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

Tworzenie apletów Java za pomocą JBuilder Personal

Ćwiczenia

Toruń, 20 marca 2003

Najnowsza wersja tego dokumentu znajduje się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad5.pdf

Źródła opisanych w tym dokumencie programów znajdują się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad5.zip

Działające aplety omówione na tej stronie znajdują się na stronie
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad5/

Spis treści

Wstęp

Co to jest Java?
Java to nowy język zorientowany obiektowo stworzony w firmie Sun, który ma kilka charakterystycznych własności:
- pełna przenaszalność na poziomie źródeł i kodu wynikowego (różnice sprzętowe ominięto przez wprowadzenie pośredniej między kodem a sprzętem wirtualnej maszyny Java (JVM od *Java Virtual Machine*).
- konsekwentna obiektowość; nie można tworzyć zmiennych lub funkcji nie będących własnościami lub metodami jakiegoś obiektu. Wszystkie obiekty tworzone są dynamicznie, użytkownik ma do nich dostęp wyłącznie przez referencje (nie ma wskaźników). JVM sam dba o ich usuwanie z pamięci (nie ma destruktorów).
- konsekwentna „wyjątkowość" – metoda przesyłająca wyjątek musi to zadeklarować i nie może być wywoływana przez przechwytywania wyjątków.

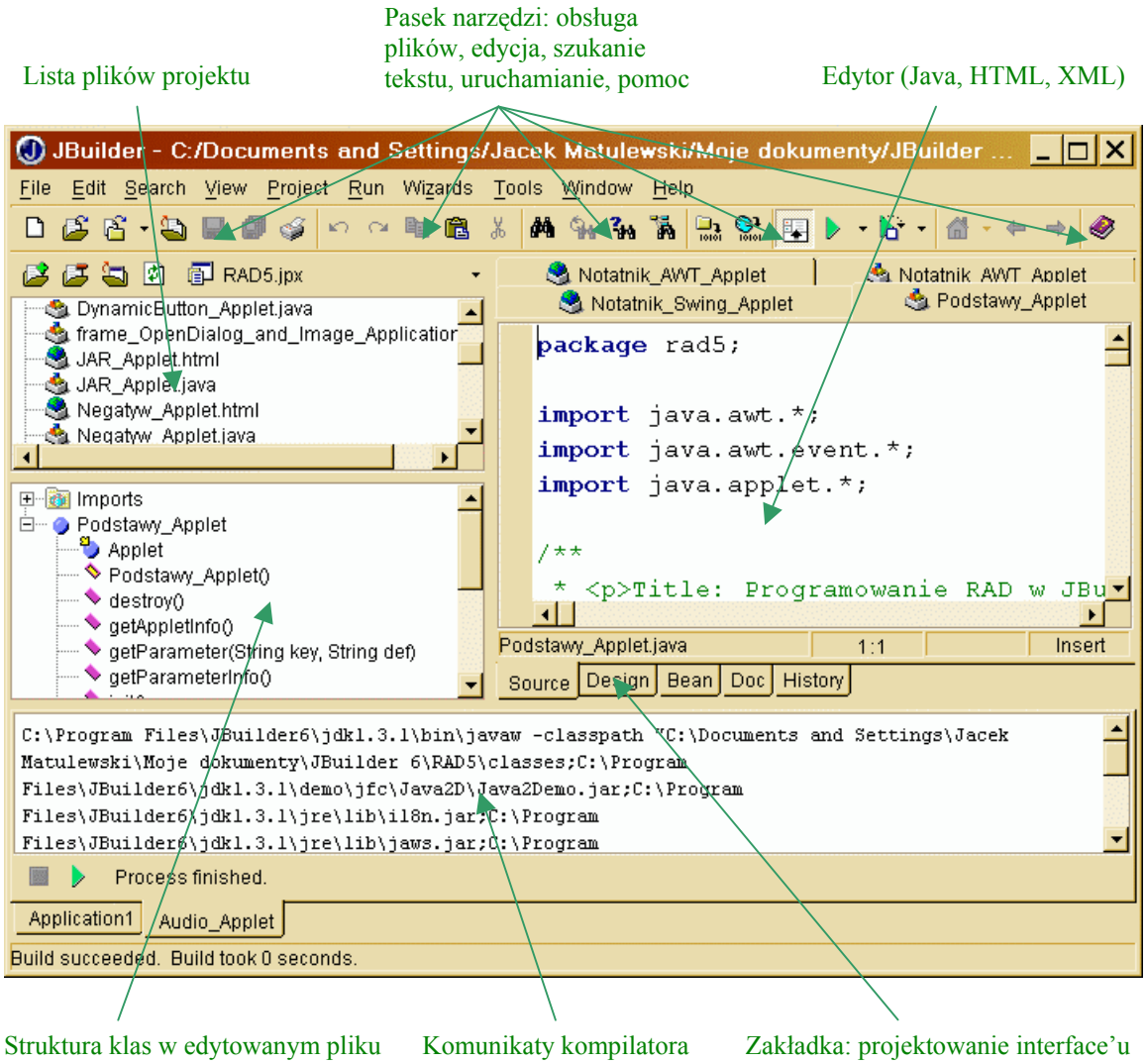
Co to jest JVM?
Wirtualna maszyna Javy to oprogramowanie dostarczane przez Sun pozwalające na uruchamianie skompilowanych apletów w przeglądarkach WWW oraz samodzielnych aplikacji. JVM jest dostarczana w pakiecie JRE (*Java Runtime Environment*) dla użytkowników oraz wspólnie z kompilatorem Javy w pakiecie JDK (*Java Development Kit*) przeznaczonym dla programistów.

Co to jest JBuilder?
Borland JBuilder jest środowiskiem wspierającym programowanie w języku Java dostępnym na platformach MS Windows, Linux/Unix (wymagane jest środowisko graficzne) oraz Macintosh. JBuilder pozwala na kompilację i debugowanie apletów i aplikacji napisanych w Javie, a w wersjach Professional i Enterprise dostarcza także dodatkowych bibliotek. JBuilder pozwala na programowanie w stylu RAD (ang. *Rapid Application Development*) tzn. pozwala na tworzenie graficznego interface'u z obiektów Javy za pomocą myszki (wybieranie komponentów, konfigurowanie ich za pomocą okienka i edytorów własności oraz tworzenie szkieletów metod zdarzeniowych), wspiera zarządzanie plikami w obrębie projektu oraz oczywiście dostarcza edytor współpracujący z debuggerem.
JBuilder jest dostarczany w trzech wersjach: darmowa wersja Personal (ten skrypt bazuje na tej wersji), Professional i Enterprise. Wersje różnią się możliwościami (ilość dodatkowych bibliotek oraz narzędzi) oraz oczywiście ceną.

Gdzie zdobyć informacje na temat Javy?
- Pierwszym miejscem, gdzie należy szukać informacji są pliki pomocy dostarczane razem z JBuilderem i JDK.
- Najlepszym źródłem jest strona utrzymywana i rozwijana przez Sun java.sun.com zawierająca bardzo dobrze opracowane bogato ilustrowane przykładami pliki pomocy i skrypty. Skrypt można ściągnąć w całości w jednym pliku zip.
- Kolejnym godnym polecenia miejscem na sieci jest strona darmowej książki *Thinking In Java* Bruce'a Eckela (można też kupić wersję drukowaną, także w wersji polskojęzycznej).
  Oficjalny mirror tej strony: http://www.phys.uni.torun.pl/~jacek/docs/tij2/Frontmatter.html.

Jak korzystać z JBuildera?

Elementy zintegrowanego środowiska programistycznego (IDE)

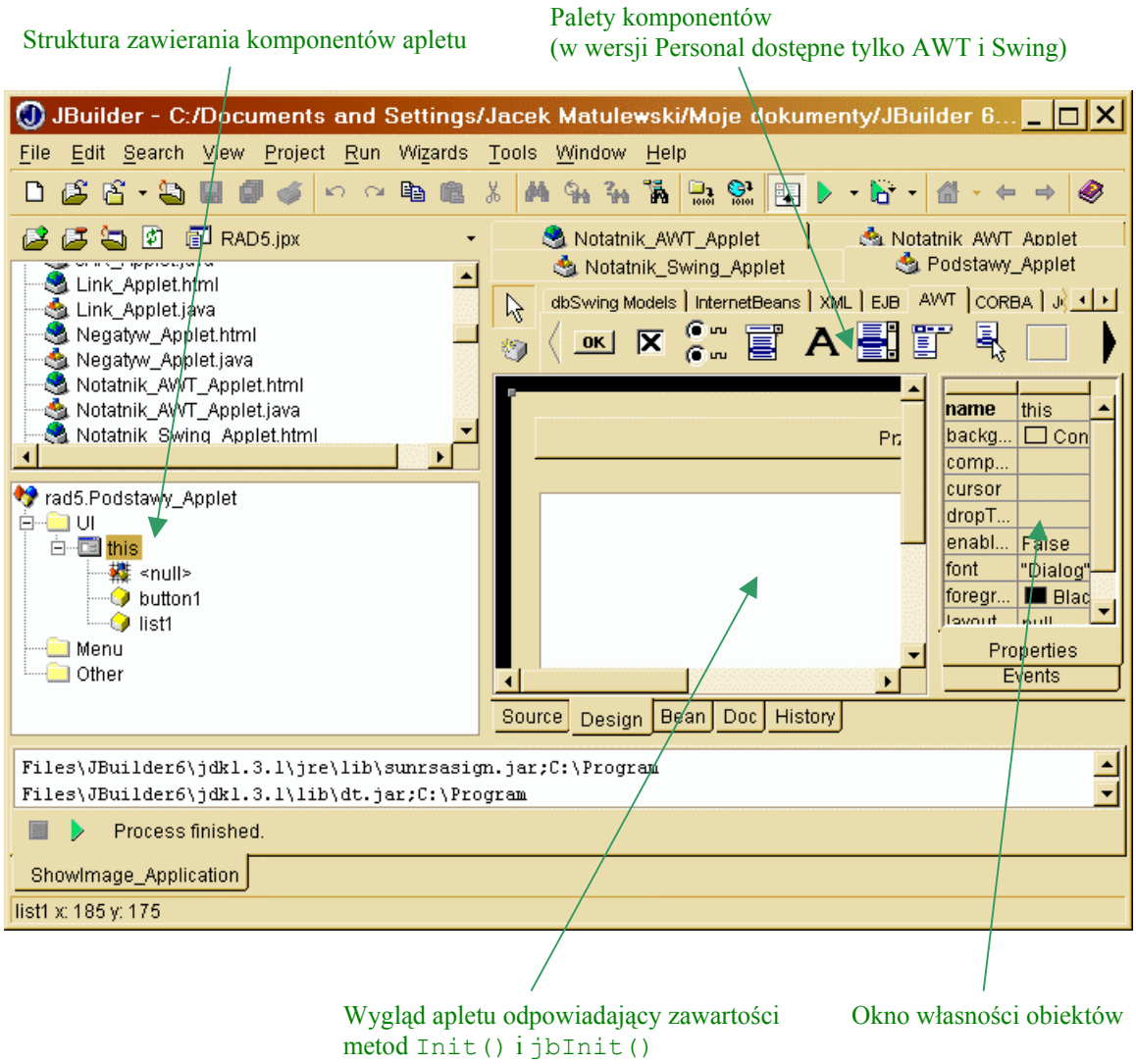


Debugowanie
Projekt można uruchomić w trybie debugowania (Shift+F9). Warto wcześniej ustawić breakpoint (F5) przed blokiem kodu, który podejrzewamy o błąd. W trakcie debugowania działają klawisze typowe dla środowisk programistycznych Borlanda: F4 – wykonuje program do linii kodu, w której znajduje się kursor (run to cursor), F7 – wykonanie następnego polecenia (z wejściem do wywoływanych funkcji), F8 – wykonanie następnej linii kodu i Ctrl+F2 – zamknięcie debugowanej aplikacji.

W trybie debugowania pojawia się nowy panel debugowania, który umożliwia śledzenie wartości zmiennych (watches), sprawdzenia jakie biblioteki zostały załadowane do pamięci, śledzenie aktualnej pozycji w strukturze apletu i oczywiście śledzenie komunikatów debbugera.

Projektowanie RAD

JBuilder umożliwia projektowanie wyglądu apletów i aplikacji w sposób analogiczny jak w Delphi, C++ Builderze czy Kylixie. Można wybrać obiekt wizualny z palet komponentów (w wersji Personal dostępne są tylko palety zawierające obiekty z pakietów java.awt.* oraz javax.swing.*, w wersjach rozbudowanych dostępne są również pakiety stworzone przez Borlanda i innych firm) i umieścić na aplecie. Jego własności można modyfikować za pomocą okna analogicznego do Object Inspectora w Delphi/C++ Builderze/Kylixie). Dwukrotne kliknięcie na zdarzenie spowoduje stworzenie odpowiedniej metody zdarzeniowej.



I. Projektowanie obiektowe w Javie

Informacje nt. projektowania klas w Javie: http://www.phys.uni.torun.pl/~jacek/dydaktyka/klasy.html
Informacje nt. podstawowych instrukcji sterujących: http://www.phys.uni.torun.pl/~jacek/dydaktyka/prog.html

II. Podstawy projektowania apletów – komponenty sterujące[1]

Aplet jest to program, który napisany jest w taki sposób, że może być uruchamiany i kontrolowany przez inną aplikację, w kontekście której został uruchomiony. Aplety Java są uruchamiane najczęściej w przeglądarkach WWW (IE lub Netscape). Przeglądarka steruje apletem wywołując kilka standardowych funkcji (a dokładniej metod, bo z punktu widzenia Javy aplet jest po prostu projektowaną przez programistę klasą dziedziczącą z klasy `Applet` (z pakietu `java.awt.*`)):

`init()`	ta metoda wywoływana jest przez przeglądarkę jako pierwsza, uruchamiana jest przy ładowaniu strony, gdy inicjowany jest zawarty na niej aplet. Metoda wykorzystywana do „zbudowania" interface'u graficznego apletu, w niej znajdują zazwyczaj się polecenia umieszczające na stronie komponenty. W JBuilderze elementy budujące stronę (uwzględniające opcje z Object Inspectora) przeniesione są do funkcji `jbInit()`, która wywoływana jest z `init()`.
`start()`	metoda wywoływana zaraz po `init()` oraz wówczas gdy użytkownik wraca do uprzednio załadowanej strony WWW (okno przeglądarki z załadowanym apletem staje się aktywne). Metoda ta jest wykorzystywana zazwyczaj wówczas gdy aplet ma wykonać jakieś działania niezależnie od zdarzeń kreowanych przez użytkownika. W przeciwnym przypadku, gdy aplet jest „interaktywny", w zasadzie nie ma potrzeby jej wykorzystywania.
`stop()`	metoda wywoływana przez przeglądarkę gdy użytkownik opuszcza stronę WWW. **Uwaga!** Wywołanie metody `stop()` nie oznacza, że aplet lub jego wątki są zatrzymywane, ale daje to taką możliwość programiście.
`destroy()`	metoda wywołana gdy aplet jest zamykany
`paint(Graphics)`	wywoływana asynchronicznie przez wirtualną maszynę Javy gdy zachodzi konieczność „odmalowania" graficznej reprezentacji apletu. Można wymusić jej uruchomienie wywołując metodę `Applet.repaint();`.[2]. W tej metodzie należy umieścić wszystkie polecenia rysujące na „powierzchni" apletu (np. `drawImage`, `drawLine`), aby zapewnić im trwałość – inaczej znikną przy pierwszym odświeżeniu apletu.

Metody te mogą, choć nie muszą być zdefiniowane przez użytkownika (należy pamiętać o odpowiednim zdefiniowaniu argumentów i zwracanych wartości)[3]. Wszystkie te metody powinny być publiczne (tzn. widziane także poza klasą), powinny zwracać „pustą" wartość **`void`** (tzn. deklaracja metody `init()` powinna być następująca: **`public void`** `init()`). Dodatkowo metoda `main`() powinna być zadeklarowana jako **`static`**[4].

Wszystkie elementy języka Java poza zmiennymi prostych typów i tablicami są obiektami z bogatych pakietów będących elementem specyfikacji Java. Również graficzne elementy interface'u umieszczone przez użytkownika na aplecie są obiektami. Obiekty i zmienne mogą być deklarowane zarówno globalnie w całej klasie, jak i lokalnie w poszczególnych metodach na zasadach podobnych jak w C++. Zwyczajowo umieszcza się deklaracje obiektów na początku klasy, a ustalanie ich własności i umieszczanie na formie[5] (poleceniem `Applet.add()`) w funkcji `init()`.

---

[1] W tym rozdziale znajduje się wiele porównań z innymi środowiskami programistycznymi Borlanda, ale ich znajomość nie jest konieczna do jego zrozumienia.

[2] Dokładnie metoda `Applet.repaint()` wywołuje asynchronicznie (czyt. w najdogodniejszym dla siebie momencie) metodę `Applet.update()`, która dopiero wywołuje zdefiniowaną przez nas metodę `Applet.paint()`.

[3] Poniżej w pierwszym przykładzie przetestujemy działanie tych funkcji.

[4] **`static`**, obecne także w C++, oznacze, że metoda ta jest własnością klasy, a nie obiektu, tzn. może być wywołana bez stworzenia obiektu (`Podstawy_Applet.main()`).

[5] Będę dość swobodnie nazywał okno apletu; m.in. na wzór biblioteki VCL i CLX mówiąc o nim forma.

Komponenty wizualne mogą pochodzić z dwóch bibliotek AWT (ang. *Abstract Windowing Toolkit*) i Swing, które należą do standardu Java (to jest zupełnie inna sytuacja niż np. w C++ Builderze, gdzie na standard C++ nakłada się bibliotekę VCL lub CLX, która zawiera komponenty wizualne). Wybór między komponentami Swing i AWT przy tworzeniu umieszczanych na sieci apletów jest, jak na razie, prosty. Nawet najnowsza wersja Internet Explorera i Netscape'a bez zainstalowanych plug-inów Java 2 wspiera jedynie AWT. Z kolei Sun uznaje bibliotekę AWT za projekt zamknięty i nie będzie już rozwijana. Promowaną przez Sun biblioteką kontrolek jest Swing, który jest pakietem znacznie bardziej rozbudowanym[6] i doskonale nadaje się do budowania samodzielnie działających aplikacji. Można również pokusić się o zaprojektowanie w niej apletu, ale należy pamiętać, że będzie to wymagało od użytkownika ściągnięcia i zainstalowania J2RE (ang. *Java 2 Runtime Environment* ze strony http://java.sun.com/getjava/).

AWT to biblioteka zawierająca elementy sterujące dostępne w postaci graficznej (przyciski, pola opcji (checkboxy), pola wyboru (radiobuttony), okienka edycyjne, rozwijalne listy (comboboxy), paski przewijania pola tekstowe itp.). Korzystając z Borland JBuildera możemy projektować aplety i aplikacje korzystające z AWT i Swing w stylu RAD podobnie jak w C++ Builderze i Delphi. Zaznaczamy odpowiedni komponent i umieszczamy go na formie, zmieniamy własności, oprogramowujemy zdarzenia (istnieje tu okienko analogiczne do Object Inspectora) itd.

1. Tworzenie projektu/pakietu

Java wymusza umieszczanie klas (w tym apletów) w pakietach. JBuilder otacza ponadto pakiety projektem, który zwalnia programistę o dbanie o szczegóły, zarządza plikami i ułatwia debugowanie.kodu i modyfikowanie zawartości pakietu. Pierwszym krokiem musi więc być stworzenie projektu, które jest niezwykle proste dzięki trójetapowemu kreatorowi.

W pierwszym kroku określamy
a) nazwę pakietu (powiedzmy **rad5**),
b) typ pliku projektu jpx/jpr. Nie wnikając w szczegóły typ jpx jest łatwiejszy do obsługi przy większej ilości programistów zajmujących się jednym projektem. W pozostałych przypadkach należy wybrać jpr.
c) wybór katalogu, w którym znajduje się projekt
d) można wybrać szablom projektu spośród uprzednio tworzonych lub dostarczonych z JBuilderem. My wybieramy **(Default project)**.

W drugim kroku wybieramy:
a) wersję JDK (jako domyślna pojawi się wersja dostarczona razem z JBuilderem),
b) podkatalogi na kod wynikowy (domyślnie podkatalog classes), kopie źródeł i katalog roboczy. Wybór katalogu roboczego jest ważny przy testowanie apletów (np. określanie względnej ścieżki do pliku na podstawie katalogu roboczego). Domyślnym katalogiem roboczym jest katalog projektu.

W ostatnim kroku wybieramy
a) stronę kodową (zestaw znaków powyżej kodu 127 ASCII),
b) tytuł projektu, opis, numer wersji, nazwisko autora i jego firmy.

