helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

JBuilder X. Efektywne programowanie w Javie



Autor: Jacek Matulewski
ISBN: 83-7361-293-9
Format: B5, stron: 384

Jeśli wierzyć prognozom firmy Borland – producenta JBuildera – pod koniec obecnego dziesięciolecia prawie 80% aplikacji będzie tworzonych w Javie i bazujących na niej środowiskach. Rozmaite dziedziny ludzkiej działalności stawiają przed informatyką coraz to większe wymagania. Coraz cenniejszym zasobem staje się inwencja i produktywność projektantów-programistów, którym trzeba w jak największym stopniu umożliwić twórcze działanie. Warto ich odciążyć od drugorzędnych zadań, które z powodzeniem można powierzyć coraz lepszym i coraz tańszym maszynom. Obecnie, w dobie upowszechnienia się komputerów osobistych, trudno wyobrazić sobie profesjonalne tworzenie aplikacji bez usprawniających to przedsięwzięcie nowoczesnych narzędzi (RAD), z JBuilderem na czele. Warto pamiętać, że firma Borland jest producentem dwóch innych narzędzi RAD, które zrewolucjonizowały proces tworzenia aplikacji: Delphi i C++Buildera — kolejne wersje tych produktów obecne są na rynku od niemal dziesięciu lat.

Niniejsza książka stanowi praktyczny podręcznik zarówno dla początkujących, którzy na gruncie JBuildera zamierzają zdobywać swe pierwsze doświadczenia programistyczne, jak i tych, którzy w swojej pracy używają Delphi i C++Buildera, a w JBuilderze szukają interesującej alternatywy.

W książce omówiono między innymi:

- Podstawy języka Java i bogate mechanizmy środowiska zintegrowanego JBuildera X
- Korzystanie z nowoczesnych technik programistycznych oraz narzędzi i mechanizmów wspomagających tworzenie aplikacji i apletów
- Stosowanie bibliotek i szablonów oraz JavaBeans
- Bazodanowe zastosowania JBuildera X, mechanizmy JDataStore, JDBC i ODBC
- Tworzenie i używanie archiwów JAR
- Modelowanie i projektowanie komponentów
- Dokumentowanie procesu projektowego za pomocą JavaDoc

Nie trać czasu na coś, w czym może Cię wyręczyć komputer; zajmij się projektowaniem apletów i aplikacji.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp **9**

Rozdział 1. Pierwsze spotkanie z JBuilderem **11**

Trivia 11
Co to jest JBuilder? 11
Nie tylko JBuilder 12
Co nowego w darmowej edycji JBuilder X Foundation? 13
Zintegrowane środowisko programistyczne 14
Przewodnik po oknach i zakładkach IDE 14
Jak bezboleśnie przesiąść się z Delphi lub C++ Buildera do JBuildera? 17
Pomoc 18
Konfigurowanie edytora i innych elementów środowiska programisty 19
Tworzenie i zarządzanie projektem 25
Ustawienia formatowania kodu źródłowego 29
JBuilder w służbie programisty, czyli przez Javę na skróty 30
Kreator apletu. Pierwszy aplet „Hello World!” 30
Aby obejrzeć aplet w zewnętrznej przeglądarce… 37
Analiza kodu pierwszego apletu 40
Osadzanie apletu w dokumencie HTML 43
Kontrola działania apletu przez przeglądarkę 44
CodeInsight — budka suflera 47
Parametry odbierane z dokumentu HTML 48
Kontrola działania apletu poprzez komponenty sterujące i ich metody zdarzeniowe 52
Jak skasować niepotrzebną metodę zdarzeniową? 59
Modyfikacja konfiguracji uruchamiania 59
Kilka słów o debugowaniu 63

Rozdział 2. Java **71**

Kilka słów wstępu 71
Nowe i „stare” cechy Javy 71
Wirtualna maszyna Javy. Przenośność 72
Czy można za pomocą JBuildera…? 73
Język 73
Komentarze 74
Konwencja wielkości liter 74
Proste typy danych 75
Łańcuchy 77
Tablice 79
Instrukcje sterujące 81

Klasy i obiekty ... 96
Nie warto unikać klas i obiektów! ... 96
Projektowanie klasy ... 97
Kilka dodatkowych wiadomości o klasach w skrócie ... 130

Rozdział 3. Przykładowe aplety i aplikacje ... 135
Kolejne przykłady wykorzystania biblioteki AWT do projektowania RAD w JBuilderze ... 136
Aplet Scrollbars ... 136
Tworzenie apletu będącego równocześnie aplikacją. Pobieranie informacji o wirtualnej maszynie Javy ... 144
Biblioteka Swing (Java 2) ... 155
Swing kontra AWT ... 155
Pierwsza aplikacja Swing — Notatnik ... 156
Aplikacja JPictureView — obrazy, ikony i pasek narzędzi ... 186

Rozdział 4. Praca z kodem. Szablony ... 199
Korzystanie z szablonów podczas „ręcznego" projektowania interfejsu. Aplet Puzzle ... 199
Projektowanie RAD interfejsu vs. wykorzystanie tablic komponentów ... 199
Menedżer położenia komponentów ... 200
Ręczne przygotowanie interfejsu apletu. