysowanie wskazówek pod właściwymi dla aktualnej godziny kątami

Pozostaje odczytanie aktualnej godziny. Służy do tego klasa `java.util.GregorianCalendar` (konieczne jest oczywiście zaimportowanie jej lub całego pakietu `java.util`). Jej obsługa jest dość nietypowa, ponieważ zamiast udostępnić kilka oddzielnych metod do pobrania godziny, minuty, sekundy itd. posiada tylko jedną metodę `get`, w której przez argument wskazujemy na interesującą nas wielkość. W ten sposób, po stworzeniu obiektu trzeba wywołać tę metodę kilka razy z różnymi argumentami (odpowiednie stałe identyfikujące są zdefiniowane jako własności statyczne klasy):

```
GregorianCalendar cal = new GregorianCalendar();

int godzina=cal.get(cal.HOUR);
int minuta=cal.get(cal.MINUTE);
int sekunda=cal.get(cal.SECOND);
int milisekunda=cal.get(cal.MILLISECOND);

showStatus(""+godzina+":"+minuta+":"+sekunda);
```

Po skopiowaniu tych poleceń do metody `paint` zobaczymy na pasku stanu, dzięki wywołaniu metody `showStatus` aktualną godzinę aktualizowaną co sekundę. Następnie obliczamy kąty, pod jakimi powinny być rysowane wskazówki:

```
int godzina_kat=(int)(360.0*(godzina+minuta/60.0)/12.0);
int minuta_kat=(int)(360*minuta/60);
int sekunda_kat=(int)(360*(sekunda+milisekunda/1000.0)/60);
```

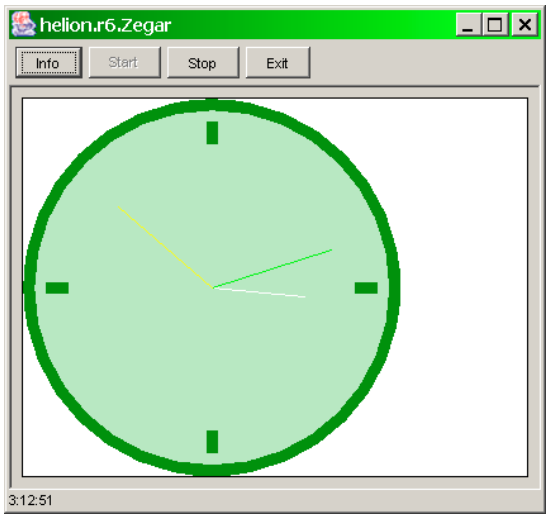
Warto zauważyć, że uwzględniamy ilość milisekund w przypadku obliczania kąta dla wskazówki sekundowej na wypadek zmniejszenia kroku czasowego w obiekcie `Timer`. Jest to konieczne dla jej płynnego ruchu w takim przypadku. Odpowiednio uwzględniamy też ilość minut przy obliczaniu kąta wskazówki godzinowej, ale pomijamy sekundy we wskazówce minutowej. W ten sposób uzyskamy efekt przeskakiwania tej wskazówki co minutę.

Teraz zdefiniujemy trzy obiekty typu strzałka o początkach w środku tarczy (x = `rozmiar/2`, y = `rozmiar/2`) i długościach (trzeci argument) równych jednej czwartej dla godziny i jednej trzeciej dla pozostałych. Wszystkie będą rysowane bez grota (ostatni argument ustawiony na `false`). Rozmiar wskazówki minutowej i sekundowej będzie identyczny, równy jednej trzeciej rozmiaru zegara, wskazówka godzinna będzie krótsza.

```
Strzalka wsk_godzina=new Strzalka(rozmiar/2,rozmiar/2,rozmiar/4,godzina_kat,false);
Strzalka wsk_minuta =new Strzalka(rozmiar/2,rozmiar/2,rozmiar/3,minuta_kat,false);
Strzalka wsk_sekunda=new Strzalka(rozmiar/2,rozmiar/2,rozmiar/3,sekunda_kat,false);
```