2. Kreator apletu, projektowanie RAD

Z menu File, New... wybieramy Applet (położenie tego obiektu zależy od wersji JBuildera – w Personal jest na zakładce New, a w Proffesional i Enterprise na zakładce Web). Naciskamy OK.
Następnie zostaniemy spytani o nazwę pakietu, w którym umieszczona będzie klasa apletu (niech to będzie uprzednio stworzony przez nas pakiet rad5), nazwę klasy (np. Podstawy_Applet) i klasę bazową (pierowzór) tworzonej przez nas klasy. Jeżeli zdecydowaliśmy się na AWT musi to być klasa Applet (zawarta w bibliotece java.applet). Alternatywą jest JApplet z biblioteki javax.swing. W możliwych do zaznaczenia polach wyboru należy zaznaczyć „Generate standard methods" (zostaną stworzone szkielety metod `start()`, `stop()` i

[6] AWT korzysta z elementów graficznych dostarczanych przez system, w którym jest uruchamiana wirtualna maszyna Javy i dlatego wygląd kontrolek może różnić się w różnych systemach. Swing jest zbudowany od zera przez Suna.

`destroy()`) . Opcję „Can run standalone” zostawmy niezaznaczoną. Tworzy ona funkcję `main()` wywoływaną wówczas gdy applet uruchamiany jest jako samodzielna aplikacja[7]. Naciskamy Next.
Następna karta ułatwia pobieranie parametrów podawanych przez kod HTML do apletu Javy (znacznik PARAM w kodzie HTML). Tym razem pominiemy tę możliwość. Naciskamy Next.
Ostatni etap to tworzenie strony testowej w HTML. Można zmienić tytuł strony, nazwę apletu i parametry osadzenia apletu na stronie. Naciskamy przycisk Finish, żeby zakończyć działanie kreatora.

Wszystkie nasze decyzje skutkują zmianami w kodzie Javy i HTML[8], więc wszystkie parametry można modyfikować w każdej chwili w edytorze.

Zadaniem pierwszego apletu jest pokazanie podstaw projektowania RAD, a znającym inne środowiska Borlanda pokazać podobieństwa i różnice projektowania w JBuilderze względem Delphi/C++ Buildera. Przejdźmy na kartę **Design** (u dołu edytora) i połóżmy na formę przycisk. Widać, że nie pojawił się w miejscu przez nas zakreślonym, a w na środku, przy górnej krawędzi apletu. Co więcej nie można go z tamtąd ruszyć. Dzieje się tak dlatego, że położeniem elementów interface’u graficznego steruje menadżer położenia (Layout manager). W pełni można go docenić przy wykorzystywaniu biblioteki Swing, a tymczasem przełączmy go zmieniając wartość własności formy **layout** z <default layout> na null[9]. Teraz możemy w dowolny sposób ustawiać położenie przycisku. Ustalmy też jego własność **label** na „Przycisk”. Jeżeli zajrzymy teraz do kodu (zakładka Source) zobaczymy, że w funkcji `jbInit()` zostały uwzględnione zmiany wprowadzone w projekcie apletu. Po pierwsze ustalono własność layout poleceniem

```
this.setLayout(null);
```

oraz powołany został przycisk. Obiekt tworzony jest poza jakąkolwiek funkcją jednak w obrębie deklaracji klasy naszego apletu poleceniem podobnym jak w C++:

```
Button button1 = new Button();
```

Zasadniczą różnicą względem C++/C++ Buildera jest brak gwiazdki przy deklaracji zmiennej button1. W C++ w tym kontekście mielibyśmy:

```
Button* button1 = new Button();
```

tzn. stworzony zostałby wskaźnik `button1` i operator `new` zapisałby do niego adres utworzonego obiektu. Jednak w Javie nie ma wskaźników, w zamian deklarujemy zmienną typu obiektowego (**referencję**) i przypisujemy jej utworzony operatorem `new` obiekt. W przeciwieństwie do C++ w Javie nie ma operatora delete. Usuwaniem zbędnych zmiennych i obiektów zajmuje się za nas wirtualna maszyna Javy (ang. *Java Virtual Machine*) – JVM.

Ustalenie własności i umieszczenie przycisku na formie realizowane jest następującymi poleceniami:

```
button1.setLabel("Przycisk1");
button1.setBounds(new Rectangle(158, 14, 75, 27));
this.add(button1, null);
```

Pierwsza i druga linia ustala własności, etykietę oraz położenie i rozmiar[10]. Trzecia linia umieszcza obiekt na oknie apletu (podobnie jak ustalenie wartości `Parent` w VCL i CLX).

Uwaga! Java jest językiem czułym na wielkość liter.

---

[7] Ustalamy stan tej opcji na nie włączony, żeby samodzielnie ją napisać w jednym z poniższych ćwiczeń, ale przy normalnych zastosowaniach, jeżeli chcemy, aby applet mógł działać jako samodzielna aplikacja, znacznie praktyczniej jest pozwolić zbudować ją JBuilderowi.
[8] W przeciwieństwie do Delphi i C++ Buildera nie ma w projekcie plików binarnych (np. .dfm), które nie są dostępne z dowolnego edytora.
[9] Podobnie jak w Delphi aby w okienku własności pojawił się wybrany obiekt, w tym wypadku forma, należy kliknąć na wybrany obiekt w zakładce `Design` lub wybrać go w drzewku struktury obiektu.
[10] Polecenie w drugiej linii wykorzystuje jawnie numery pixeli we współrzędnych zorientowanych tak, że punkt (0, 0) znajduje się w lewym górnym rogu appletu i współrzędne rosną w dół i w prawo.

3. Struktura kodu apletu tworzonego przez JBuildera

Przyjrzyjmy się kodowi Java utworzonemu przez JBuildera.
Pierwsza linia

```
package rad5;
```

to deklaracja pakietu do którego należy aplet. Pakiet musi mieć taką samą nazwę jak katalog, w którym znajdą się skompilowane pliki. O to zadba JBuilder.
Następnie znajdują się polecenia importu bibliotek np. `import java.awt.*;` (zastępują one dyrektywy kompilatora C++ `#include`). Za nimi rozpoczyna się klasa zadeklarowana jako:

```
public class Podstawy_Applet extends Applet
```

(klasa publiczna o nazwie `Podstawy_Applet` dziedzicząca z klasy bazowej `Applet`). Nazwa głównej klasy powinna być identyczna jak nazwa pliku, w której znajduje się kod (tzn. w naszym przypadku Podstawy_Applet.java). Zaraz na początku klasy JBuilder umieścił deklaracje obiektów będących własnościami nadrzędnego obiektu. Wśród metod klasy ważna jest funkcja o nazwie identycznej jak nazwa obiektu – podobnie jak w C++ jest to konstruktor klasy. JBuilder, podobnie jak C++ Builder, tworzy go, ale nie wykorzystuje. Niezmiernie istotną w przypadku apletów jest funkcja `init()` wywoływana przez przeglądarkę do inicjacji obiektu. JBuilder nie zapisuje poleceń bezpośrednio do `init()`, ale wywołuje z niej metodę `jbInit()`. W `jbInit()` znajdą się polecenia określające wygląd wszystkich komponentów, jakie znajdą się na formie apletu.

W zależności od ustalenia opcji „Generate standard methods" w pierwszym kroku kreatora apletu poza wymienionymi metodami mogą w klasie być jeszcze zdefiniowane metody `start()`, `stop()` i `destroy()` – wszystkie puste. Zawsze są tam również metody służące do pobierania informacji o aplecie `getAppletInfo()` zwracający rozpoznawane przez aplet parametry uruchomienia.

4. Zdarzenia i metody zdarzeniowe

Zgodnie z filozofią programowania RAD możemy projektować wygląd formy i przypisywać do poszczególnych komponentów, a właściwie do ich zdarzeń, odpowiednie metody. Można to zrobić klikając dwukrotnie na obiekt lub z zakładki Events w oknie analogicznym do Object Inspectora.

Kliknijmy dwukrotnie na przycisk. JBuilder utworzy metodę:

```
void button1_actionPerformed(ActionEvent e)
{
}
```

Powiązanie jej ze zdarzeniem jest bardziej złożone niż w C++ Builderze, gdzie zdarzenie było po prostu wskaźnikiem do metody. Tutaj korzysta się z obiektów „nasłuchujących", obserwujących obiekt. Odpowiednie powiązanie umieszczone w metodzie `jbInit()` poleceniem

```
button1.addActionListener(new java.awt.event.ActionListener()
{
  public void actionPerformed(ActionEvent e)
  {
    button1_actionPerformed(e);
  }
});
```

Jest to jedno polecenie `button1.addActionListener( )`, w którym jako argument podany jest nowoutworzony obiekt „nasłuchujący" klasy `java.awt.event.ActionListener`, który posiada jedną

publiczną metodę `actionPerformed` wywoływaną gdy nastąpi oczekiwane zdarzenie (tj. gdy wykonana została na przycisku jakaś akcja). Metoda ta wywołuje z kolei metodę `button1_ActionPerformed()`, z argumentem obiektu-zdarzenia, którą możemy dowolnie modyfikować.
W istocie całym tym mechanizmem nie trzeba sobie zaprzątać głowy – martwi się o to JBuilder. Nam wystarczy wiedza, że po naciśnięciu przycisku wywołana zostanie metoda `button1_ActionPerformed()`.

Umieśćmy w metodzie `button1_ActionPerformed()` polecenie zmieniające etykietę przycisku na opis akcji użytkonika:

```
button1.setLabel(e.toString());
```

Żeby cały łańcuch opisujący zdarzenie zmieścił się na przycisku należy rozszerzyć go i być może także samą formę apletu.

Kompilujemy i uruchamiamy aplet. Tradycyjnie, jak we wszystkich produktach Borlanda, można to zrobić klawiszem funkcyjnym F9.

Łańcuch jaki pojawi się na przycisku po jego naciśnięciu powinien być następujący:

java.awt.event.ActionEvent[ACTION_PERFORMED,cmd=Przycisk1] on button0

Aby pobrać samą nazwę komendy (równą etykiecie) możemy wykorzystać metodę `e.getActionCommand()`.

Można o tych różnicach nie myśleć zbyt wiele, bo zdarzenie jakie może wywołać użytkownik na przycisku jest tylko jedno. Wystarczy więc zająć się uzupełnianiem metody zdarzeniowej bez analizowania jak jest ona wywoływana. Niech po naciśnięciu przycisku kolor apletu zmieni się na różowy.
Cała metoda zdarzeniowa powinna wyglądać następująco:

```
void button1_actionPerformed(ActionEvent e)
  {
    button1.setLabel(e.toString());
    this.setBackground(Color.pink);
  }
```

Uwaga! Poza kompilacją i obejrzeniem apletu jako samodzielnej aplikacji w narzędziu Borlanda (F9) można go zobaczyć na stronie HTML w przeglądarce wbudowanej w JBuildera. W tym celu w okienku pokazującym pliki projektu należy dwukrotnie kliknąć Podstawy_Applet.html. Nie wszystko co tam zobaczymy, można będzie oglądać w zewnętrznej przeglądarce. Dotyczy to w szczególności apletów korzystających z biblioteki Swing.

5. Metody standardowe apletu

Przetestujmy teraz działanie metod standardowych `start()`, `stop()` i `destroy()` dodanych przez kreator do projektu. Jeżeli w kreatorze nie zaznaczono opcji „Generate standard methods" można je również dodać ręcznie deklarując np.

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

Test będzie bardzo prosty. Zmusimy każdą z metod do powiadomienia o swoim działaniu przez dodanie do listy odpowiedniego łańcucha. W tym celu na formę połóżmy komponent `java.awt.List` z zakładki AWT. Następnie do poszczególnych metod wpiszmy po jednym poleceniu:

```
public void start()
  {
    list1.add("Wywolanie metody start()");
  }
```

```
public void stop()
  {
    list1.add("Wywolanie metody stop()");
  }

public void destroy()
  {
    list1.add("Wywolanie metody destroy()");
  }
```

Analogiczne polecenie należy dodać również do metody `init()`, najlepiej po wywołaniu `jbInit()`.

Uruchamiając aplet (F9) można teraz sterować działaniem apletu z przeglądarki Borlanda wykorzystując klawisze Start, Stop i Exit. Można go również obejrzeć we wbudowanej przeglądarce lub w zewnętrznej przeglądarce obejrzeć Podstawy_Applet.html[11].

6. Osadzanie apletu w dokumencie HTML

JBuilder tworzy i dodaje do projektu stowarzyszoną z apletem stronę HTML, która pozwala na testowanie działania apletu. Przykładowa zawartość dokumentu może być następująca:

```
<html>
<head>
<title>
HTML Test Page
</title>
</head>
<body>
rad5.Podstawy_Applet will appear below in a Java enabled browser.<br>
<applet
  codebase = "."
  code     = "rad5.Podstawy_Applet.class"
  name     = "Test_Podstawy_Applet"
  width    = "525"
  height   = "300"
  hspace   = "0"
  vspace   = "0"
  align    = "top"
>
</applet>
</body>
</html>
```

Część wytłuszczona odpowiada za osadzenie pakietu w dokumencie. Należy pamiętać o uwzględnieniu nazwy pakietu/katalogu we własności `code`. Aby aplet został załadowany musi znajdować się w katalogu `.\rad5`, a plik/klasa główna musi mieć nazwę `Podstawy_Applet.class`.

Uwaga! Przenosząc aplet do innego katalogu lub innego komputera (serwera) należy skopiować podkatalog o nazwie identycznej jak pakiet w katalogu classes, a w nim wszystkie pliki z rozszerzeniem *.class używane w przenoszonym aplecie.

[11] W starszych przeglądarkach (szczególnie Netscape) aplet nie był uruchamiany ponownie przy przeładowywaniu strony lub wyjścia do innej strony i powrocie. Dzięki temu apletowi można to łatwo wykryć.

7. Uruchamianie apletu jako samodzielnej aplikacji

Do przedstawionej na początku rozdziału tabelki można dodać jeszcze jeden wiersz:

`main(String args[])`	metoda wywoływana (jako jedyna) wówczas gdy program Java uruchamiany jest jako samodzielna aplikacja. W tej funkcji sami musimy stworzyć formę, klasę apletu i musimy zadbać o wywołanie jej funkcji `init()` i `start()`. Zob. paragraf „Uruchamianie apletu jako samodzielnej aplikacji”.

W pakiecie JDK dostarczanym razem z JBuilderem znajduje się narzędzie do uruchamiania aplikacji i apletów Javy w środowisku Windows – java.exe (znajduje się w katalogu JBuilder#\jdk#.#.#\bin, gdzie hashe # oznaczają numery wersji). Narzędzie to traktuje aplet podobnie jak aplikację konsolową, tzn. wywołuje jedynie metodę `main()`[12] klasy apletu (o nazwie identycznej jak nazwa pliku)[13]. Ta funkcja musi sama zbudować okienko klasy `Frame` oraz umieścić na nim zaprojektowany obiekt apletu (proszę zauważyć, że obiekt klasy `Podstawy_Applet` tworzymy we wnętrzu jego metody, ale jest to metoda statyczna) oraz wywołać jego funkcje `init()` i `start()`. Metoda `main()` musi być oczywiście zadeklarowana jako publiczna (tzn. można ją uruchomić bez konieczności tworzenia obiektu tej klasy).

```
public static void main(String args[])
  {
    //Ten napis pojawi sie w konsoli,
    //gdy applet uruchamiany jest za pomoca samodzielnego interpretatora
    System.out.println("Wywolanie metody main");

    //Tworzymy obiekt apletu
    Podstawy_Applet pApplet=new Podstawy_Applet();
    pApplet.isStandalone=true;

    //Przygotowujemy okno i umieszczamy na nim aplet
    Frame forma=new Frame("Podstawy");
    forma.setSize(640,480);
    forma.add("Center",pApplet);
    forma.show();

    //Inicjacja apletu
    pApplet.init();
    pApplet.start();
  }
```

To powoduje, że aplet zachowuje się jak w pełni funkcjonalna aplikacja Windows. Można go teraz uruchomić poleceniem

```
java rad5.Podstawy_Applet
```

wywołanym, to istotne, z katalogu nadrzędnego. rad5 jest określeniem katalogu i jednocześnie nazwą pakietu, do którego należy klasa Podstawy_Applet. Podobnie jak nazwa klasy głównej musi być identyczna jak nazwa pliku, w którym się znajduje, nazwa pakietu musi być identyczna z nazwą katalogu.
Aplet, a właściwie aplikacja, nie daje się na razie zamknąć, ale z tym problemem nie będziemy się borykać teraz (należy go zamknąć z poziomu okna linii poleceń, w którym został uruchomiony).

[12] Metoda main() zostanie stworzona automatycznie przez JBuildera jeżeli podczas tworzenie apletu zaznaczymy opcję „Can run standalone”.
[13] Podobieństwo do C++ może być mylące. W C++ wywoływana jest funkcja `main()`, a w Javie w uruchamianym samodzielnie aplecie wywoływana jest metoda `main()` głównego obiektu apletu.

Uwaga!
O tym jak umieścić aplet/aplikację Javy w pliku JAR i uruchamiać go bez jawnego korzystania z java.exe można przeczytać w rozdziale „Jak .. ?”.

Uwaga!
W części poświęconej bibliotece Swing znajduje się krótki rozdział poświęcony kreatorowi aplikacji w JBuilderze.

III. Przykładowe aplety AWT

Biblioteka komponentów sterujących (kontrolek) AWT (ang. *Abstract Window Toolkit*) jest traktowana przez twórców Javy jako przestarzała i nie będzie dalej rozwijana. Ma ona oczywiście kilka wad, np. to, że kontrolki korzystają z obiektów dostarczanych przez API systemu i w konsekwencji ich wygląd różni się na różnych platformach, jednak niezaprzeczalną zaletą jest obsługa AWT przez przeglądarki bez konieczności instalowania jakichkolwiek pluginów.

1. Scrollbars

Tworzymy nowy aplet o nazwie `ScrollBars_Applet`. Nanosimy na niego trzy komponenty `java.awt.Scrollbar` z palety AWT. Należy pamiętać, żeby wcześniej na karcie Design ustalić własność apletu `this.layout` na `null`. Paski przewijania (`scrollbar1 – 3`) zorientowane są pionowo. Aby je przeorientować poziomo należy własność `orientation` każdego z nich ustalić na 0. Dorzućmy jeszcze panel (`java.awt.Panel`) i ustalmy jego kolor tła (`background`) na czarny.

a) RGB
Paski przewijania mają posłużyć do ustalenia składowych RGB koloru panelu, powinny więc przyjmować liczby z zakresu od 0 do 255. Wobec tego ich własność maximum musimy zmienić na 255.

Z pierwszym paskiem związemy metodę, którą następnie przypiszemy do dwóch kolejnych zdarzeń. Przechodzimy do okna własności w widoku projektowania (Design), zmieniamy zakładkę na Events i dwukrotnie klikamy zdarzenie `adjustValueChanged`. Następnie w metodzie zdarzeniowej określamy kolor panelu korzystając z ustawień pasków przewijania:

```
void scrollbar1_adjustmentValueChanged(AdjustmentEvent e)
  {
    Color kolor=new Color(scrollbar1.getValue(),
                          scrollbar2.getValue(),
                          scrollbar3.getValue());
    panel1.setBackground(kolor);
  }
```

W pierwszej linii kodu metody deklarowany jest obiekt kolor, którego jeden z przeciążonych konstruktorów przyjmuje trzy wartości składowych RGB. Równie łatwo dostępna jest obsługa HSB.

Ponieważ argumentem metody `setBackground()` ma być obiekt klasy `Color`, który nie jest więcej wykorzystywany, kod metody można skrócić do jednego polecenia wykorzystując w argumencie wynik działania operatora `new` (będący w Javie referencją):

```
void scrollbar1_adjustmentValueChanged(AdjustmentEvent e)
  {
    panel1.setBackground(new Color(scrollbar1.getValue(),
                                   scrollbar2.getValue(),
                                   scrollbar3.getValue()));
  }
```

Podłączymy teraz `scrollbar2` i `scrollbar3` do tej samej metody zdarzeniowej. Postąpimy identycznie jak należałoby postąpić w Delphi/C++ Builderze. Na zakładce Design w „Object Inspectorze” przy zdarzeniu `adjustmentValueChanged` wpisujemy `scrollbar1_adjustmentValueChanged`. W metodzie jbInit pojawi się wywołanie `scrollbar2.addAdjustmentValueChanged` wskazujące na tą właśnie funkcję. W przypadku `scrollbar3` postępujemy identycznie. Dzięki temu możliwa jest kontrola koloru `panel1` za pomocą trzech pasków przewijania.

b) HSB

Poza układem współrzędnych RGB, w Javie równie proste jest posługiwanie się układem HSB. Klasa `Color` posiada odpowiedni zestaw metod.

Dodajmy do apletu dwa komponenty `java.awt.Checkbox`, pierwszy z etykietą RGB, drugi – HSB. Aby nadać im charakter typowy dla komponentów pól wyboru (radiobox) na aplecie należy położyć niewizualny komponent `java.awt.CheckboxGroup` i przypisać go do własności checkboxGroup obydwu pól opcji. Po uruchomieniu apletu pola będą wyglądały jak typowe pola wyboru, ale podczas projektowania ich kształt pozostanie taki jak dla pól opcji. Ustalmy stan pola z etykietą „RGB" na zaznaczony.

Aby umożliwić definiowanie koloru we współrzędnych HSB musimy zmodyfikować metodę zdarzeniową związaną ze zmianą pozycji pasków przewijania. Jeżeli zaznaczony jest `checkbox2` należy zastąpić standardowy konstruktor przez metodę `Color.getHSBColor()`, która jest metodą statyczną i zwraca obiekt typu `Color`. Można go więc nazwać alternatywnym konstruktorem dla współrzędnych HSB. Metoda ta przyjmuje trzy argumenty typu `float` z zakresu od 0 do 1, stąd odpowiednie rzutowanie.

```
void scrollbar1_adjustmentValueChanged(AdjustmentEvent e)
  {
    if (checkbox1.getState())
      panel1.setBackground(new Color(scrollbar1.getValue(),
                                     scrollbar2.getValue(),
                                     scrollbar3.getValue()));
    if (checkbox2.getState())
      {
      panel1.setBackground(Color.getHSBColor(
                  scrollbar1.getValue()/(float)scrollbar1.getMaximum(),
                  scrollbar2.getValue()/(float)scrollbar2.getMaximum(),
                  scrollbar3.getValue()/(float)scrollbar3.getMaximum()));
      }
  }
```

W momencie przełączania między współrzędnymi kolorów zmienia się sens wartości określanych przez paski przewijania, ale ich wartość pozostaje niezmieniona. Powoduje to skokową zmianę kolorów przy poruszenia któregokolwiek paska przewijania. Aby pozbyć się tego efektu należy równocześnie ze zmianą współrzędnych zmienić pozycję suwaków na paskach. Ze zdarzeniami `itemStateChanged` obu checkboxów zwiążmy metodę:

```
void checkbox1_itemStateChanged(ItemEvent e)
  {
    if(checkbox1.getState())
      {
      scrollbar1.setValue(panel1.getBackground().getRed()) ;
      scrollbar2.setValue(panel1.getBackground().getGreen()) ;
      scrollbar3.setValue(panel1.getBackground().getBlue()) ;
      }
    if(checkbox2.getState())
      {
      Color kolor=panel1.getBackground();
      float[] hsbkolory=Color.RGBtoHSB(kolor.getRed(),
                                       kolor.getGreen(),
                                       kolor.getBlue(),null);
      scrollbar1.setValue((int)(255*hsbkolory[0]));
      scrollbar2.setValue((int)(255*hsbkolory[1])) ;
      scrollbar3.setValue((int)(255*hsbkolory[2])) ;
      }
  }
```

W przypadku współrzędnych RGB działanie tej metody jest oczywiste. W przypadku wybrania współrzędnych HSB wygodnie jest skorzystać z metody statycznej `Color.RGBtoHSB` zamieniającej współrzędne RGB na współrzędne HSB zapisywanej do trójelementowej tablicy `float`. Metoda ta zwraca referencję do tablicy, a

ponadto pozwala też na zwrócenie wartości przez argument (my korzystamy z pierwszej metody, dlatego jako trzeci argument podajemy „pusty” adres `null`).

2. Wczytywanie plików tekstowych. Notatnik

Notatnik będzie kolejną ilustracją wykorzystania AWT, a przy okazji pokażę jak aplet może pobierać parametry ustalane przez przeglądarkę znacznikiem <param> umieszczonym w obrębie <applet> i </applet>.

Parametr uruchomienia apletu
Podczas tworzenia apletu, po podaniu pakietu (rad5) i nazwy apletu (Notatnik_AWT_Applet), pokazuje się okno pozwalające zdefiniować parametry rozpoznawane przez aplet. Dodanie parametru o nazwie „Tekst” związanej ze zmienną `par_tekst` i domyślną wartością „sample.txt” spowoduje:

1) W kodzie HTML dodanie linii
   `<param name = "FileName" value = "sample.txt">`
   w bloku znacznika `<applet>`
2) W kodzie Notatnik_AWT_Applet.java, w metodzie `init()` głównej klasy dodane zostanie wywołanie funkcji `getParameters()`:

```
try
     {
       parFileName = this.getParameter("FileName", "sample.txt");
     }
catch(Exception e)
     {
       e.printStackTrace();
     }
```

3) Do klasy `Notatnik_AWT_Applet` dodana zostanie własność `parTekst` typu `String`.

Metoda `getParameters()` jest odpowiedzialna za pobranie parametru o podanej w pierwszym argumencie nazwie, a jeżeli takie nie jest zdefiniowana – zwraca wartość podaną w argumencie drugim.

Interface
W zakładce Design ustalamy `layout` na `null`, a następnie umieszczamy na aplecie następujące komponenty z zakładki AWT:

klasa	nazwa	konfiguracja
`java.awt.TextArea`	`textArea1`	
`java.awt.Checkbox` (7 razy)	`checkbox1 - checkbox7`	1) `label = Możliwa edycja, state = True` 2) `label = Wytłuszczenie` 3) `label = Kursywa` 4) `label = Czarny` 5) `label = Czerwony` 6) `label = Zielony` 7) `label = Niebieski`
`java.awt.Button` (5 razy)	`button1 - button5`	1) `label = Czytaj` 2) `label = Zapisz` 3) `label = Zaznacz wszystko` 4) `label = Kopiuj` 5) `label = Wklej`

Każdy checkbox będzie związany z metodami zdarzeniowymi związaną z `itemStateChanged`:

Aby uwzględnić przekazywany z przeglądarki należy w `jbInit()` zmodyfikować funkcję ustalającą tekst w `textArea1` na `textArea1.setText(parTekst);`.

Możliwa edycja:

```
void checkbox1_itemStateChanged(ItemEvent e)
  {
    textArea1.setEnabled(checkbox1.getState());
  }
```

Kursywa, Wytłuszczenie (oba komponenty związane z tą samą metodą)**:**

```
void checkbox2_itemStateChanged(ItemEvent e)
  {
    Font czcionka=textArea1.getFont();
    //zaczynamy od czystego i ewentualnie dodajemy wlasnosci
    int styl=Font.PLAIN;
    if (checkbox2.getState()) styl=styl+Font.BOLD;
    if (checkbox3.getState()) styl=styl+Font.ITALIC;
    //argumenty konstruktora: nazwa, styl, rozmiar
    Font nowa_czcionka=new Font(czcionka.getName(),styl,czcionka.getSize());
    textArea1.setFont(nowa_czcionka);
  }
```

W pierwszej linii metody do zmiennej (referencji) pobieramy metodą `TextArea.getFont()` obiekt opisujący aktualną czcionkę w textarea1. Następnie w kolejnych trzech liniach sprawdzając czy checkbox2 i checkbox3 są zaznaczone ustalamy wartość zmiennej styl uwzględniając wytłuszczenie i kursywę czcionki. Tworzymy obiekt nowej czcionki ustalając krój czcionki i jej wielkość na podstawie własności obiektu `czcionka` oraz jej styl zdefiniowany w poprzednich liniach. W ostatniej linii przypisujemy metodą `TextArea.setFont()` stworzoną czcionkę do textArea1.

Czarny:

```
void checkbox4_itemStateChanged(ItemEvent e)
  {
    textArea1.setForeground(Color.black);
  }
```

Pozostałe metody odpowiedzialne za zmianę kolorów będą analogiczne. Zmieniać się będzie tylko stała określająca kolor (`Color.red`, `Color.green`, `Color.blue`).

Klawisze `button1` – `button5` związemy z metodami zdarzeniowymi typu actionPerformed.
Obsługa plików w Javie wymaga zaimportowania pakietu `java.io`, co oznacza, że na początku pliku musimy dodać:

```
import java.io.*;
```

Operacje na plikach nie są zbyt wygodne w obsłudze, a sprawę komplikuję dodatkowo polityka bezpieczeństwa przeglądarki gospodarza (ang. *host*), które w domyślnej konfiguracji pozwalają apletom Java na odczyt pliku, ale zabraniają zapisu danych na lokalnym dysku.

Czytaj
Klikając dwukrotnie na przycisk z etykietą „Czytaj” tworzymy metodę `button1_actionPerformed()`. Ustalmy wpierw nazwę pliku, który ma być wczytany do pola tekstowego. Nazwa pliku została podana przez parametr z przeglądarki (lub jeżeli brak odpowiedniego parametru – ustalona na domyślną, tj. „sample.txt”). Należy jednak uzupełnić ją o ścieżkę dostępu. W tym celu w metodzie deklarujemy zmienną `nazwapliku` typu `String` i przypisujemy jej wartość korzystając z metody `Applet.getCodeBase().getFile()` jeżeli plik znajduje się w katalogu, w którym znajduje się dokument HTML z osadzonym apletem[14]:

```
String nazwapliku=this.getCodeBase().getFile()+parFileName;
```

[14] Nazwę ze ścieżką do dokumentu można uzyskać metodą `this.getDocumentBase()`.

Aby odczytać plik musimy wykorzystać klasę `FileInputStream` (z biblioteki `java.io`) pozwalającą na dostęp do plików na komputerze klienta. Do czytania pliku tekstowego (odczyt znaków zamiast bajtów) wykorzystamy opakowanie strumienia wejścia w postaci obiektu klasy `BufferedReader`.

```
FileInputStream fis=new FileInputStream(nazwapliku);
BufferedReader buffer=new BufferedReader(new InputStreamReader(fis));
```

Obie metody wymagają wyłapywania wyjątków.
Metodą `BufferedReader.read()` zwraca kolejne znaki, które zapiszemy do zmiennej typu `String`. Następnie zamkniemy dostęp do pliku i tekst pokażemy w `textArea1`:

```
String tmpTekst="";
int litera;
while ((litera=buffer.read())!=-1) {tmpTekst=tmpTekst+(char)litera;}
//czyta dopoki operacja przypisania nie zroci bledu (-1)
buffer.close();
fis.close();
textArea1.setText(tmpTekst);
```

Cała metoda powinna wyglądać następująco:

```
void button1_actionPerformed(ActionEvent e)
  {
  //trzeba zadeklarowac java.io.*
  String nazwapliku=this.getCodeBase().getFile()+parFileName;
  try
    {
    //Buforowane (trzeba skorzystac z czytania lancucha)
    FileInputStream fis=new FileInputStream(nazwapliku);
    BufferedReader buffer=new BufferedReader(new InputStreamReader(fis));
    String tmpTekst="";
    int litera;
    //czyta dopoki operacja przypisania nie zroci bledu (-1)
    while ((litera=buffer.read())!=-1) {tmpTekst=tmpTekst+(char)litera;}
    buffer.close();
    fis.close();
    textArea1.setText(tmpTekst);
    showStatus("Edytowany plik: "+nazwapliku);
    }
  catch(Exception exc)
    {
    showStatus("Komunikat bledu przy czytaniu z pliku: "+exc.getMessage());
    }
  }
```

Zadanie: Stworzyć metodę LoadFromTextFile(*nazwa_pliku*) zwracającą łańcuch znajdujący się w pliku o wskazanej w argumencie nazwie.

Zapisz
Tworzymy metodę zdarzeniową dla klawisza „Zapisz”
Bardzo podobnie wygląda zapisywanie do plików (oczywiście to czy się ono powiedzie zależy od uprawnień jakie posiada aplet). Tym razem korzystamy z obiektów `FileOutputStream` oraz `BufferedWriter`. Ten ostatni posiada metodę `write()`, która potrafi zapisać do pliku String.

```
void button2_actionPerformed(ActionEvent e)
  {
  String nazwapliku=this.getCodeBase().getFile()+parFileName+".saved";
  try
    {
    //Buforowane (tylko bufor umie pisac stringi)
    FileOutputStream fos=new FileOutputStream(nazwapliku);
```

```
    BufferedWriter buffer=new BufferedWriter(new OutputStreamWriter(fos));
    buffer.write(textArea1.getText());
    buffer.close();
    fos.close();
    showStatus("Zawartość edytora zapisana do pliku: "+nazwapliku);
    }
  catch(Exception exc)
    {
    showStatus("Komunikat bledu przy zapisie do pliku: "+exc.getMessage());
    }
  }
```

Zaznacz wszystko
Aby zaznaczyć cały tekst należy skorzystać z metody `TextArea.selectAll()`:

```
void button3_actionPerformed(ActionEvent e)
  {
    textArea1.selectAll();
    textArea1.requestFocus();
    showStatus("Zaznaczenie całego tekstu");
  }
```

Kopiuj, **Wklej** – obsługa schowka
Podobnie jak w przypadku zapisu plików na dysk, kopiowanie do i ze schowka jest domyślnie zablokowane ze względu na bezpieczeństwo danych. Można sobie bowiem wyobrazić aplet, który pobiera informacje ze schowka i natychmiast przesyła go do serwera. Zawsze będą działały klawisze Ctrl+C, Ctrl+X, Ctrl+V, ale one są obsługiwane nie przez aplet, a przez przeglądarkę i dlatego są bezpieczne (informacja pozostaje w systemie hosta).

Aby korzystać ze schowka musimy zaimportować pakiety `java.awt.datatransfer.*`.
Schowek jest obsługiwany jest przez klasę `Clipboard`, którą możemy zainicjować pobierając schowek systemowy:

```
Clipboard schowek=getToolkit().getSystemClipboard();
```

Istotnymi metodami klasy `Clipboard` są `setContents()` i `getContents()` pozwalające na zapisanie do niego i odczytanie jego zawartości. Pozostała część obu metod wiąże się z pobraniem zaznaczonego fragmentu tekstu (dla „Kopiuj") i wstawieniem tekstu do `textArea1` („Wklej").

W schowku nie może być przechowywany `String`, który nie współdzieli interface'u `ClipboardOwner` i `Transferable`), a jedynie obiekt klasy `StringSelection`. Dlatego zaznaczony fragment tekstu pobrany metodą `textArea1.getSelectedText()` kopiujemy do zmiennej `tekstschowek` klasy `StringSelection`. Ta zmienna może być argumentem `Clipboard.setContents()` zarówno pierwszym podającym zawartość, jak i drugim, wskazującym na właściciela tej zawartości.

```
void button4_actionPerformed(ActionEvent e)
  {
    //Obsluga schowka wymaga java.awt.datatransfer
    Clipboard schowek=getToolkit().getSystemClipboard();
    String zaznaczonytekst=textArea1.getSelectedText();
    if (zaznaczonytekst==null) return; //warunkowe opuszczenie funkcji
    StringSelection tekstschowek=new StringSelection(zaznaczonytekst);
    schowek.setContents(tekstschowek,tekstschowek);
    showStatus("Tekst skopiowany do schowka");
  }
```

Metoda `getContents()` zwraca obiekt typu `Transferable`, który będzie reprezentował zawartość schowka (może nim być także `StringSelection`, konieczne jest wówczas zrzutowanie na ten typ). Interface Transferable posiada metodę `getTransferData()`, która zwraca zawartość schowka. Jej argumentem jest

typ pobieranych ze schowka danych. Możliwe do przechowywania w nim typy są przechowywane w klasie `DataFlavour` (statycznie, tzn. nie trzeba tworzyć obiektu tej klasy, żeby zobaczyć tą listę typów).

```
void button5_actionPerformed(ActionEvent e)
  {
    Clipboard schowek=getToolkit().getSystemClipboard();
    Transferable tekstschowek=schowek.getContents(this);
    try
      {
      //Operacja odczytywania schowka wymaga try
      textArea1.insert(
             (String)tekstschowek.getTransferData(DataFlavor.stringFlavor),
             textArea1.getCaretPosition());
      showStatus("Wklejona zawartość schowka");
      }
    catch (Exception exc)
      {
      showStatus("Blad odczytu schowka: "+(String)exc.getMessage());
      }
  }
```

Informacje dotyczące polityki bezpieczeństwa znajdują się w rozdziale „Jak ... ?”.

3. Dynamiczne tworzenie obiektu

Zadaniem aplikacji jest dodawanie dynamicznie tworzonego przycisku po każdym kliknięciu okna apletu w miejscu jej kliknięcia i z opisem odpowiadającym naciśniętemu przyciskowi myszy.

Tworzymy nowy aplet o nazwie `DynamicButton_Applet`. Ustalamy `this.layout` na `null`. Działanie apletu będzie polegało na tworzeniu nowego przyciku za każdym razem gdy użytkownik kliknie okno apletu. Musimy zatem stworzyć metodę zdarzeniową związaną z odpowiednim zdarzeniem obiektu `DynamicButton_Applet`. Przejdźmy do zakładki Design i w oknie własności, w zakładce Events, dwókrotnie kliknijmy zdarzenie `mouseClicked` (ewentualnie można użyć `mousePressed`). Powstanie interesująca nas metoda zdarzeniowa. Jej jedynym argumentem jest obiekt `MouseEvent` przenoszący informacje o myszy w chwili kliknięcia.

```
void this_mouseClicked(MouseEvent e)
  {
    Button DynamicButton=new Button(); //deklaracja obiektu
    String etykieta="("+e.getX()+","+e.getY()+")"; //położenie do łańcucha

    if ((e.getModifiers() & e.BUTTON1_MASK)==e.BUTTON1_MASK) etykieta="Lewy "+etykieta;
    if ((e.getModifiers() & e.BUTTON2_MASK)==e.BUTTON2_MASK) etykieta="Środ. "+etykieta;
    if ((e.getModifiers() & e.BUTTON3_MASK)==e.BUTTON3_MASK) etykieta="Prawy "+etykieta;

    DynamicButton.setLabel(etykieta); //ustalenie etykiety
    DynamicButton.setBounds(e.getX(),e.getY(),110,25); //rozmiar i położeni
    this.add(DynamicButton,null); //dodanie do formy
  }
```

Komentarza wymagają trzy podobne linie sprawdzające który z klawiszy myszy został naciśnięty[15]. Przyjmijmy, że naciśnięty został przycisk lewy. Wówczas metoda `e.getModifiers` zwraca wartość 16 ponieważ maska tego klawisza tyle właśnie wynosi, a inne modyfikatory nie są obecne. W zapisie binarnym oznacza to, że tylko piąty bit jest zapalony. Możliwa jest jednak sytuacja, w której `getModifiers` zwróci wartość będącą

[15] Narzuca się wykorzystanie polecenia wyboru `switch`, ale wówczas nie byłoby tak proste wykrycie naciśnięcia jednocześnie dwóch klawiszy myszy.

wynikiem nałożenia kilku modyfikatorów (np. gdy równocześnie z kliknięciem naciśniemy klawisz Alt, Ctrl lub Shift) tzn. więcej bitów jest zapalonych. Jak wówczas sprawdzić czy wśród nich znajduje się lewy przycisk. Wystarczy wykonać operację logiczną AND na pobranej wartości oraz masce i sprawdzić czy jest równa tej masce. Ponieważ maska ma tylko jeden bit zapalony oznacza możliwe są jedynie dwie wartości: zero lub wartość maski.
Załóżmy, że naciśnięto klawisz Alt (ALT_MASK=8) oraz lewy przycisk myszy (BUTTON1_MASK=16) wówczas wartość zwrócona przez getModifiers równa jest 1*ALT_MASK+1*BUTTON1_MASK=8+16=24. Wówczas wynik operacji AND (zwraca liczbę, która w przedstawieniu dwójkowym ma jedynki tylko na tych pozycjach, w których oba argumenty posiadały jedynki): 24 & BUTTON1_MASK = 24 & 16 = BIN(11000) & BIN(10000) = BIN(10000) = 16 = BUTTON1_MASK[16]. Stąd postać wyrażenia w instrukcji `if`.

W praktyce nie jest to jednak takie piękne, bo, jak łatwo się przekonać, np. wartość maski ALT_MASK jest identyczna jak BUTTON2_MASK.

4. Odtwarzanie plików dźwiękowych

Aplety mogą odtwarzać w tle pliki dźwiękowe. Niestety aplety Java (wersja 1) działające w przeglądarkach mogą odtwarzać tylko jeden format plików dźwiękowych, a mianowicie stworzony przez Suna format .au (8 bit, µlaw, 8000 Hz, one-channel (mono))[17]. Java 2 dopuszcza także inne formaty np. mid lub wav (np. wówczas gdy do uruchamiamy aplety w wewnętrznej przeglądarce JBuildera).

Tworzymy nowy aplet należący do pakietu rad5 o nazwie `Audio_Applet`. Dodajemy parametr o nazwie `FileName` związany ze zmienną `parFileName`. Dodajemy dwa przyciski z zakładki AWT („Odtwórz” i „Zatrzymaj”). Deklarujemy własność klasy `Audio_Applet` będącą obiektem `AudioClip` o nazwie `ac` i inicjujemy ją w metodzie `init()` poleceniem:

```
ac=this.getAudioClip(this.getCodeBase(),parFileName);
```

Plik o nazwie wskazanej przez parametr uruchomienia szukany jest w bieżącym katalogu (`this.getCodeBase()`). Nie ma niestety prostej metody, aby sprawdzić, czy ta operacja się powiodła. Klasa AudioClip posiada trzy metody `play()`,`loop()` i `stop()`. Pierwsza pozwala na pojedyncze odtworzenie zawartości pliku, druga na odtworzenie go zadaną ilość razy, a trzecia na zatrzymanie odtwarzania.