Teraz można narysować wskazówki zmieniając kolory odpowiednie dla każdej z nich, np.:

```
g.setColor(Color.white);wsk_godzina.rysuj(g);
g.setColor(Color.green);wsk_minuta.rysuj(g);
g.setColor(Color.yellow);wsk_sekunda.rysuj(g);
```



Ćwiczenie: Korzystając z klasy Java 2 Graphics2D narysować wskazówki o większej szerokości

Jak widać na rysunku wskazówki naszego zegara są zbyt cienkie. I niestety nie można ich pogrubić (chyba, że rysując kilka obok siebie) bez korzystania z klas umieszczonych dopiero w pakietach Java 2. Ponieważ do uruchomienia apletu zegara i tak konieczne jest JDK zgodne z Java 2 ze względu na klasę `Timer`, więc możemy nadal korzystać z udogodnień Java 2, a dokładniej z klasy `Graphics2D`, bez dalszej straty.

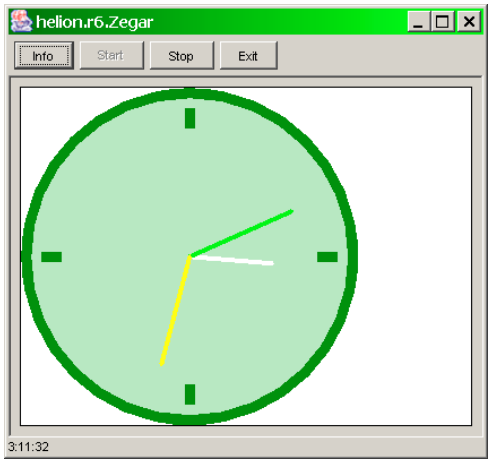
Klasa `Graphics2D`, podobnie jak `Graphics`, jest abstrakcyjna. Nie można więc stworzyć obiektu jej typu. Wystarczy jednak gdy zrzutujemy na ten typ obiekt typu `Graphics` podawany w metodzie `paint` przez głowę (czyli przez argument), aby uzyskać dostęp do nowych metod.

Z ich pomocą ustalmy grubość na równą 1/60 rozmiaru zegara, ale nie mniejszą niż 4 piksele. Następnie korzystając z metody `Graphics2D.setStroke` ustalamy parametry rysowania linii. Jej argumentem jest obiekt typu `Stroke`, a ponieważ jest on abstrakcyjny tworzymy obiekt klasy `BasicStroke`, pochodnej względem `Stroke`, ale już nie abstrakcyjnej. Wybieramy konstruktor trójargumentowy pozwalający ustalić grubość linii, sposób rysowania końców (zaokrąglone, ostre, prostokątne itp.) i sposób rysowania łączenia odcinków w liniach łamanych. Do ustalenia odpowiednich parametrów służą zdefiniowane stałe statyczne. Ponieważ użyjemy

zaokrąglonych końców jako drugi argument używamy `BasicStroke.CAP_ROUND`, a jako trzeci `BasicStroke.JOIN_ROUND`. Uzupełnienie metody `paint` powinno więc wyglądać następująco:

```
Graphics2D g2=(Graphics2D) g;
int grubosc=(int)(rozmiar/60.0);
int min_grubosc=4;
g2.setStroke(new BasicStroke(grubosc>min_grubosc?grubosc:min_grubosc,
                             BasicStroke.CAP_ROUND,
                             BasicStroke.JOIN_ROUND));
```

Te polecenia należy oczywiście wstawić przed wywołaniem metod `rysuj` wskazówek zegara.



Zakres wiedzy dotyczącej grafiki w Java jest oczywiście znacznie znacznie szerszy niż problemy poruszone w tym dodatku. Odsyłam chcącego dowiedzieć się więcej na ten temat Czytelnika do dokumentacji JDK dodanej do JBuildera oraz przede wszystkim do oficjalnego poradnika Java ze strony Sun (http://java.sun.com/tutorial/). Moim celem było przekazanie Czytelnikowi podstawowych intuicji pozostawiając do samodzielnego studiowania poznawanie poszczególnych klas i metod.