Klikając dwukrotnie na przycisk „Odtwarzaj” tworzymy odpowiednią metodę zdarzeniową i wpisujemy do niej następujący kod:

```
void button1_actionPerformed(ActionEvent e)
  {
  if (ac!=null)
    {
    ac.play();
    showStatus("Odtwarzany plik: "+parFileName);
    }
  }
```

Podobnie dla „Zatrzymaj”:

```
void button2_actionPerformed(ActionEvent e)
  {
  if (ac!=null)
    {
    ac.stop();
    showStatus("Odtwarzanie pliku "+parFileName+" zatrzymane");
```

[16] Najprostszym sposobem zamiany sprawdzenia reprezentacji binarnej liczby jest ... kalkulator w MS Windows.
[17] Przykładowy plik można znaleźć w dołączonych źródłach.

```
    }
  }
```

Plik może być pobrany nie tylko z lokalnego dysku, ale z dowolnego miejsca na sieci. Pierwszy parametr funkcji `Applet.getAudioClip()` powinien być obiektem klasy `URL`. Klasa `URL` znajduje się w bibliotece `java.net`. Należy więc dodać na początku programu polecenie

```
import java.net.*;
```

Aby pobrać plik `sample.au` z miejsca wskazanego na sieci można zastąpić pierwszy argument funkcji `Applet.getAudioClip()` przez
`new URL("http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/sample/")`.

Java wymaga, aby połączenie z siecią uwzględniało możliwość powstania wyjątków, dlatego w tej wersji ta linia kodu musi być otoczona przez `try ... catch`.

Uwaga!
W przypadku uruchamiania apletu Audio_Applet w przeglądarce należy się spodziewać dodatkowych kłopotów.

Zadanie:
Manipulować własnościami `enabled` przycisków, w taki sposób, żeby podczas odtwarzania aktywny był tylko przycisk „Zatrzymaj”, a po jego naciśnięciu tylko „Odtwarzaj”.

5. Obróbka plików graficznych. Negatyw

Następnym zadaniem jest stworzenie apletu, który pobiera obraz z pliku i na żądanie tworzy negatyw wybranego fragmentu.

a) Pobieranie i wyświetlanie obrazka z pliku
Tworzymy aplet `Negatyw_Applet` i kładziemy na jego formę dwa przycisku (etykiety „Pobierz” i „Negatyw”).
Deklarujemy na początku klasy obiekt `Image` oraz jego przyszłe położenie:

```
Image obrazek=null;
int obr_x=10, obr_y=70;
```

Metoda zdarzeniowa przycisku „Pobierz” ma za zadanie pobrać obrazek. Służy do tego metoda `Applet.getImage`, która podobnie jak `getAudioClip` pozwala pobrać obrazek lokalnie lub z podanego adresu sieciowego. My pobieramy obrazek „sample.gif” z katalogu classes projektu.

```
void button1_actionPerformed(ActionEvent e)
  {
    obrazek=this.getImage(this.getCodeBase(),"sample.gif");
    this.repaint();
  }
```

W następnej linii wywołujemy metodę `repaint()` apletu wymuszającą „odmalowanie” apletu, tj. odświeżenie jego interface’u graficznego. Przy okazji takiego odświeżenia wywoływana jest, o ile jest zdefiniowana, metoda `paint()`. Jeżeli umieścimy w niej polecenie obrazka będzie ono wyświetlone nie tylko po naciśnięciu przycisku, ale i przy każdym następnym odświeżaniu apletu[18]. Funkcja `paint()` może być bardzo prosta:

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

[18] Naturalne wydawałoby się po prostu wyświetlenie obrazka np. poleceniem
`this.getGraphics().drawImage(obrazek,30,10,this);`
ale to nie działa zbyt dobrze (warto się o tym samemu przekonać).

```
  {
    g.drawImage(obrazek, obr_x, obr_y, this);
  }
```

Metoda `Graphics.drawImage()` może przyjmować kolejne argumenty określające szerokość i wysokość obrazka, a więc pozwalające na jego przeskalowanie Obiekt `g` typu `Graphics` jest przekazywanym do metody `paint()` obiektem wymagającym odmalowania.

b) Zaznaczanie fragmentu obrazka
Aplet posiada kilka zdarzeń pozwalających na śledzenie myszy. Dwa z nich: `mousePressed` i `mouseReleased` zachodzące gdy klawisz myszy został naciśnięty i zwolniony, pozwolą nam na wyznaczenie obszaru zaznaczonego myszką. Zanim zajmiemy się programowaniem tych zdarzeniem musimy zadeklarować zmienne typu **`int`** dostępne w całej klasie przechowujące położenie i wielkość zaznaczonego fragmentu oraz zmienna logiczną sterującą wyświetlaniem prostokąta opisującego zaznaczony fragment:

```
int fr_x,fr_y,fr_szer,fr_wys;
boolean fr_widoczny=false;
```

Klikamy dwukrotnie na zdarzeniu `mousePressed` w okienku własności (widok Design) i w jej metodzie zdarzeniowej pobieramy współrzędne punktu w którym została naciśnięta mysz, a który będzie rogiem zaznaczonego fragmentu.

```
void this_mousePressed(MouseEvent e)
  {
  fr_x=e.getX();
  fr_y=e.getY();
  }
```

Podobnie robimy dla zdarzenia `mouseReleased`. Ponadto ustalamy wartość zmiennej `fr_widoczny` na `true` i wymuszamy odświeżenie obrazka, żeby, o ile do metody `paint()` została dopisana odpowiednia instrukcja, zobaczyć zaznaczony fragment.
Algorytm ustalania położenia i rozmiarów musi uwzględniać możliwość przeciągnięcia myszy w dowolnym kierunku – dopiero w momencie podniesienia myszy dowiemy się, czy położenie myszy w momencie naciśnięcia klawisza jest lewym-górnym rogiem.

```
void this_mouseReleased(MouseEvent e)
  {
  int x=e.getX(), y=e.getY();
  fr_szer=Math.abs(x-fr_x);
  fr_wys=Math.abs(y-fr_y);
  if (x<fr_x) fr_x=x;
  if (y<fr_y) fr_y=y;
  fr_widoczny=true;
  this.repaint();
  }
```

Do `paint()` należy dodać instrukcję rysowania prostokąta:

```
if (fr_widoczny)
  {
  g.setColor(Color.gray);
  g.drawRect(fr_x,fr_y,fr_szer,fr_wys);
  }
```

c) Negatyw
Aby stworzyć negatyw obrazka wykorzystamy istniejący w pamięci obiekt `Image` (warunkiem zobaczenia negatywu jest więc wcześniejsze pobranie obrazka). Tworzymy metodę zdarzeniową związaną z przyciskiem „Negatyw”. W tej metodzie musimy wykonać następujące operacje:

1) obliczenie położenia prostokąta we współrzędnych obrazka zaznaczonego fragmentu ograniczonego do zakresu samego obrazka.
2) pobranie tablicy pikseli odpowiadających obrazkowi
3) zdekodowanie kolorów RGB oraz przezroczystości alpha dla każdego piksela[19]
4) odwracamy kolory pikseli bez zmieniania przezroczystości
5) kodujemy zmienione piksele
6) Zapisujemy tablicę pikseli do zmiennej typu `Image` i wymuszamy jej „namalowanie" (w tym celu należy dodać dodatkową zmienną `Image` negatyw do klasy i zadbać o jej wyświetlenie w metodzie `paint()`).

Punkty 3-5 będą oczywiście wykonywane w pętli, żeby ograniczyć ilość wykorzystywanej pamięci.

1)
Obliczamy prostokąt jaki wyznaczony jest przez zaznaczony fragment (użytkownik nie jest związany przez rozmiary obrazka i może wyznaczyć fragment nie pokrywający się z wyświetlonym obrazkiem):
Dla ułatwienia pobieramy wpierw i zapisujemy do zmiennych wysokość i szerokość oryginalnego obrazka.

```
int obr_szer=obrazek.getWidth(this), obr_wys=obrazek.getHeight(this);
```

Następnie zapisujemy współrzędne wybranego fragmentu formy przesunięte do układu odniesienia obrazka:

```
int obr_fr_x=fr_x-obr_x, obr_fr_y=fr_y-obr_y,
    obr_fr_szer=fr_szer, obr_fr_wys=fr_wys;
```

Jeżeli położenie fragmentu (we współrzędnych obrazka) jest ujemne to należy przesunąć odpowiednią krawędź zaznaczonego fragmentu do krawędzi obrazka i skrócić o tą długość odpowiedni wymiar:

```
if (obr_fr_x<0)
  {
  obr_fr_x=0;
  obr_fr_szer-=(obr_x-fr_x); //(z C++: a-=b oznacza a=a-b)
  }
if (obr_fr_y<0)
  {
  obr_fr_y=0;
  obr_fr_wys-=(obr_y-fr_y);
  }
```

Podobnie w sytuacji, gdy prawy dolny róg wystaje poza obręb obrazka – fragment zostanie przycięty do wielkości obrazka.

```
if (obr_fr_x+obr_fr_szer>obr_szer) obr_fr_szer=obr_szer-obr_fr_x;
if (obr_fr_y+obr_fr_wys>obr_wys) obr_fr_wys=obr_wys-obr_fr_y;
```

Aby pokazać nowy przycięty zakres zaznaczonego fragmentu przepisujemy nowe współrzędne i rozmiary do zmiennych `fr_x`, `fr_y`, `fr_szer` i `fr_wys`, które sterują wyświetlanym prostokątem i wymuszamy jego ponowne odmalowanie:

```
fr_x=obr_x+obr_fr_x; fr_y=obr_y+obr_fr_y;
fr_szer=obr_fr_szer; fr_wys=obr_fr_wys;
this.repaint();
```

2)
Do pobrania tablicy pikseli ze zmiennej typu `Image` służy klasa `PixelGrabber`, a właściwie jej metoda `grabPixels()`. Deklarujemy obiekt tej klasy związany ze zmienną `obrazek`. Musimy też przygotować tablicę liczb naturalnych. Metoda pobierająca tablice pikseli wymaga przechwycenia zgłaszanych w niej wyjątków, stąd konstrukcja `try – chatch`. Metoda ta zwraca jednowymiarową tablicę **`int`**, które są liczbami

[19] W Javie kolory kodowane są 32 bitami, w przeciwieństwie do WinAPI, gdzie korzysta się z 24 bitów. Dodatkowy czwarty bajt wykorzystany jest na parametr alpha – przezroczystość. System kodowania nazywa się ARGB (alpha+RGB)

32-bitowymi (4-bajtowymi). Problem może wiązać się z najstarszym bitem (ostatnim z lewej), który koduje znak. W ten sposób wartość liczby jest kodowana w 31 bitach, co zmniejsza zakres absolutny o połowę. Znającym C++ odpowiadam od razu na cisnące się na usta pytanie: w Javie nie ma niestety `unsigned int`.

```
int piksele[]=new int[obr_fr_szer*obr_fr_wys];
PixelGrabber pg=new PixelGrabber(obrazek, obr_fr_x, obr_fr_y, obr_fr_szer,
                                 obr_fr_wys, piksele, 0, obr_fr_szer);
try
  {
  pg.grabPixels();
  }
catch(Exception exc)
  {
  label1.setText(exc.getMessage());
  }
```

3) – 5)
Należy rozłożyć zakodowany w liczbie naturalnej kolor na jego składowe ARGB. Następnie należy część RGB zamienić na jego binarne dopełnienie i złożyć całość z powrotem w liczbę naturalną. Wiedza ta może się z pewnością przydać przy pisaniu programów korzystających z grafiki. Natomiast w drugiej części paragrafu pokażę jak można cały kod znacznie uprościć w przypadku tak prostego algorytmu jakim jest negatyw.

Metoda 1:
Składowe koloru kodowane są w liczbie naturalnej według następującego algorytmu[20]:
$256^3 \cdot A + 256^2 \cdot R + 256 \cdot G + B$.
Z tego wynika, że każda składowa zajmuje oddzielny bajt, a cały bajt zakodowany jest (przynajmniej na maszynie 32-bitowej) w jednym słowie procesora (w czterech bajtach). Kolejne bajty (czytane od najstarszego, czyt. od lewej) to A (alpha), R (czerwony), G (zielony) i B (niebieski). Jak odczytać je z całej liczby. Można wykonać oczywiście serię odejmowań i dzieleń, ale znacznie prościej operować na bitach.
Skorzystamy z poznanego wcześniej operatora binarnego `&`[21]. Aby odczytać najmłodszy bajt *B* należy nałożyć na czterobajtową liczbę maskę przepuszczającą jedynie ów najmłodszy bajt i odczytać jego wartość tzn.

```
int blue  = (piksele[indeks]      ) & 255;
```

Kolejny bajt koduje większe liczby. Można nałożyć na niego maskę 0xFF00 (tzn. czytać jedynie drugi bajt), a następnie odczytany wynik podzielić przez 256. Można również przesunąć bity o osiem pozycji w prawo (jeden bajt w prawo) i nałożyć znowu bajt na najmłodszy bajt:

```
int green = (piksele[indeks] >>  8) & 255;
```

Analogicznie przesuwamy kolejne bajty i odczytujemy ich wartości:

```
int red   = (piksele[indeks] >> 16) & 255;
int alpha = (piksele[indeks] >> 24) & 255;
```

Musimy teraz odwrócić kolory:

```
red=255-red;
green=255-green;
blue=255-blue;
```

i zakodować je z powrotem w liczbę naturalną. Mamy do wyboru zastosowanie operacji na bitach

```
(alpha << 24) | (red << 16) | (green << 8)  | blue;
```

lub bezpośrednie zastosowanie powyższego podstawowego wzoru[22]:

[20] Pomijam na razie problem związany z kodowaniem znaku przez najstarszy bit bajtu zajętego przez *A*.
[21] Komentarz znajduje się w paragrafie „Dynamiczne tworzenie obiektu”.

```
blue+(256)*green+(256*256)*red+(256*256*256)*alpha;
```

Komentarza wymagają użycie operatora | w pierwszym przypadku i przekroczenie zakresu liczb `int` w drugim przypadku. Operator | jest operatorem logicznym lub (OR), co oznacza, że w wyniku działania *liczba1* | *liczba2* uzyskamy liczbę, w której zapalone są bity na pozycjach, na których przynajmniej w jednej z obu liczb był zapalony bit. W tym kontekście operator ten oznacza więc sumowanie, a operatory przesunięcia odpowiadają mnożeniom przez odpowiednie potęgi 256. Korzystając z operatorów bitowych nie musimy się martwić o to, co poszczególne bity oznaczają. W szczególności nie musimy się martwić o kodowanie znaku. W drugim przypadku, jak się okazuje, też nie musimy się martwić, gdyż w Javie przypisanie do `int` liczby większej od jej zakresu oznacza zakodowanie młodszych bitów tak jak powinny być ułożone w `long`. Zamazujemy więc ostatni bit wartością zgodną z naszymi oczekiwaniami. Nie powinno nas martwić, że wartość koloru będzie dla niektórych wartości alpha ujemna, bo Java przy odczytywaniu koloru też ignoruje interpretację znaku (w innym przypadku funkcje Javy kodowały by kolor w tablicy `long`).

Operacje powyższe wykonamy w pętli po wszystkich elementach tablicy `piksele[]`.

Metoda 2:
Druga metoda korzysta z operatora jednoargumentowego dopełnienia binarnego, czyli zwyczajnego operatora negacji ~ (NOT). Operator ten zmienia w każdym bicie wartość na przeciwną (jedynki na zera, a zera na jedynki). W przypadku RGB sprawa byłaby zupełnie prosta, wystarczyłoby wartość piksela zanegować, żeby uzyskać kolor negatywu. Tu jest to jednak trudniejsze, bo negacja obejmowałaby również najstarszy bajt kodujący przezroczystość A. Ponieważ domyślnie przezroczystość ma wartość 255 (zupełnie nieprzezroczysty), to w wyniku uzyskalibyśmy zupełnie przezroczysty negatyw, którego nie byłoby oczywiście widać. Dlatego oddzielimy wpierw osiem najstarszych bitów i pozostałą część korzystając z wyżej opisanych metod:

```
int A=piksele[indeks] & 0xff000000;
int RGB=piksele[indeks] & 0x00ffffff;
```

Wykorzystaliśmy tutaj zapis szesnastkowy dla maski, w której zapalone są jedynie bity najstarszego bajtu `0xff00000000 = BIN(11111111 00000000 00000000 00000000) = 4278190080` oraz dla zapisania liczby z zapalonymi odwrotnie bitami[23]. Liczba A nie jest tutaj liczbą z zakresu 0 – 255, gdyż jest liczbą czterobajtową z zapalonymi liczbami w najstarszym bajcie. Uzyskalibyśmy odpowiednią wartość alpha dzieląc przez $256^3 = 16777216$, ale aby zakodować z powrotem ARGB musielibyśmy ją i tak pomnożyć przez 256^3.

Aby otrzymać kolor negatywu wykonujemy operację negacji binarnej na kolorach RGB, a następnie składamy w ARGB:

```
RGB=~RGB; //negatyw, alpha pozostaje bez zmian
piksele[indeks]=A | (RGB & 0x00ffffff);
```

Znowu musimy zastosować odczyt z maską, gdyż negacja odwróciły 32 bity, a więc także czwarty zerowy (dzięki zastosowaniu maski przy odczycie) bajt.
Ostatnia linia mogłaby mieć również postać:

```
piksele[indeks]=A + (RGB & 0x00ffffff);
```

6)
Ostateczny wynik zapisujemy do zmiennej negatyw typu `Image`. Musimy więc zadeklarować na początku klasy zmienną tego typu:
`Image negatyw=null;`.

---

[22] W Javie, podobnie jak w C++, nie ma operatora potęgowania ^ lub ** stąd wielokrotne mnożenie. Można oczywiście użyć funkcję `Math.pow(`*`podstawa, wykładnik`*`)`, co wymaga zadeklarowanie wykorzystania biblioteki `java.math.*`.
[23] Liczby te są swoimi negacjami (dopełnieniami) binarnymi tj. `0xff000000=~0x00ffffff`.

Do przepisania tablicy `int` do zmiennej `Image` służy metoda `Applet.createImage()`. Argumentem tej metody jest obiekt klasy `MemoryImageSource` tworzony z tablicy `int` wymagający zaimportowania biblioteki `java.awt.image`:

```
negatyw=this.createImage(
  new MemoryImageSource(fr_szer, fr_wys, piksele, 0, fr_szer));
```

Aby zobaczyć efekt wymuszamy na końcu tej metody odświeżenie formy:

```
this.repaint();
```

Dodatkowe komentarze dotyczące tej metody można znaleźć w źródłach:
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad5.zip.

6. Wymuszenie załadowania dokumentu HTML

Tworzymy nowy aplet o nazwie `Link_Applet` zawierający obiekty `textField1` (klasy `java.awt.TextField`), `choice1` (`java.awt.Choice`) i `button1` (`java.awt.Button`). Zawartość okienka edycyjnego ustalamy na dowolny adres internetowy (np. „http://java.sun.com” lub „http://localhost/”), Edytor własności nie pozwala na wypełnienie pól comboboxów, więc elementy wyboru do `choice1` musimy dodać ręcznie na początku metody `init()`:

```
public void init()
  {
  choice1.add("_self");
  choice1.add("_parent");
  choice1.add("_top");
  choice1.add("_blank");

  //ciag dalszy metody init();
```

Z przyciskiem „Obejrzyj” wiążemy metodę zdarzeniową do której zapisujemy następujący kod:

```
void button1_actionPerformed(ActionEvent e)
  {
  //URL wymaga java.net.*
  URL url = null;
  try
    {
    url = new URL(textField1.getText());
    }
  catch(MalformedURLException exc)
    {
    showStatus("Blad: "+exc.getMessage());
    return;
    }

  if (url != null)
```

```
    {
    //AppletContext appletcontext=getAppletContext();
    //appletcontext.showDocument(url,choice1.getSelectedItem());
    this.getAppletContext().showDocument(url,choice1.getSelectedItem());
    //inne: _self, _blank, _parent, _top lub nazwa ramki
    }
  }
```

Pierwsza część metody tworzy obiekt `URL` na podstawie adresu zapisanego w okienku edycyjnym `textField1`. Jeżeli zapisany tam adres nie jest poprawny (np. zła nazwa protokołu) w pasku stanu zostanie pokazany odpowiedni komunikat i metoda zakończy działanie.

W przeciwnym przypadku wywołana zostanie metoda obiektu `AppletContext.showDocument()`, która powoduje, że do wskazanej w drugim argumencie ramki zostanie załadowana strona o adresie wskazanym przez pierwszy argument. Możliwe wartości drugiego argumentu są identyczne jak przyjmowane przez interpreter HTML przeglądarki wartości parametru `TARGET` w znaczniku `<HREF>`. `AppletContext` jest interface'em, który opisuje środowisko uruchomienia apletu (dokument HTML, w który aplet jest wklejony i inne aplety znajdujące się na tym dokumencie).

7. Obsługa archiwów JAR i ZIP

Biblioteka obsługi wejścia/wyjścia posiada funkcje wspomagające obsługę kilku typów skompresowanych archiwów (`java.util.zip`, `java.util.jar`). Najważniejszymi z nich są oczywiście JAR (ang. *Java ARchive*) ze względu na jego rolę w rozpowszechnianiu programów Javy oraz ZIP/GZIP ze względu na swoją powszechność. Pliki JAR są zresztą również skompresowane algorytmem ZIP.

Zadaniem apletu jest wyświetlenie informacji o plikach znajdujących się w archiwum `sample.jar` lub innym wskazanych przez parametr przekazany z przeglądarki.

8. Pobieranie informacji o środowisku uruchomienia apletu

System bezpieczeństwa pozwala na pobranie niektórych informacji dotyczących systemu, w którym doszło do uruchomienia apletu. Umożliwia to metoda `String System.getProperty(String key)`. Jej argumentem jest klucz identyfikujący rodzaj informacji, a zwracaną wartością łańcuch zawierający odpowiednią informację lub „null", jeżeli podany w argumencie klucz jest nieodpowiedni.

Zdefiniujmy prywatną metodę:

```
private void readSystemInfo(TextArea textarea)
  {
    textarea.setText("Informacje o Javie:");
    textarea.append("\n   wersja Javy: "+System.getProperty("java.version"));
    textarea.append("\n   dostawca Javy: "+
                    System.getProperty("java.vendor")+" ("+
                    System.getProperty("java.vendor.url")+")");

    textarea.append("\n\nInformacje o systemie");
    textarea.append("\n   OS: "+System.getProperty("os.name")+
                    ", wersja: "+System.getProperty("os.version"));
    textarea.append("\n   typ: "+System.getProperty("os.arch"));

    textarea.append("\n\nInformacje niedostępne dla apletu:");
    textarea.append("\n   JVM: "+System.getProperty("java.vm.name")+
                    ", wersja: "+System.getProperty("java.vm.version"));
    textarea.append("\n   JRE: "+System.getProperty("java.home"));
    textarea.append("\n   nazwa użytkownika: "+System.getProperty("user.name"));
```

```
    textarea.append("\n   katalog domowy użytkownika: "+System.getProperty("user.home"));
    textarea.append("\n   bieżący katalog użytkownika: "+System.getProperty("user.dir"));
  }
```

którą możemy wywołać w metdzie `init()`. Metoda pobiera przez głowę referencję do obiektu `java.awt.TextArea`, który musimy umieścić na formie apletu (należy pamiętać, żeby własność `layout` ustawić wcześniej na `null`). Wywołanie funkcji powinno być więc następujące: `readSystemInfo(textArea1);`.

Uwaga! Jeżeli aplet ma działać w dowolnej przeglądarce należy zakomentować ostatni blok instrukcji.

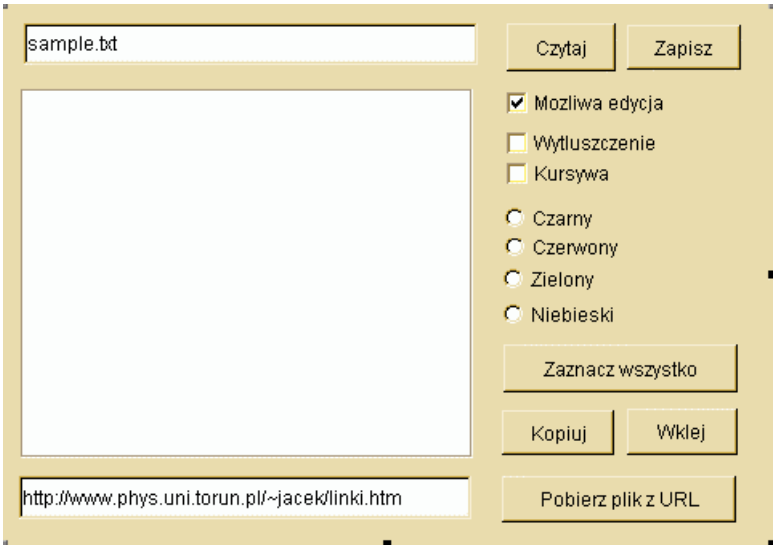
IV. Przykładowe aplety Swing

Aby uruchamiać aplety korzystające z biblioteki `javax.swing` na komputerze kliencie należy wyposażyć przeglądarkę WWW w plug-in Java 2 (JRE 1.1 i wyższe)[24]. Bez tego aplet nie zostanie załadowany, a zamiast tego na pasku stanu pojawi się komunikat „load: *nazwa_pakietu.nazwa_apletu* can't be instantiated".

Nie wnikając w głębszą ideologię swingujących kontrolek przejdźmy do projektowania przykładowych apletów.

1. Notatnik

Tworzymy nowy aplet (Uwaga! Aby zapewnić „łagodne przejście" pierwszy aplet opieramy jeszcze na standardowej klasie `java.applet.Applet`). Jako parametr uruchomienia podajemy klucz `Tekst` związany ze zmienną `parTekst`. Przeglądarka będzie mogła ustalić początkowy tekst jaki ma być wyświetlany w notatniku.



Przygotujmy nową wersję apletu notatnik korzystającą z komponentów biblioteki Swing (zakładki Swing i Swing Containers JBuildera). Okno projektujemy analogicznie jak na poniższym rysunku.

Na formie umieszczamy dwa okienka edycyjne `javax.swing.JTextField`. W `jTextField1` ustalamy własność `text` na sample.txt, w `jTextField2` na dowolny adres sieciowy (poprzedzony nazwą protokołu). Następnie w umieszczamy `javax.swing.JScrollPane` z zakładki Swing Containers, a na nim `javax.swing.JTextPane`. To zapewni nam okno edycyjne z paskami przewijania w przypadku, gdy cały tekst zajmuje obszar większy niż rozmiar pola tekstowego. Po prawej stronie appletu rozmieszczamy przyciski w podobnym ułożeniu jak w przypadku Notatnika AWT.
Ponieważ RadioButtony mają należeć do jednej grupy (tylko jeden z nich może być jednocześnie zaznaczony) powinny być zgrupowane za pomocą komponentu `javax.swing.ButtonGroup`, który zrzucamy na aplet. Obiekt, kóry powstanie (`buttonGroup1`) nie posiada graficznej reprezentacji. Radiobuttony grupujemy ustalając wartość ich własności `buttonGroup`.

Do wczytania tekstu z pliku, zarówno w przypadku pliku tekstowego, jak i dokumentu HTML, lokalnie i z sieci, posłużymy się metodą `jTextPane1.setPage(URL)` pobierającą jako argument adres URL dokumentu i ładującą jego zawartość. Stworzenie obiektu URL wymaga zaimportowania `java.net.*`.

[24] Instalacja JBuildera (i razem z nim JDK) nie zawsze powoduje, że przeglądarka WWW na tym komputerze widzi odpowiedni plug-in. Trzeba wówczas zainstalować niezależnie JRE.

Aby notatnik pokazywał tekst podany przez przeglądarkę w parametrze `Tekst` należy odszukać linię metody `jbInit()`, która ustala tekst w polu tekstowym (jak było wcześniej wspomniane metoda `jbInit()` zawiera wszystkie ustawienia domyślne lub wybrane w okienku własności w widoku projektowania interface'u apletu (zakładka Design)). Należy zmienić ją na

```
jTextPane1.setText(parTekst);
```

Napiszmy metody reagujące na modyfikowanie stanów kontrolek sterujących:

Czytaj

```
void jButton1_actionPerformed(ActionEvent e)
  {
  try
    {
    jTextPane1.setPage(new URL(getCodeBase().toExternalForm()+
                               jTextField1.getText()));
    }
  catch(MalformedURLException exc)
    {
    showStatus("Blad: "+exc.getMessage());
    }
  catch(IOException exc)
    {
    showStatus("Blad: "+exc.getMessage());
    }
  }
```

Pobierz plik z URL

```
void jButton6_actionPerformed(ActionEvent e)
  {
  try
    {
    jTextPane1.setPage(new URL(jTextField2.getText()));
    }
  catch(Exception exc)
    {
    showStatus("Blad: "+exc.toString());
    }
  }
```

Zapisz
Metoda związana z klawiszem „Zapisz" jest kopią metody, którą napisaliśmy przy implementacji Notatnika za pomocą komponentów AWT. Zamienione zostały tylko nazwy komponentów i sposób ustalania nazwy pliku:

```
void jButton2_actionPerformed(ActionEvent e)
  {
  String nazwapliku=this.getCodeBase().getFile()+
                    jTextField1.getText()+".saved";
  try
    {
    FileOutputStream fos=new FileOutputStream(nazwapliku);
    BufferedWriter buffer=new BufferedWriter(new OutputStreamWriter(fos));
    buffer.write(jTextPane1.getText());
    buffer.close();
    fos.close();
    }
```

```
  catch(Exception exc)
    {
    showStatus("Blad: "+exc.getMessage());
    }
  }
```

Obsługa pól opcji (checkboxów) i pól wyboru (comboboxów) (`javax.swing.JCheckBox` i `javax.swing.JRadioButton`) także jest niemal identyczna jak w przypadku `Notatnik_AWT_Applet`. Wystarczy zmienić nazwę obiektu, nazwy i argumenty metod pozostają w większości przypadków identycznie (wyjątkiem jest metoda sprawdzająca czy pole opcji jest zaznaczone, tzn. zamiast `textArea1.setEnabled(checkbox1.getState());` mamy `jTextPane1.setEnabled(jCheckBox3.isSelected());`).

Znacznie prostsza jest obsługa schowka. Można teraz skorzystać z metod `jTextPane1.copy()` i `jTextPane1.paste()`:

```
void jButton4_actionPerformed(ActionEvent e)
  {
    jTextPane1.copy();
  }

void jButton5_actionPerformed(ActionEvent e)
  {
    jTextPane1.paste();
  }
```

2. Stoper

Wreszcie stworzymy aplet w JBuilderze, który po pierwsze zawiera napisaną przez nas klasę, a po drugie aplet oparty na `javax.swing.JApplet`.

Tworząc za pomocą kreatora klasę apletu (niech nazywa się `Stoper_Swing_Applet`) należy pamiętać, żeby jako klasę bazową wskazać `javax.swing.JApplet`. Pomijamy wszelkie parametry i przechodzimy do konfigurowania apletu w wioku Design. Na aplet rzucamy `javax.swing.JPanel` (z zakładki Swing Containers) i ustalamy go jako `ContentPane` naszego apletu (w oknie własności obiektu apletu `this`) . Aplet nie ma już własności `layout`, wszelkie tego typu ustawienia przejmuje `jPanel1`. Ustawmy więc jPane1.layout na np. FlowLayout. Zrzućmy na panel przycisk z zakładki Swing.

Teraz zajmiemy się stworzeniem własnej klasy, pochodnej względem `JButton`, która będzie miała funkcje prostego stopera.

```
class JStoper extends JButton
{
}
```

Przejmuje on całą funkcjonalność klasy `JButton`, więc można go umieścić na formie. Niestety na tym etapie JBuilder Personal w niczym nam nie pomoże. Musimy dodać komponent klasy `JStoper` samodzielnie. Aby ułatwić sobie zadanie dodajemy do apletu, a dokładniej na `jPanel1`, zwykły przycisk z zakładki Swing (`javax.swing.JButton`) – to nam wskaże miejsca, w których należy dodać coś do kodu. Przeglądając klasę zobaczymy na jej początku deklarację przycisku `JButton jButton1 = new JButton();`.
Dodajemy analogiczną deklarację:

```
JStoper jStoper1 = new JStoper();
```

W `jbInit()` znajdują się dwie linie:

```
jButton1.setText("jButton1");
jPanel1.add(jButton1, null);.
```

Dodajemy analogiczne linie:

```
jStoper1.setText("jStoper1");
jPanel1.add(stoper1, null);
```

Po uruchomieniu apletu zobaczymy po prostu dwa przyciski[25]. Na zakładce „Design" stoper jest widoczny jedynie jako czerwony kwadrat symbolizujący komponent wizualny.

Z wnętrza klasy `JStoper` widzimy wszystkie własności i metody zadeklarowane w klasie `JButton` jako **`protected`** lub **`public`**. Niewidoczne są metody i własności prywatne (**`private`**). W każdym przypadku lepiej jest jednak nadal posługiwać się metodami dostępowymi `set...()` i `get...()` choćby dlatego, że zawierają obsługę błędów.

W tworzonej klasie można wykorzystać wyrażenie **`this`** na określenie bieżącej klasy oraz **`super`** na określenie klasy bazowej.

W przypadku braku konstruktora Java tworzy konstruktor domyślny pozbawiony argumentów. `JButton` posiada również konstruktor bez argumentów, który może być wówczas wywołany. Napisanie własnego konstruktora uniemożliwia automatyczne stworzenie konstruktora domyślnego przez kompilator[26]. Większość reguł dotycząca tworzenia konstruktorów jest identyczna jak w C++.

Napiszmy przykładowy konstruktor ustalający kolory przycisku:

```
public JStoper()
  {
  this.setBackground(Color.yellow);
  this.setForeground(Color.black);
  }
```

W Javie, posiadającej mechanizm czyszczenia pamięci z nieużywanych obiektów, nie ma destruktowów.

Naszą klasę uzupełnijmy o prywatne własności:

```
class JStoper extends JButton
  {
  private long miliseconds=0;
  private Timer timer=new Timer(250,
                          new ActionListener()
                               {
                               public void actionPerformed(ActionEvent evt)
                                    {
                                   Tick();
                                    }
                               });
  private int State=2;

  //reszta klasy
```

Pierwsza własność typu `int` będzie przechowywała ilość milisekund od momentu wywołania metody `start()`. Druga jest obiektem klasy `javax.swing.Timer`[27]. Obie (zarówno zmienna typu wbudowanego, jak i obiekt) są inicjowane w miejscu deklaracji, co w C++ jest niedopuszczalne i skończyłoby się błędem z

[25] Dla zabawy można zmienić na chwilę bazową klasę Stopera na np. `JCheckbox`.
[26] W przypadku dziedziczenia z klasy bazowej posiadającej konstruktor przyjmujący argumenty należy zadbać o wywołanie kontruktora klasy bazowej **`super`**(*argumenty*)`;` z konstruktora bieżącej klasy. Inaczej pojawi się błąd kompilatora stwierdzający brak domyślnego konstruktora klasy bazowej.
[27] Istnieje jeszcze klasa `java.util.Timer`, ale i ona dostępna jest dopiero JRE obsługującym Java 2.

komunikatem „Cannot initialize class member here”. Ostatnia własność `State` typu `int` będzie przechowywała stan stopera (0 – zatrzymany, 1 – uruchomiony, 2 – zatrzymany, ale nie zresetowany).

Klasa `Timer` posiada konstruktor przyjmujący dwa argumenty: pierwszy podaje co ile milisekund ma być kreowane zdarzenie, a drugi jest obiektem klasy `ActionListener`, który posiada jedną funkcję `actionPerformed(ActionEvent)`, której argumentem jest owo zdarzenie. Jako reakcję na „tyknięcie” zegara wywołujemy z `actionPerformed` metodę `Tick()`.
Metoda `Tick()` jest niezwykle prosta, jeżeli stan zegara równy jest 1, to zwiększamy ilość milisekund o 1 i aktualizujemy etykietę na przycisku.

```
private void Tick()
    {
    if (State!=1) return;
    miliseconds++;
    this.setText(""+miliseconds+" ms");
    };
```

Przy wywołaniu metody `setText()` zastosowano sztuczkę – przez dodanie na początku argumentu pustego łańcucha, operator dodawania jest traktowany jako łączenie łańcuchów (można również użyć `String.valueOf()` na `miliseconds` lub klasy opakowującej `Integer.toString()`).

Aby ułatwić sobie sterowanie klasą zdefiniujemy kilka małych metod odpowiadających za poszczególne czynności, które można wykonywać na stoperze. Myślę, że ich budowa nie wymaga dodatkowych wyjaśnień.

```
private void Start()
    {
    timer.start();
    State=1;
    };

  private void Stop()
    {
    timer.stop();
    State=2;
    this.setText("Reset ("+this.getText()+")");
    };

  private void Reset()
    {
    this.setText("Start");
    State=0;
    miliseconds=0;
    };
```

Nasz stoper może więc już „tykać” i posiada funkcje typowe dla zwykłego stopera. Pozostaje jedynie przekazać kontrolę nad nim użytkownikowi. Zdefiniujmy metodę `Pressed()`, która będzie przełączała stan stopera i podłączmy ją do zdarzenia naciśnięcia przycisku stopera. Postać metody `Pressed()` może być tylko jedna:

```
private void Pressed()
    {
    switch(State)
      {
      case 0: Start(); break;
      case 1: Stop(); break;
      case 2: Reset(); break;
      }
    }
```

Pozostaje przypisać tą funkcję do naciśnięcia klawisza stopera. Do konstruktora musimy dodać wywołanie metody `JButton.addActionListener()`:

```
this.addActionListener(new java.awt.event.ActionListener()
                                {
                                public void actionPerformed(ActionEvent e)
                                      {
                                      Pressed();
                                      }
                                });
```

Zauważmy, że żadna zdefiniowana przez nas metoda nie jest publiczna. Nowe funkcje przycisku są więc całkowicie niezależne od działań programisty wykorzystujący klas. Należy jednak pamiętać, że przycisk posiada pełną funkcjonalność zwykłej klasy `JButton` – można w programie zmienić wszystkie jego funkcję, także zamienić metodę zdarzeniową związaną z naciśnięciem i w ten sposób zablokować działanie stopera.

3. Zegar

Zastosujmy jeszcze raz komponent `javax.swing.Timer`, żeby przygotować zegar korzystający z `javax.swing.JLabel`. Klasa zegara posiada elementy, które poznaliśmy przy okazji stopera:

```
class JZegar extends JLabel
{
private Timer timer=new Timer(1,
                              new ActionListener()
                                 {
                                 public void actionPerformed(ActionEvent evt)
                                      {
                                      Calendar cal = Calendar.getInstance();
                                      Date date = cal.getTime();
                                      DateFormat dateFormatter =
                                             DateFormat.getDateTimeInstance();
                                      setText(dateFormatter.format(date));
                                      }
                                 });

public JZegar()
  {
  super();

  timer.start();
  }
}
```

Klasy `Calendar` i `Date` wymagają zaimportowania pakietów `java.util.*`, `Timer`: `javax.swing.Timer`, a klasa pomagająca w formatowaniu daty – `java.text.DateFormatter`:

```
import javax.swing.Timer;
import java.util.*;
import java.text.DateFormat;
```

Aby umieścić obiekt klasy `JZegar` na aplecie należy go zadeklarować (`JZegar jZegar1 = new JZegar();`), a w `jbInit()` dodać go do okna apletu:

```
this.getContentPane().add(jZegar1, BorderLayout.NORTH);
```

Podobnie jak w poprzednim przykładzie z poziomu JBuildera można modyfikować te ustawienia `JZegara`, które odziedziczył po `JLabel`. Niektóre formatowania można również modyfikować w konstruktorze. Aby zwiększyć czcionkę wystarczy do konstruktora dopisać:

```
this.setFont(new Font(this.getFont().getName(),
                      this.getFont().getStyle(),25));
```

Jeżeli aplet ma być pokazywany w przeglądarce warto również dodać do metody `jbInit()` polecenie zmieniające tło na białe:

```
this.getContentPane().setBackground(Color.white);
```

Alternatywnie można tekst sformatować własnoręcznie zastępując powyższą metodę actionPerformed Timera na następującą:

```
public void actionPerformed(ActionEvent evt)
      {
      Calendar cal = Calendar.getInstance();

      setText(""+cal.get(Calendar.DAY_OF_MONTH)+"-"+
                 cal.get(Calendar.MONTH)+"-"+
                 cal.get(Calendar.YEAR)+"  "+
                 cal.get(Calendar.HOUR_OF_DAY)+":"+
                 cal.get(Calendar.MINUTE)+":"+
                 cal.get(Calendar.SECOND));
     }
```

3. Otwieranie okna modalnego

Aplety mogą wywoływać własne okna modalne (jeżeli pozwala na to polityka bezpieczeństwa).

Tworzymy aplet (`ShowImage_Swing_Applet`). W widoku projektowania dodajemy do apletu jeden przycisk nie zmieniając layout (powinien pojawić się przy górnej krawędzi apletu). Ręcznie dodajemy właściwość klasy typu Image o odkrywczej nazwie obrazek (`private Image obrazek = null;`).

Tworzymy metodę zdarzeniową dla przycisku:

```
void jButton1_actionPerformed(ActionEvent e)
  {
  final JFileChooser fc = new JFileChooser();
  fc.setCurrentDirectory(new File("."));
  fc.setDialogTitle("Wybierz plik graficzny");
  int returnVal = fc.showOpenDialog(this);
  File file = fc.getSelectedFile();
  File directory = fc.getCurrentDirectory();
  if (returnVal == JFileChooser.APPROVE_OPTION)
    {
    try
      {
      obrazek=this.getImage(fc.getCurrentDirectory().toURL(),
                            fc.getSelectedFile().getName());
      JOptionPane.showMessageDialog(this,
             "Otwieram plik: "+fc.getSelectedFile().getName()+
             " z katalogu: "+fc.getCurrentDirectory().toString());
      repaint();
      }
    catch(MalformedURLException exc)
      {
      JOptionPane.showMessageDialog(this,"Blad: "+exc.getMessage()+".");
      }
```

```
    }
    else
    {
    JOptionPane.showMessageDialog(this,
                              "Otwarcie pliku anulowane przez użytkownika.");
    }
  }
```

Pierwsza część metody służy do pobrania nazwy pliku graficznego za pomocą odpowiedniego okna dialogu. W tym celu tworzony jest obiekt klasy `JFileChooser` i wywoływana jego metoda `showOpenDialog()`. Następnie do zmiennych file i directory, obie typu File, pobierane są nazwa pliku i nazwa katalogu. Służą one do wczytania obrazka metodą `Applet.getImage()` do zmiennej globalnej (w klasie) `obrazek`. W razie błędów, do komunikacji z użytkownikiem wykorzystywane są okna dialogowe tworzone metodą `JOptionPane.showMessageDialog`.

Pozostaje stworzenie własnej metody `paint()` rysującej na oknie apletu pobrany do zmiennej `obrazek` obraz:

```
public void paint(Graphics g)
    {
    super.paint(g);
    if (obrazek!=null)
      {
      g.drawImage(obrazek, 0, jButton1.getHeight(), this.getWidth(),
                  this.getHeight()-jButton1.getHeight(), null);
      }
    }
```

`Graphics.drawImage` jest jedną z metod klasy `Graphics`, które zostaną szczegółowo omówione w następnym rozdziale. Tutaj tylko wymienię jej argumenty. Po kolei są to: zmienna typu Image, x, y, szerokość, wysokość, `ImageObserver` – obiekt informowany o konieczności odmalowania rysunku.
Metoda wymaga dodania bibliotek `java.io.*` oraz `java.net.*`.

Zadanie
Przygotować i wykorzystać w oknie dialogowym filtry plików ograniczające wybór do typowych plików graficznych.

8. Kreator Aplikacji

Wybieramy z menu File pozycję New, następnie Application. W pierwszym kroku, podobnie jak w przypadku apletu wskazujemy na pakiet oraz na nazwę klasy (rad5 oraz ShowImage_Application). Tym razem nie mamy wyboru co do klasy bazowej aplikacji ponieważ klasa główna nie dziedziczy z żadnej klasy, a jej głównym zadaniem jest stworzenie okna aplikacji (por. z paragrafem II.7). W drugim kroku kreatora podajemy parametry klasy będącej oknem głównym (domyślnie jedynym) naszej aplikacji (również nie mamy wyboru – klasą bazową aplikacji tworzonej przez JBuildera może być tylko `javax.swing.JFrame` – taka aplikacja wymaga siłą rzeczy JRE zgodnego z Java 2). Można ręcznie zamienić klasę bazową na `java.awt.Frame` i trzymać się AWT, ale to wyklucza użycia menu itp. Decydując się na korzystanie z możliwości oferowanych przez bibliotekę Swing możemy stworzyć pasek menu (menu bar), pasek narzędzi (toolbar), pasek stanu (status bar) oraz okienko informacji (About box). Można również zażądać wyśrodkowania okna na ekranie. Zaznaczmy wszystkie opcje poza paskiem narzędzi.

Do projektu dodane zostaną trzy pliki java: ShowImage_Application.java, ShowImage_Application_Frame.java oraz ShowImage_Application_Frame_AboutBox.java. Nas będzie interesował jedynie plik zawierający klasę okna głównego aplikacji (ShowImage_Application_Frame.java), w którym chcemy uzupełnić menu o pozycję wyboru i wczytywania pliku. W tym celu przechodzimy do widoku projektowania i klikamy dwukrotnie na `jMenuBar1` w oknie struktury dokumentu. (Jeżeli w kreatorze aplikacji pominęliśmy menu – teraz możemy je

dodać z zakładki Swing Containers.) Pojawi się edytor menu, w którym za pomocą menu kontekstowego dodajemy pozycję do menu File. Nadajmy jej etykietę „Otwórz plik graficzny". W oknie własności dla tej pozycji przechodzimy na zakładkę Events i tworzymy metodę zdarzeniową dla actionPerformed. Metoda jest niemal identyczną kopią odpowiedniej metody z poprzedniego przykładu. Różnica pojawia się tylko w linii pobierającej obrazek ponieważ obiekt `Application` nie ma dostępnej metody `getImage()`:

```
if (returnVal == JFileChooser.APPROVE_OPTION)
    {
    obrazek=(new ImageIcon(fc.getSelectedFile().getPath())).getImage();
    JOptionPane.showMessageDialog(this,
             "Otwieram plik: "+fc.getSelectedFile().getPath());
    repaint();
    }
    else
    {
    JOptionPane.showMessageDialog(this,
                            "Otwarcie pliku anulowane przez użytkownika.");
    }
```

Nieco zmienia się też metoda `paint()`:

```
public void paint(Graphics g)
    {
    super.paint(g);
    if (obrazek!=null)
      {
      int yoffset=this.getInsets().top+1;
      int xoffset=this.getInsets().left+1;
      g.drawImage(obrazek,
                  xoffset,
                  yoffset+jMenuBar1.getHeight(),
                  this.getWidth()-2*xoffset,
                  this.getHeight()-2*yoffset,
                  null);
      }
    }
```

Edytor menu głównego
Przechodząc do widoku projektowania kliknijmy dwukrotnie na pozycji Menu w „drzewie zawierania komponentów". Należy rozwinąć podmenu File i korzystając z menu kontekstowego dodać pozycję menu (Insert Menu Item). Za pomocą okienka własności należy ustawić własność `Text` na „Open graphics file". Na zakładce Events tworzymy metodę związaną z `actionPerformed`.

Uruchamianie aplikacji
Nie kryje się tu żadna pułapka: `java rad5.ShowImage_Application` względnie `javaw rad5.ShowImage_Application`.

Uwaga!
O tym jak stworzyć uruchamialny plik JAR można przeczytać w rozdziale „Jak ...?".

V. Grafika

Rysowanie na aplecie możliwe jest dzięki obiektowi typu `Graphics`, który związany jest z tym apletem (jest jego własnością (*data member*)). Klasa `Graphics` posiada wiele metod służących do rysowania na „płótnie" apletu, między innymi znajduje się tam metoda `drawLine()` pozwalająca rysować linie, metody pozwalające ma rysowanie prostych figur geometrycznych (także z wypełnieniem), metody `drawPolygon()` i `drawPolyline()` rysujące wielokąty i linie łamane oraz metody sterujące kolorami, wyświetlające łańcuchy (`drawString()`), opisana wcześniej metodą `drawImage()` służąca do wyświetlania plików graficznych i wiele innych.

Warunkiem trwałości rysunków przygotowanych przez programistę jest ich odświeżanie (np. wówczas gdy aplet został przesłonięty i odsłonięty). Wszelkie polecenia rysowania na aplecie najlepiej umieszczać wobec tego w metodzie `paint()`. Mamy wówczas gwarancję, że zostaną wywołane przy każdym odświeżeniu apletu. Sami możemy wymusić odświeżenie wywołując bezargumentową metodę `repaint()`. Ta metoda wywołuje asynchronicznie metodę apletu `update(Graphics)`, o której więcej w kolejnych paragrafach, i dopiero ona wywoła `paint(Graphics)`. Nie należy jawnie wywoływać ani `update()`, ani `paint()` – jedynie poprzez wywołanie `repaint()`.

Nazwa metody	Nazwa metody (arg. jak w draw)	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 `drawLine()` cztery argumenty to współrzędne punktów początku i końca linii.

1. Rysowanie linii metodą `drawLine`

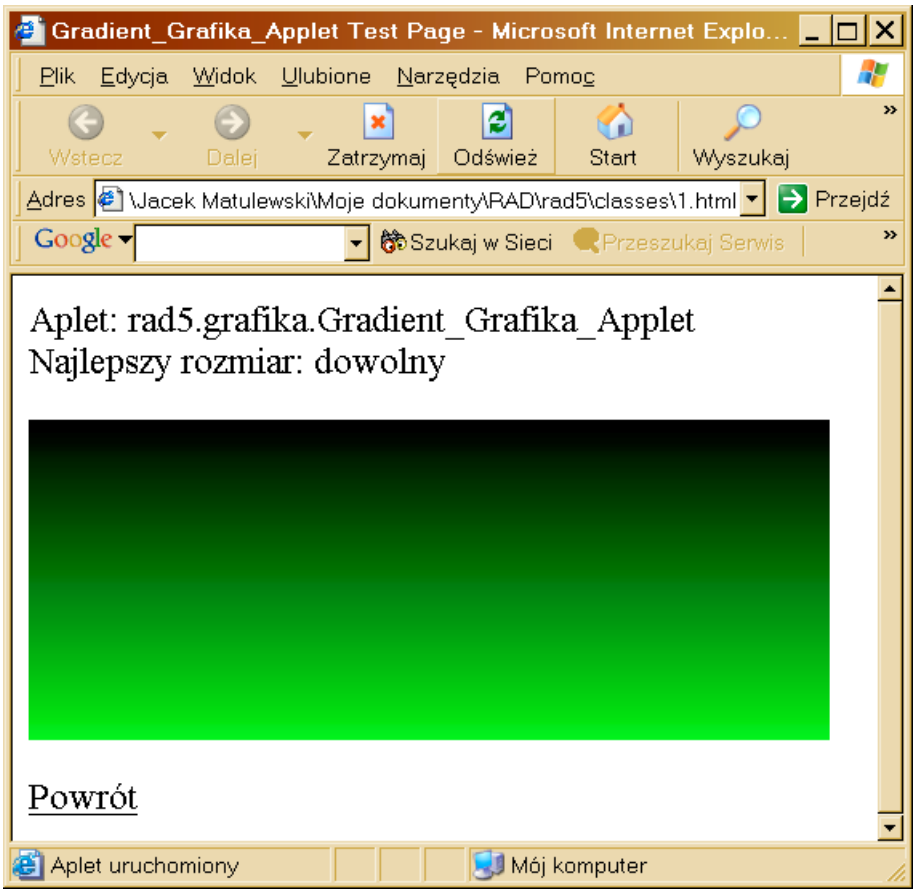
Rysowanie linii jest podstawowym elementem grafiki. Aby zilustrować wykorzystanie metody `drawLine` przygotujemy aplet, w którym rysowane obok siebie linie będą stopniowo zmieniać kolor tworząc miły dla oka efekt.

Za pomocą kreatora apletu stwórzmy nowy aplet `Gradient_Grafika_Applet` w pakiecie `rad5.grafika` (w katalogu `src\rad5` powstanie podkatalog `grafika`, w którym zostanie, a przynajmniej powinno być umieszczone źródło nowego apletu). Warto zaznaczyć opcję tworzące metody standardowe, aby móc wywołać w nich `repaint()` – to spowoduje, że nasz gradient kolorów będzie odświeżany przy wejściu na stronę lub jej odświeżeniu.
Do klasy dodajmy następującą metodę `paint()`:

```
public void paint(Graphics g)
  {
  int br=(int)(0.5+Math.random()); //0 lub 1
  int bg=(int)(0.5+Math.random()); //0 lub 1
  int bb=(int)(0.5+Math.random()); //0 lub 1
  for(int indeks=0;indeks<=this.getBounds().height;indeks++)
      {
```

```
        int kolor=255*indeks/this.getBounds().height;
        g.setColor(new Color(br*kolor,bg*kolor,bb*kolor));
        g.drawLine(0,indeks,this.getBounds().width,indeks);
        }
    }
```

Efekt jest bardzo ładny – uzyskujemy płynną zmianę koloru od czerni do jednego z kilku losowo wybranych kolorów. Pierwsze trzy linie ustalają wartości zmiennych `br`, `bg` oraz `bb` na 0 lub 1. Następnie pętla wypełnia aplet poziomymi liniami zmieniając w każdym kroku kolor pióra.
Aplet dostosowuje się do przeznaczonego mu przez przeglądarkę obszar korzystając z własności `Applet.getBounds().width` oraz `.height`.



2. Rysowanie figur geometrycznych

Kolejny aplet będzie ilustracją wykorzystania metod rysujących figury geometryczne wymienione w powyższej tabeli. Za pomocą kreatora apletu tworzymy nowy aplet o nazwie `DrawDemo_Grafika_Applet` w pakiecie `rad5.grafika`. Na aplet kładziemy komponenty AWT: `java.awt.List`, `java.awt.Checkbox` oraz `java.awt.Choice` podobnie jak na rysunku.

Aby wypełnić je odpowiednimi wartościami definiujemy funkcję `jmInit()`, którą wywołujemy w `Init()` jako ostatnią:

```
private void jmInit()
    {
    list1.add("Prostokat \"prostokatny\"");
    list1.add("Prostokat zaokraglony");
    list1.add("Prostokat pseudo-3D");
    list1.add("Owal");
    list1.add("Luk");

    choice1.add("Czarny");
    choice1.add("Niebieski");
    choice1.add("Blekitny");
    choice1.add("Ciemno szary");
    choice1.add("Szary");
    choice1.add("Zielony");
    choice1.add("Jasno szary");
    choice1.add("Fioletowy");
    choice1.add("Pomaranczowy");
    choice1.add("Rozowy");
    choice1.add("Czerwony");
    choice1.add("Bialy");
    choice1.add("Zoltry");

    list1.select(2);
    choice1.select(8);
    }
```

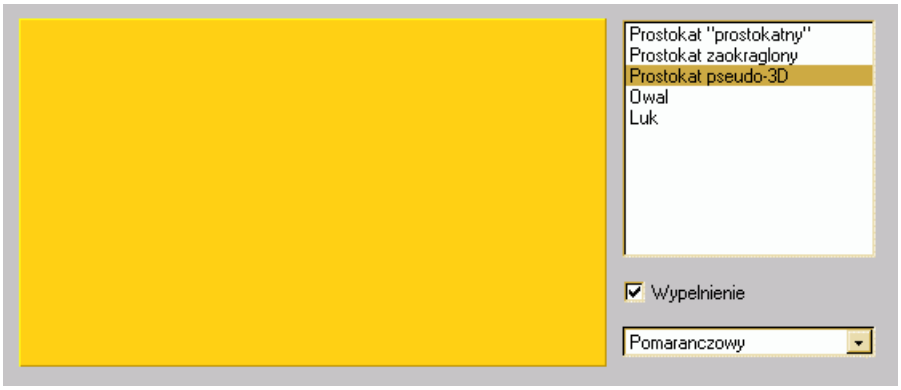
Jak zwykle rysowanie odpowiedniej figury umieścimy w metodzie `paint()`. Musimy przy tym uwzględnić odpowiedni kolor korzystając z instrukcji wyboru `switch` uzależnionej od wartości `choice1.getSelectedIndex()` oraz wybór kształtu zależny `list1.getSelectedIndex()`. Musimy również sprawdzić wartość pola opcji (`checkbox1.getState()`) aby zdecydować czy figura ma być wypełniona czy rysować tylko kontur.
Bardzo ważną rzeczą w apletach zawierających elementy AWT lub Swing jest zadbanie o wywołanie metody `paint()` klasy bazowej. Jeżeli tego nie zrobimy elementy te mogą być źle wyświetlane i źle odświeżane.

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

    int x=10;
    int y=10;
    int w=list1.getBounds().x-2*x;
    int h=choice1.getBounds().y+choice1.getBounds().height-y;

    Color kolor=null;
    switch(choice1.getSelectedIndex())
      {
      case 0: kolor=Color.black; break;
      case 1: kolor=Color.blue; break;
      case 2: kolor=Color.cyan; break;
      case 3: kolor=Color.darkGray; break;
      case 4: kolor=Color.gray; break;
      case 5: kolor=Color.green; break;
      case 6: kolor=Color.lightGray; break;
      case 7: kolor=Color.magenta; break;
      case 8: kolor=Color.orange; break;
      case 9: kolor=Color.pink; break;
      case 10: kolor=Color.red; break;
```

```
      case 11: kolor=Color.white; break;
      case 12: kolor=Color.yellow; break;
      }
    g.setColor(kolor);

    if (checkbox1.getState())
      {
      switch(list1.getSelectedIndex())
        {
        case 0: g.fillRect(x,y,w,h); break;
        case 1: g.fillRoundRect(x,y,w,h,w/10,h/10); break;
        case 2: g.fill3DRect(x,y,w,h,true); break;
        case 3: g.fillOval(x,y,w,h); break;
        case 4: g.fillArc(x,y,w,h,95,200); break;
        default: showStatus("Wybierz pozycje w liscie ksztaltow");
        }
      }
      else
      {
      switch(list1.getSelectedIndex())
        {
        case 0: g.drawRect(x,y,w,h); break;
        case 1: g.drawRoundRect(x,y,w,h,w/10,h/10); break;
        case 2: g.draw3DRect(x,y,w,h,true); break;
        case 3: g.drawOval(x,y,w,h); break;
        case 4: g.drawArc(x,y,w,h,95,200); break;
        default: showStatus("Wybierz pozycje w liscie ksztaltow");
        }
    }
  }
```

Aby kształt zmieniał się po zmianie zaznaczonego elementu w komponentach sterujących należy do ich zdarzenia `itemStateChanged` podłączyć wspólną metodę, która wymusi odświeżenie apletu. W tym celu w widoku projektu zaznaczamy listę, a następnie w oknie własności, w zakładce zdarzeń klikamy dwukrotnie na zdarzeniu `itemStateChanged`. Do powstałej w ten sposób metody wpisujemy:

```
void list1_itemStateChanged(ItemEvent e)
  {
  showStatus("Wybrany ksztalt: "+list1.getSelectedItem()+
             ", wybrany kolor: "+choice1.getSelectedItem());
  repaint();
  }
```



3. Klasa Strzalka

W animacjach wygodnie jest czasem rysować linię podając współrzędne punktu początkowego oraz długość i kierunek. Żeby to umożliwić zdefiniujemy klasę `Strzalka`, która będzie konstruowana z takimi właśnie argumentami, a poza tym umożliwi dodanie grotu do linii, czyli pozwoli na rysowanie typowego wektora.

W menu File JBuildera wybierzmy New Class... – zbudujmy nowy publiczny obiekt należący do pakietu `rad5.grafika` o nazwie `Strzalka` bez metody `main` i bez konstruktora domyślnego. Klasa zawiera własności określające punkt zaczepienia `xpocz` i `ypocz`, długość i kierunek `dlg` i `kat` oraz wartość logiczną `grot` decydującą o tym, czy należy rysować grot wektora czy jedynie linię. Wszystkie te zmienne będą ustalane w konstruktorze poprzez wywołanie serii funkcji `set...`. Konstruktor wywołuje funkcję `oblicz`, która oblicza końcowe współrzędne linii korzystając ze zmiennej `dlg_rad` przechowującej kąt wektora w radianach. Pozycja końcowa jest oczywiście niezbędna, aby wykreślić linię za pomocą `drawLine`, poza tym informacje te są udostępniamy na zewnątrz klasy (metodami `get_xkonc()` i `get_ykonc()`), co ułatwia np. „zaczepienie" jakiegoś obiektu graficznego na końcu wektora.

```
public class Strzalka
{
final int grot_rozchylenie=20; //w stopniach
final int grot_skrocenie=5;

private int xpocz,ypocz;
private int xkonc=0,ykonc=0;
private int dlg,kat; //kat w stopniach
private double kat_rad;
private boolean grot;

private void set_pocz(int axpocz,int aypocz)
  {
  xpocz=axpocz;
  ypocz=aypocz;
  };
private void set_dlgkat(int adlg,int akat)
  {
  dlg=adlg;
  kat=akat;
  };
private void set_grot(boolean agrot)
  {
  grot=agrot;
  }

private void oblicz()
  {
  kat_rad=2*Math.PI*(kat-90)/360.0; //kat w radianach od pionu
  xkonc=xpocz+(int)(dlg*Math.cos(kat_rad));
  ykonc=ypocz+(int)(dlg*Math.sin(kat_rad));
  }


//konstruktor z parametrami (nie ma konstruktora domyslnego)
public Strzalka(int axpocz,int aypocz,int adlg,int akat,boolean agrot)
  {
  set_pocz(axpocz,aypocz);
  set_dlgkat(adlg,akat);
  set_grot(agrot);
  oblicz();
  }
}
```

Wewnątrz klasy zadeklarowane są także dwie stałe określające wielkość i rozchylenie grota `grot_rozchylenie` i `grot_skrocenie`. Druga zmienna określa ile razy grot ma być krótszy niż długość wektora, a więc jego wielkość będzie proporcjonalna do długości.
Aby klasa była w pełni użyteczna należy dodać metodę rysującą reprezentację graficzną wektora na wskazanym obiekcie typu `Graphics`. Narysowanie głównej linii nie jest oczywiście problemem. Więcej wysiłku wymaga określenie parametrów dwóch linii grota. 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.

```
public void rysuj(Graphics g)
  {
  //glowna linia
  g.drawLine(xpocz,ypocz,xkonc,ykonc);

  if (grot) //grot
    {
    //prawa linia
    int grotdlg=dlg/grot_skrocenie;
    double grot_roz_rad=2*Math.PI*grot_rozchylenie/360.0;
    double grotkat_rad=kat_rad+Math.PI+grot_roz_rad;
    int xgrotkonc=xkonc+(int)(grotdlg*Math.cos(grotkat_rad));
    int ygrotkonc=ykonc+(int)(grotdlg*Math.sin(grotkat_rad));
    g.drawLine(xkonc,ykonc,xgrotkonc,ygrotkonc);
    //lewa linia
    grotkat_rad=kat_rad+Math.PI-grot_roz_rad;
    xgrotkonc=xkonc+(int)(grotdlg*Math.cos(grotkat_rad));
    ygrotkonc=ykonc+(int)(grotdlg*Math.sin(grotkat_rad));
    g.drawLine(xkonc,ykonc,xgrotkonc,ygrotkonc);
    }
  }
```

Jak zapowiedziałem zdefiniujemy metody publiczne zwracające współrzędne punktu końcowego:

```
public int get_xkonc()
  {
  return xkonc;
  }
public int get_ykonc()
  {
  return ykonc;
  }
```

Podobnie dla kierunku i długości wektora:

```
public int get_dlg()
  {
  return dlg;
  }
public int get_kat()
  {
  return kat;
  }
```

Warto również dodać metody pozwalającą na obrót wektora o zadany w stopniach kąt i zmianę jego długości:

```
public void obrot(int dkat)
  {
  set_dlgkat(dlg,kat+dkat);
  oblicz();
```

```
  }
public void zmiana_dlg(int nowa_dlg)
  {
  set_dlgkat(nowa_dlg,kat);
  oblicz();
  }
```

Tak zdefiniowana klasa może być wykorzystywana we wszystkich apletach pakietu bez żadnych ograniczeń. Aby zilustrować jej działanie naszej klasy stwórzmy aplet `StrzalkaDemo_Grafika_Applet`. Umieśćmy na aplecie dwa paski przewijania `java.awt.Scrollbar` oraz pole opcji `java.awt.Checkbox`, którymi będziemy kontrolować długość i kierunek oraz rysowanie grota. Pasek odpowiadający za określenie kąta powinien przyjmować wartości od 0 do 360.
Metoda `paint()` jest niezwykle prosta:

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

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

    //strzalka
    Strzalka s=new Strzalka(320/2,250/2,
                            scrollbar1.getValue(),scrollbar2.getValue(),
                            checkbox1.getState());
    s.rysuj(g);
    }
```

Wytłumaczenia wymaga dlaczego nie tworzę obiektu typu `Strzalka` globalnie w całej klasie. W przypadku większego obiektu lub obiektu, którego inicjacja wiąże się z jakimiś obliczeniami jego deklaracja wewnątrz metody `paint()` jest niekorzystna, ponieważ wiąże się ze stratą czasu. Ale w naszym przypadku nie ma to wielkiego znaczenia, ponieważ stworzenie strzałki nie zajmuje zbyt wiele czasu. Nie warto również przechowywać tego obiektu dla przechowania jego parametrów, ponieważ te są w pełni określone przez pozycje pasków przewijania.

Aby aplet był czuły na zmianę pozycji suwaka na paskach przewijania należy stworzyć metodę (może być wspólna dla obu pasków) związaną z ich zdarzeniem `adjustmentValueChanged`, która będzie wywoływała tylko `repaint()`. Podobnie należy zrobić dla odpowiedniego zdarzenia pola opcji.

4. Wahadło

Wykorzystamy teraz klasę `Strzalka` do przygotowania apletu ilustrującego ruch i rozkład sił w wahadle matematycznym. W pierwszym podejściu wykorzystamy obiekt `javax.swing.Timer` przez co aplet będzie wymagał maszyny wirtualnej zgodnej z Java 2, ale potem, korzystając z osobnego wątku do odmierzania czasu, pozbędziemy się tego warunku.

Na aplecie umieśćmy pasek przewijania, który posłuży nam do kontrolowania początkowego wychylenia wahadła, jego zakres powinien obejmować kąty od –90 do 90. Dodajmy także dwa przyciski „Zwolnij" i „Pauza" oraz pole opcji (`java.awt.Checkbox`), którym w następnych paragrafach będziemy sterowali buforowaniem obrazu. Ich położenie nie musi być szczególnie dobrane, ponieważ i tak dopasujemy ich rozmiar i położenie do wielkości apletu w metodzie `start()`.

Ręcznie dodajmy do klasy apletu następujące własności:

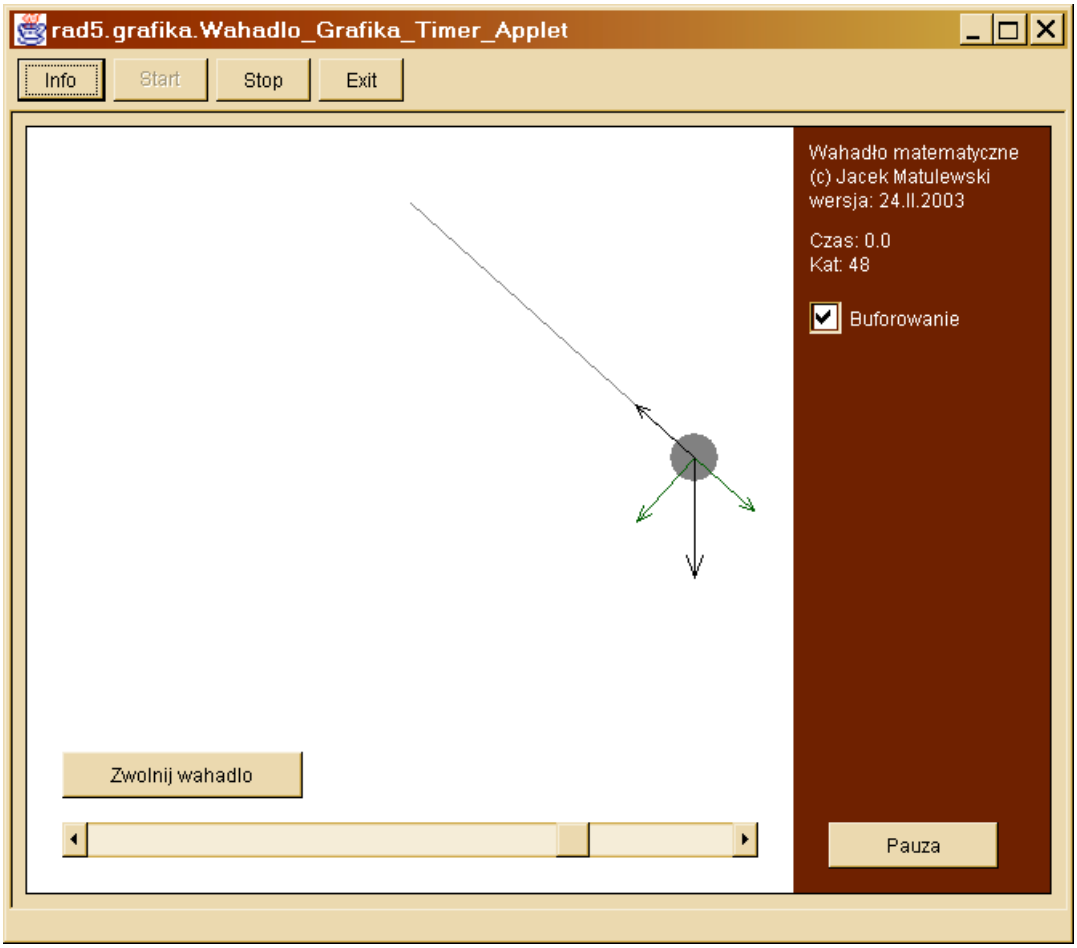
```
double t=0; //czas biezacy
double l=0.1; //dlugosc wahadla (regulacja okresu)
final double g=9.81; //przyspieszenie ziemskie
Timer timer=null; //obiekt odmierzajacy uplyw czasu
boolean ruch=false; //flaga: recznie czy ruch automatyczny
final int krok_czasowy=10; //w milisekundach
int kat=0; //kat wychylenia wahadla (w stopniach)
boolean start_flag=true; //flaga dbajaca o jednokrotne ustawienie kontrolek
static Color ciemny_czerwony = new Color(100,0,0); //kolor ciemnoczerwony
static Color ciemny_zielony = new Color(0,100,0); //kolor ciemnozielony
```

a) Dostosowanie wyglądu apletu do wielkości okna

Aby aplet dostosowywał się do wielkości okna, część poleceń, które określają pozycję i wielkość kontrolek sterujących należy przenieść z `jbInit()`, względnie `Init()` do `start()`, ponieważ dopiero tam znane są rzeczywiste rozmiary przydzielone apletowi przez przeglądarkę.

```
public void start()
  {
  if (start_flag)
    {
    start_flag=false;

    int width=this.getBounds().width;
    int height=this.getBounds().height;
    int wys=22; //wysokosc paska przewijania i przycisku
    int szer=150; //szerokosc przycisku

    scrollbar1.setBounds(new Rectangle(wys, height-2*wys,
                                       3*width/4-2*wys, wys));
    button1.setBounds(new Rectangle(wys, height-4*wys, szer, 4*wys/3));
    button2.setBounds(new Rectangle(3*width/4+wys, height-2*wys,
                                    szer-2*wys, 4*wys/3));
    checkbox1.setBounds(new Rectangle(3*width/4+10, 105,
                                      szer-20, 4*wys/3));
    }
  }
```

Po ustaleniu wysokości i szerokości apletu za pomocą `this.getBounds().width` i `height` (podobnie działa `this.getWidth()` i `this.getHeight()`) rozmieszczamy przyciski, pasek przewijania oraz pole opcji stosując ich metodę `setBounds()` w taki sposób, żeby pasek przewijania znajdował się przy dolnej krawędzi ekranu, nad nim przycisk „Zwolnij", a po prawej – przycisk „Pauza". Prawa część apletu (mniej więcej 1/4 szerokości) będzie zajęta przez panel z opisem i polem opcji „Buforowanie" oraz wspomnianym przyciskiem „Pauza".



Metoda `start()` może być wywoływana wiele razy przez przeglądarkę, stąd warunek pozwalający na ustawienie pozycji kontrolek, tylko jeden raz, przy pierwszym uruchomieniu metody `start()`. Odpowiada za to zmienna `start_flag`.

a) Wykorzystanie klasy Strzalka

Kolejnym krokiem, najważniejszym przy projektowaniu apletów bazujących na grafice, jest napisanie metody `paint()`. Jeżeli flaga ruch jest wyłączona, do zmiennej `kat` pobierana jest wartość określana przez pasek przewijania (należy pamiętać o prawidłowym określeniu jego zakresu wartości). Do zmiennych `size_x` i `size_y` pobierana jest wielkość obszaru przeznaczonego na ruch wahadła.
Następnie rysujemy ciemnoczerwony[28] panel z prawej strony i umieszczamy na nim metodą `Graphics.drawString()` opis apletu.
Potem rysujemy kolejno linkę wahadła korzystając z obiektu typu `Strzalka` bez grota, kulkę oraz cztery wektory działających na wahadło sił i ich składowych:

a) siła ciężkości $\vec{Q}$ skierowany pionowo w dół
b) jej składowa wzdłuż linki $-\vec{N}$ i przeciwnie do niej skierowana siła $\vec{N}$, z którą na kulkę działa linka
c) składowa prostopadła do linki $\vec{F}$.

[28] Wśród predefiniowanych w klasie `Color` kolorów brakuje ciemnoczerwonego, dlatego definiujemy go sami jako statyczną własność klasy naszego apletu pod nazwą `ciemny_czerwony`.

```
public void paint(Graphics g)
  {
  super.paint(g);
  if (!ruch) kat=scrollbar1.getValue(); //przy recznym sterowaniu
  //jezeli ruch automatyczny kat ustalany przez timer
  Rectangle r=this.getBounds();
  int size_x=3*r.width/4;
  int size_y=r.height;

  //panel
  g.setColor(ciemny_czerwony);
  g.fillRect(size_x,0,r.width-size_x,r.height);
  int napisy_x=size_x+10;
  g.setColor(Color.white);
  g.drawString("Wahadło matematyczne",napisy_x,20);
  g.drawString("(c) Jacek Matulewski",napisy_x,35);
  g.drawString("wersja: 24.II.2003",napisy_x,50);

  //napisy: czas i kat
  String t_str=String.valueOf(t);
  t_str=t_str.substring(0,Math.min(5,t_str.length()));
  g.drawString("Czas: "+t_str,napisy_x,75);
  g.drawString("Kat: "+kat,napisy_x,90);

  //linka
  g.setColor(Color.gray);
  int x0=size_x/2;
  int y0=size_y/10;
  int l=5*size_y/10;
  Strzalka linka=new Strzalka(x0,y0,l,180-kat,false);
  linka.rysuj(g);

  //kulka
  int kulka_x0=linka.get_xkonc();
  int kulka_y0=linka.get_ykonc();
  int kulka_r=l/15;
  g.fillOval(kulka_x0-kulka_r,kulka_y0-kulka_r,2*kulka_r,2*kulka_r);

  //wektory
  int Q_dlg=75;
  Strzalka Q=new Strzalka(kulka_x0,kulka_y0,Q_dlg,180,true);

  int F_dlg=-(int)(Q_dlg*Math.sin(2*Math.PI*kat/360));
  Strzalka F=new Strzalka(kulka_x0,kulka_y0,F_dlg,90-kat,true);

  int N_dlg=-(int)(Q_dlg*Math.cos(2*Math.PI*kat/360));
  Strzalka N1=new Strzalka(kulka_x0,kulka_y0,-N_dlg,-kat,true);
  Strzalka N2=new Strzalka(kulka_x0,kulka_y0,N_dlg,-kat,true);

  g.setColor(ciemny_zielony);
  F.rysuj(g);
  N2.rysuj(g);

  g.setColor(Color.black);
  Q.rysuj(g);
  N1.rysuj(g);
  }
```

Aplet będzie najlepiej wyglądał, gdy jego wysokość i szerokość będą w stosunku 3:2.

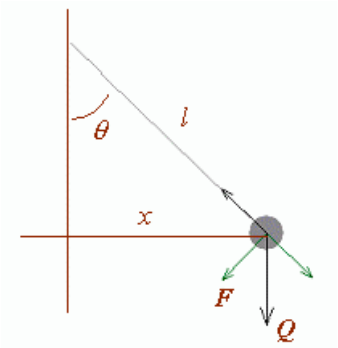
b) Timer

W chwili naciśnięcia klawisza „Zwolnij” pasek przewijania i naciśnięty klawisz zostaną ukryte, a wahadło rozpocznie ruch z wyznaczonymi przez użytkownika warunkami początkowymi (wychylenie ustalone za pomocą paska przewijania bez prędkości początkowej).

Ponieważ składowa siły ciężkości równoległa do linki zrównoważona jest przez siłę napięcia tej linki, to siła wypadkowa, która jest odpowiedzialna za przyspieszenie wahadła równa jest składowej siły ciężkości prostopadłej do linki. W przybliżeniu małych kątów (dla kątów mierzonych w radianach) jej długość równa jest:

$$F(t) = mg\sin\theta(t) = mg\frac{x(t)}{l},$$

gdzie *x* to chwilowe wychylenie, a *l* długość linki (zob. rysunek poniżej).



W efekcie otrzymujemy równanie na wychylenie postaci:

$$\ddot{x}(t) + \frac{g}{l}x(t) = 0,$$

którego rozwiązaniem dla naszych warunków początkowych ($x(0) = x_0$, $v(0) = 0$) jest funkcja periodyczna $x(t) = x_0\cos(\omega t)$ o częstości $\omega = \sqrt{g/l}$. Okres drgań jest więc równy $T = 2\pi/\omega = 2\pi\sqrt{l/g}$. Kąt w funkcji czasu jest więc równy (w przybliżeniu małych kątów):

$$\Theta(t) = \frac{x(t)}{l} = \frac{x_0}{l}\cos(\sqrt{\frac{g}{l}}t)$$

Oczywiście rozważania te są w pełni poprawne tylko dla kątów o bezwzględnej wartości mniejszej niż $\pi/4 = 45^\circ$, ale gdy zastosujemy go dla całej dziedziny tj. dla kątów $-\pi/2 < \Theta < \pi/2$ różnice w zachowaniu wahadła nie będą wyczuwalne.

Oznaczmy przez $\Theta_0 = x_0/l$ kąt początkowy odpowiadający momentowi zwolnienia wahadła. Rozwiązanie można przepisać jako:

$$\Theta(t) = \Theta_0\cos(\sqrt{\frac{g}{l}}t).$$

W tej postaci nie ma znaczenia, czy kąty mierzone są w radianach czy w stopniach. Okres wahadła najlepiej kontrolować dobierając jego długość.

Zapiszmy kąt wychylenia wahadła w momencie naciśnięcia klawisza „Zwolnij" do zmiennej lokalnej `kat0` typu `int` (kąt w stopniach może być przechowywany jako liczba naturalna bez wielkiego uszczerbku dla dokładności ustalenia kierunku wektora). Utworzymy obiekt typu `javax.swing.Timer` W bibliotekach Java dostępne są dwa obiekty tego typu: `java.util.Timer` oraz `javax.swing.Timer`. Obydwa obiekty są podobne (oba dostępne są dopiero w wyższych wersjach JDK), lecz obsługa `Timera` z biblioteki Swing jest wygodniejsza. W docelowej postaci apletu obiekt ten i tak zostanie zastąpiony przez konstrukcję dostępną w wersjach Javy domyślnie umieszczanych w przeglądarkach (bez instalowania pluginów). Obiekt `javax.swing.Timer` uruchamiany jest z dwoma argumentami: pierwszy informuje o odstępie czasowym, między kolejnymi „impulsami", a drugi jest obiektem klasy `ActionListener` posiadającym zdefiniowaną metodę `actionPerformed()`. To właśnie ta metoda będzie wywoływana periodycznie po upływie czasu określonego przez pierwszy argument. W naszym przypadku metoda ta będzie dbała o obliczenie nowych wartości zmiennych globalnych `kat` i `t`, a następnie wywoływać będzie zdefiniowaną powyżej metodę `paint()` przez wywołanie `repaint()` (jak wspomniano wcześniej nie należy jawnie wywoływać `paint()`).

Metoda związana z naciśnięciem klawisza „Zwolnij" (`button1`) powinna być następująca (stworzona jako metoda zdarzeniowy w typowy dla JBuildera sposób):

```
void button1_actionPerformed(ActionEvent e)
  {
  final int kat0=scrollbar1.getValue(); //wychylenie poczatkowe

  scrollbar1.setVisible(false);
  button1.setVisible(false);
  ruch=true;

  //javax.swing.Timer
  timer=new Timer(krok_czasowy,
                  new ActionListener()
                    {
                    public void actionPerformed(ActionEvent evt)
                      {
                      t+=0.001*krok_czasowy; //co milisekunde
                      kat=(int)(kat0*Math.cos(t*Math.sqrt(g/l)));
                      repaint();
                      }
                     });
  timer.start(); //uruchomienie watku timera
  }
```

Aby ta metoda zadziałała konieczne jest zaimportowanie obiektu `javax.swing.Timer`.

Pozostaje jeszcze oprogramowanie klawisza „Pauza", ale to, przy wykorzystaniu `Timera` z biblioteki Swing jest proste – wystarczy zatrzymać `timer`:

```
void button2_actionPerformed(ActionEvent e)
  {
  if (timer==null) return; //wyjscie, jeżeli timer nie jest zainicjowany
  if (timer.isRunning()) timer.stop(); else timer.start();
  }
```

c) Buforowanie – przedefiniowanie metody update

Działanie apletu nie może być zadawalające ze względu na migotanie związane z częstym odświeżaniem apletu[29] (częstym wywoływaniem metody `paint()`). Można oczywiście zwiększyć krok czasowy, a co za tym zmniejszyć częstość migotania – aby oszukać oko wystarczy odświeżanie z częstością ok. 60Hz. Zwiększając krok czasowy można zauważyć, że okres wahadła się zmniejsza, choć kod napisany jest w taki sposób, żeby czas był niezależny od częstości odświeżania – nie jest to błąd programu. Po prostu większość czasu zabiera apletowi wykonywanie metody `paint()`, co uzależnia go też nieco od szybkości komputera. Im komputer będzie szybszy, tym lepiej okres odświeżania będzie zgadzać się z zadanym krokiem czasowym. Istotne jest również środowisko uruchomienia apletu, w JBuilderze aplet działa wolniej niż w przeglądarce.
Niestety zmiana częstości odświeżania nie wpływa znacząco na poprawę jakości animacji apletu. Aby usunąć migotanie najlepiej zastosować technikę podwójnego buforowania. Jej istota polega na tym, żeby nie rysować bezpośrednio w pamięci ekranu (tj. w naszym przypadku na obiekcie `Graphics` związanym z apletem), tylko na obiekcie zastępczym tego samego typu i po przygotowaniu rysunku, skopiować go na aplet. Poprawa jakości animacji wiąże się z faktem, że kopiowanie obrazu trwa niezwykle krótko (przenoszenie danych między dwoma rejonami pamięci) w porównaniu z czasem potrzebnym na przygotowanie tego obrazu.

Najprościej zrealizować ten pomysł podmieniając metodę `update(Graphics)` w naszym aplecie. Standardowa metoda `Applet.update(Graphics)` składa się z dwóch poleceń: czyszczenia apletu[30] i wywołania metody `paint(Graphics)`[31]. Możemy odtworzyć jej działanie przesłaniając metodę `update()`:

```
public void update(Graphics g)
  {
  Rectangle dim=this.getBounds();
  int w=dim.width, h=dim.height;

  g.clearRect(0,0,w,h);
  paint(g);
  }
```

Zmodyfikujemy tę metodę tak, żeby wykorzystanie podwójnego buforowania było uzależnione od stanu pola opcji `checkbox1`.

```
Image bufor=null;
public void update(Graphics g)
  {
  Rectangle dim=this.getBounds();
  int w=dim.width, h=dim.height;

  boolean podwojne_buforowanie=checkbox1.getState();

  if (podwojne_buforowanie)
    {
    if (bufor==null) bufor=this.createImage(w,h); //wykonane tylko raz
    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 (calosc)
    this.paint(bg); //tu maluje do bufora
    g.drawImage(bufor,0,0,null); //maluje bufor na aplecie
    bg.dispose(); //zwalniam bufor
    }
    else
    {
```

[29] Po prawdzie, to gdyby metoda `paint()` rysowała jedynie wahadło, bez panelu – migotanie nie byłoby szczególnie uciążliwe. Panel został tutaj dodany głównie jako pretekst do buforowania.
[30] Nie zawsze opłaca się czyścić i malować cały aplet. Można, wzorem twórców filmów animowanych, czyścić i malować jedynie jego mniejszy ruchomy fragment.
[31] Jeżeli pominąć czyszczenie uzyskamy efekt nakładania się kolejnych klatek filmu.

```
    g.clearRect(0,0,w,h);
    paint(g);
    }
  }
```

Deklarujemy referencję do obiektu `Image` o nazwie `bufor`. Robimy to poza metodą `update`, żeby nie spowalniać działania apletu, przez wielokrotne rezerwowanie pamięci. Deklaracja znajduje się poza klasą, ale inicjacja musi odbywać się po wywołaniu `init()` ponieważ konieczna jest znajomość prawdziwej wielkości apletu. Dlatego przypisanie obiektu do referencji umieszczone zostało wewnątrz `update()` dbając o to, żeby polecenie inicjacji wykonane było tylko jeden raz. Następnie tworzymy obiekt `bg` typu `Graphics`. `Graphics` jest klasą abstrakcyjną, dlatego stworzyliśmy wpierw `Image`, który posiada metodę `Image.getGraphics()` tworzącą kontekst graficzny, który przypisujemy do `bg`. Czyścimy zawartość `bg` i wykonujemy metodę `paint()` podając jako argument `bg` i wreszcie zawartość bufora rysujemy na aplecie poleceniem `drawImage()`. W końcu zwalniamy pamięć zarezerwowaną dla obiektu bufora.

Po skompilowaniu i uruchomieniu apletu okaże się, że jeżeli pole opcji „Buforowanie” jest zaznaczone – aplet zupełnie przestaje migotać[32].

d) Klasa BufferedApplet

Dzięki nadpisaniu metody `update()` do zaimplementowania buforowania obrazu zbędne są jakiekolwiek modyfikacje metody `paint()`. Można więc pomyśleć taką klasę, która dziedzicząc z `java.applet.Applet`, będzie posiadała przeciążoną metodę `update()` tak, żeby aplety korzystające z tego obiektu jako obiektu bazowego były automatycznie buforowane.

Klasa, nazwijmy ją `BufferedApplet` (zapiszmy ją w oddzielnym pliku, żeby łatwiej było ją wykorzystywać przez wiele apletów; należy zadbać, żeby była klasą publiczną), powinna być następująca:

```
package rad5.grafika;

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

public class BufferedApplet extends Applet
  {
  protected boolean podwojne_buforowanie=true;
  private Image bufor=null;

  public BufferedApplet()
    {
    super(); //wywolanie konstruktora klasy Applet
    }

  public void update(Graphics g)
    {
    Rectangle dim=this.getBounds();
    int w=dim.width, h=dim.height;

    if (podwojne_buforowanie)
      {
      if (bufor==null) bufor=this.createImage(w,h);
      Graphics bg=bufor.getGraphics();
      bg.clearRect(0,0,w,h);
      bg.setClip(0,0,w,h);
      this.paint(bg);
      g.drawImage(bufor,0,0,null);
      bg.dispose();
```

[32] Na wolniejszych komputerach buforowanie może zwalniać działanie apletu.

```
      }
      else
      {
      g.clearRect(0,0,w,h);
      paint(g);
      }
    }
}
```

Jak widać klasa posiada dwie własności: `podwojne_buforowanie` typu logicznego oraz `bufor` typu `Image` oraz metodę, która jest dokładną kopią, z wyjątkiem ustalenia wartości zmiennej `podwojne_buforowanie`, napisanej przez nas metody `update()`.

Teraz wystarczy zmienić klasę bazową klasy Wahadlo_Grafika_Applet na BufferedApplet i można usunąć niepotrzebną już metodę `update()` w tym aplecie.

Aby modyfikować wartość zmiennej `podwojne_buforowanie` należy ze zdarzeniem `itemStateChanged` pola opcji `checkbox1` związać metodę:

```
void checkbox1_itemStateChanged(ItemEvent e)
  {
  podwojne_buforowanie=checkbox1.getState();
  }
```

e) Klasa Metronom (tworzenie dodatkowych wątków)

Kolejną techniką niezbędną do projektowania apletów graficznych jest wykorzystanie dodatkowego wątku do cyklicznego wywoływania metody `repaint()`. Chcę tym sposobem wyeliminować wykorzystanie `Timera`, który wymaga zainstalowania pluginu Java 2 w przeglądarce.

Napiszmy prostą klasę rozszerzającą klasę wątku `java.lang.Thread`. Konstruktor klasy będzie pobierał krok czasowy, czyli ilość milisekund pomiędzy kolejnymi generowanymi „impulsami” oraz referencję do apletu, którego odświeżania ma pilnować. Obydwa argumenty będą przechowywane we własnościach klasy. Ponadto klasa będzie posiadać własność `pauza`. Najistotniejszą metodą wątku, którą koniecznie należy przesłonić, jest `run()`. Metoda ta jest uruchamiana przez wywołanie metody `start()` i jest wykonywana niezależnie w osobnym wątku. Po zakończeniu tej metody działanie wątku kończy się. Metody `run()` nie należy wywoływać bezpośrednio, bo wtedy jej zawartość zostanie wykonana w bieżącym wątku.
Metoda `run()` działa w pętli nieskończonej. Co czas wyznaczony przez parametr przekazany w konstruktorze wywołuje metodę `repaint()` podanego w konstruktorze apletu. Do odmierzania czasu najprościej wykorzystać metodę `sleep()` „usypiającą” wątek na zadany okres czasu.

```
package rad5.grafika;

import java.applet.Applet;

public class Metronom extends Thread
  {
  private int krok_czasowy=0; //milisekund
  public boolean pauza=false; //flaga pauzy
  private Applet aplet=null;

  public Metronom(Applet aaplet,int akrok_czasowy)
    {
    aplet=aaplet;
    krok_czasowy=akrok_czasowy;
    }

  public void run()
    {
```

```
    while(true)
      {
      try
        {sleep(krok_czasowy);}
      catch(InterruptedException exc)
        {break;}
      if (!pauza) aplet.repaint();
      }
    }
  }
```

Ponieważ zmienna `kat0` jest teraz inicjowana w innej metodzie niż jej późniejsze wywołania musi być zadeklarowana jako zmienna globalną dla całej klasy apletu. Jako własność apletu należy również zadeklarować referencję `demon` do klasy `Metronom` i zainicjować ją po naciśnięciu klawisza „Zwolnij" tj. należy uzupełnić metodę `button1_actionPerformed()` o następujące polecenia:

```
//stworzenie i uruchomienie watku w button1_actionPerformed
demon=new Metronom(this,krok_czasowy);
demon.start();
```

Na początku metody `paint()` należy dodać obliczanie kąta wychylenia wahadła:

```
//obliczenia kata wychylenia wahadla
if (!ruch)
  {
  kat=scrollbar1.getValue(); //przy recznym sterowaniu
  }
  else
  {
  t+=0.001*krok_czasowy;
  kat=(int)(kat0*Math.cos(t*Math.sqrt(this.g/l)));
  }
```

Warto zauważyć, że w linii obliczającej kąt przyspieszenie ziemskie `g` jest uzupełnione o wskazanie obiektu ze względu na identyczną nazwę kontekstu graficznego będącego argumentem metody `paint()`.

Zmienić musi się również realizacja pauzy. W klasie metronom jest własność pauzy. Jej wartość decyduje o wymuszeniu odświeżenia apletu. W związku z tym wystarczy jeżeli przycisk „Pauza" będzie modyfikował wartość tej własności:

```
void button2_actionPerformed(ActionEvent e)
  {
  if (ruch) demon.pauza=!demon.pauza;
  }
```

Uwaga!
Należy zadbać o to, żeby metoda `stop()` włączała pauzę, a metoda `start()` wyłączała.

```
public void stop()
    {
    if (ruch) demon.pauza=true;
    }
```

f) Sterowanie szybkością wahadła (rysowanie poza metodą `paint()`)

Najprostszy sposób, żeby modyfikować szybkość wahadła, to regulacja długości jego linki. Aby to zrealizować dodajemy do apletu obiekt `scrollbar2` z palety AWT. W metodzie `start()` dodajemy polecenia umieszczające pasek przewijania na panelu z prawej strony apletu:

```
scrollbar2.setValue((int)(sb2_length_coeff*l));
scrollbar2.setMaximum(3*(int)sb2_length_coeff+
                      scrollbar2.getBlockIncrement());
scrollbar2.setBounds(new Rectangle(3*width/4+10, 250, szer-wys, wys));
```

Do stałych zadeklarowanych w klasie apletu dodajemy

```
final double sb2_length_coeff=100;
```

która pośrednio określa zakres wartości nowego paska przewijania.
Do `paint()` należy dodać polecenie wyświetlające informacje o długości:

```
g.drawString("Dlugosc: "+l,napisy_x,235);
```

Nie spowoduje to jednak aktualizacji napisu w sytuacji, ponieważ nie zdefiniowaliśmy odpowiedniej metody zdarzeniowej wywołującej `repaint()`. Aby nie odmalowywać całego apletu w sytuacji gdy modyfikacji wymaga jedynie mały jego wycinek będziemy pisać na aplecie poza metodą `paint()`. Zdefiniujmy odpowiednią metodę zdarzeniową dla paska przewijania związanego z długością wahadła:

```
void scrollbar2_adjustmentValueChanged(AdjustmentEvent e)
  {
  double cos_arg_old=t*Math.sqrt(g/l)+faza;
  l=scrollbar2.getValue()/sb2_length_coeff;
  faza=cos_arg_old-t*Math.sqrt(g/l);

  //napis dlugosc bez wywolywania paint()
  int size_x=3*this.getBounds().width/4;
  int napisy_x=size_x+10;
  Graphics ag=this.getGraphics();
  int font_height=ag.getFont().getSize();
  ag.setColor(ciemny_czerwony);
  ag.fillRect(napisy_x,235-font_height,
              this.getBounds().width-size_x,font_height);
  ag.setColor(Color.white);
  ag.drawString("Dlugosc: "+l,napisy_x,235);
  ag.dispose();
  }
```

Przeanalizujmy po kolei jej polecenia. pierwsze trzy linie zmieniają długość wahadła. Jednak zmiana długości wahadła powoduje skokową zmianę kąta wychylenia. Aby tego uniknąć wprowadzamy do funkcji obliczającej kąt w metodzie `paint()` fazę

```
kat=(int)(kat0*Math.cos(t*Math.sqrt(this.g/l)+faza));
```

zadeklarowaną globalnie i zainicjowaną zerową wartością

```
double faza=0; //uzywana przy zmianie dlugosci
```

Podczas zmiany długości wahadła faza będzie modyfikowana w taki sposób, żeby aktualny kąt wychylenia pozostawał niezmieniony. Chcemy, aby argument kosinusa w funkcji wyznaczającej kąt przed i po zmianie długości zostawał niezmieniony:

$$\omega_0 t + \varphi_0 = \omega_1 t + \varphi_1$$

Stąd wzór na fazę zaimplementowany w pierwszej i trzeciej linii metody:

$$\varphi_1 = \omega_0 t + \varphi_0 - \omega_1 t$$

To robią pierwsze trzy linie powyższej metody.

Zadanie
Jako ćwiczenie pozostawiam uwzględnienie wpływu zmiany długości wahadła na amplitudę jego drgań.

Zgodnie z powyższą zapowiedzią aktualizacja napisu informującego o długości wahadła ma odbywać się poza metodą `paint()`, w metodzie zdarzeniowej związanej z paskiem przewijania. Aby to było możliwe musimy pobrać metodą `Applet.getGraphics()` obiekt `Graphics` apletu. Czyścimy obszar zajęty przez napis (aby zbadać jego wielkość pobieramy informacje o wysokości czcionki) i metodą `drawString()` podajemy nową długość czcionki.

Zadanie
Aplet uzupełnić o kolejne pola opcji (`checkbox`) za pomocą których można będzie decydować o rysowaniu wektorów siły i ich składowych.

Uwaga!
Pełen kod apletu i klas `Metronom` oraz `BufferedApplet` znajduje się w źródłach (adres na stronie tytułowej).

g) Przeciąganie myszką

W naszym aplecie brakuje jeszcze jednego elementu – chciałoby się ustalać początkowe wychylenie wahadła za pomocą myszki. Bardzo łatwo to uzyskać korzystając z metod `Applet.mousePressed`, `Applet.mouseDragged` (`Applet.mouseMoved` nie jest wywoływana gdy klawisz myszy jest naciśnięty) oraz `Applet.mouseReleased`. Te trzy metody wystarczają, żeby zrealizować mechanizm drag&drop w obrębie apletu lub aplikacji. W naszym przypadku sytuacja jest prosta: po naciśnięciu i myszy i jej ewentualnym przesunięciu wychylenie wahadła powinno być ustalone w taki sposób, żeby kursor myszy znajdował się na lince lub jej przedłużeniu. W tym celu napiszmy metodę zdarzeniową (wspólną dla `Applet.mousePressed` i `Applet.mouseDragged`) :

```
void this_mouseDragged(MouseEvent e)
  {
  if (ruch) return; //metoda wykonywana tylko w fazie ustawiania

  //pozycja zaczepienia wahadla
  Rectangle r=this.getBounds();
  int size_x=3*r.width/4;
  int size_y=r.height;
  int x0=size_x/2;
  int y0=size_y/10;
  int l=5*size_y/10;

  //ustalanie pozycji kursora
  int x=e.getX();
  int y=e.getY();

  double kat_rad=Math.atan((x-x0)/(double)(y-y0));
  if (y-y0<0) kat_rad=(x-x0>0)?Math.PI/2.0:-Math.PI/2.0;

  //rad->deg i przypisanie
  scrollbar1.setValue((int)Math.toDegrees(kat_rad));
```

```
this.repaint();
}
```

Metoda zaczyna się od warunku sprawdzającego, czy samodzielny ruch wahadła już się nie rozpoczął. Jeżeli tak – reszta kodu nie jest wykonywana.
W następnych liniach inicjowane są zmienne przechowujące rozmiar pola dostępnego dla rysowania wahadła i co najważniejsze jego punkt zaczepienia (zob. analogiczne w metodzie `paint()`). Dalej ustalana jest pozycja kursora.
Na podstawie względnej pozycji kursora i punktu zaczepienia można obliczyć kąt pod jakim powinno być ustawione wahadło korzystając z metody `Math.atan()`. Należy pamiętać, że zwraca kąt w radianach. Aby wrócić do wartości w stopniach najwygodniej posłużyć się metodą `Math.toDegrees()`.

Aby przekazać nową wartość do metody `paint()` najłatwiej zmodyfikować pozycję `scrollbar1`, która jest tam odczytywana i wywołać `repaint()`.

Zadanie
Możemy dodać warunek, żeby przeciąganie myszką było aktywowane jedynie, gdy myszka znajduje się nad kulką wahadła. W tym celu musimy napisać osobną metodę zdarzeniową dla `Applet.mousePressed`, która to sprawdza i przełącza odpowiednią flagę. Do tego konieczny jest dostęp do aktualnej pozycji kulki wahadła (najprościej byłoby przenieść deklarację obiektu `linka` poza metodę `paint()` i sterować jej wychyleniem stosując metodę `Strzalka.obrot()`).

Zadanie
Przygotować aplet „Zegar tarczowy” korzystając z obiektów klasy `Strzalka` do rysowania wskazówek. Pogrubienie linii można uzyskać korzystając z klasy `Graphics2D` (zob. kod źródłowy apletu `Zegar_Grafika2D_Applet.java` w dołączonych źródłach).

5. Przygotowywanie wykresów. Klasa Plot – instrukcja obsługi

Klasa `Plot` umieszczona w pakiecie `rad5.grafika` (dostępne w dołączonych źródłach) jest klasą abstrakcyjną. Nie można wobec tego stworzyć obiektu tego typu – jest tak, żeby użytkownik w klasie pochodnej mógł zdefiniować własną wykreślaną funkcję.
Klasa zawiera metodę abstrakcyjną zadeklarowaną jako

```
abstract double function(double x);
```

W klasie pochodnej musimy ją przesłonić własną funkcją.
Poniższa klasa `SinPlot` (najwygodniej stworzyć ją za pomocą kreatora klasy, należy pamiętać, aby zaznaczyć pole „Override abstract methods") pokazuje jak to zrobić, a przy okazji jest ilustracją wykorzystania metod konfigurujących klasę Plot: `setPlotRange()` definiującej zakres zmiennej widocznej na wykresie oraz `setPlotColors()` pozwalającej na dobranie kolorów osi, linii i osi zaznaczającej zero.

```
public class SinPlot extends Plot
{
  public SinPlot()
    {
    super();
    this.setPlotRange(-Math.PI,-1,Math.PI,1);
    this.setPlotColors(Color.black,Color.red,Color.white);
    }

  double function(double x)
  {
    /**@todo: implement this rad5.grafika.Plot abstract method*/
    return Math.sin(x);
  }
}
```

Dziedziczenie z klasy `Plot` jest okazją do modyfikacji zachowania klasy, np. dodania opisu osi.

VI. Jak ... ?

1. Jak skompilować aplety lub aplikacje bez JBuildera?

Przykłady umieszczone powyżej i dostępne w źródłach zostały napisane za pomocą JBuilder 6 Personal, ale można je również skompilować i uruchomić niezależnie (niezbędny jest wówczas JDK - *Java Development Kit*). W tym celu należy skopiować z wybranego projektu katalog zawierający cały pakiet ("src\rad5"), a następnie w katalogu nadrzędnym (katalog "rad5" musi być skopiowany jako całość, jego nazwa musi być zachowana) uruchomić kompilator poleceniem (niezbędne jest podanie pełnej ścieżki do javac.exe lub dopisanie jej do zmiennej środowiskowej PATH):

```
javac rad5\*.java
```

Jeżeli chcemy skompilować także pakiet `rad5.grafika` musimy wpisać polecenie:

```
javac rad5\*.java rad5\grafika\*.java
```

2. Jak umieścić aplety w archiwum JAR lub ZIP?

Należy spakować skompilowane klasy (pliki *.classes) razem z drzewem katalogów odpowiadającym nazwom pakietów do formatu JAR (`jar cf rad5\*.class rad5\grafika\*.class`) lub ZIP. Następnie w kodzie HTML osadzającym aplet na stronie WWW (przykład można znaleźć w paragrafie II.2) należy uzupełnić znacznik `<APPLET>` o własność `archive = "`*`nazwa`*`.jar"` lub `archive = "`*`nazwa`*`.zip"`.

3. Jak stworzyć uruchamialny plik JAR?

Należy przygotować prosty plik (np. "rad5Manifest") zawierający nazwę wykonywalnej klasy w jednej linii zakończonej znakiem końca linii (w naszym przypadku linia ta powinna zawierać: `Main-Class: rad5.ShowImage_Application`). Teraz można spakować potrzebne (skompilowane) klasy poleceniem:

```
jar cmf rad5Manifest ShowImage_Application.jar rad5\ShowImage_Appl*.class
```

i uruchomić je klikając w Windows Explorerze dwukrotnie myszką na plik klasy.jar lub wpisując:

```
java -jar ShowImage_Application.jar
```

4. Jak kontrolować politykę bezpieczeństwa?

Przy uruchomieniu na serwerze WWW (także na lokalnym komputerze, jeżeli strona pobierana jest przez serwer, np. IIS) kontrola dostępu apletów do plików jest zaostrzona ze względu na zasady bezpieczeństwa. W przypadku wbudowanej w system Windows (i co za tym idzie w IE) wirtualnej maszyny Java (Microsoft VM) oraz pluginów Suna starszych niż JDK 1.2 uzyskanie uprawnień do zapisu plików nie jest w ogóle możliwa – polityka bezpieczeństwa jest ustalona na stałe. Nowsze wersje JDK posiadają narzędzie

W katalogu JDK...\bin, pakietu dostarczającego maszyny wirtualnej (poprzez odpowiedni plug-in) znajduje się program policytool, który ułatwia edycję pliku `.java.policy` w katalogu użytkownika. Po pierwszym uruchomieniu program `policytool` zgłasza brak pliku `.java.policy` w katalogu domowym użytkownika – to normalne.

Po edycji należy sprawdzić, czy plik JDK..\security\java.security zawiera wskazanie na nasz plik polityki bezpieczeństwa.

5. Co można skasować z katalogu projektu?

Odpowiedzią jest plik wsadowy umieszczony w podstawowym katalogu projektu:

```
rmdir /s bak
del *.*~
del classes\rad5\*.class classes\rad5\grafika\*.class
```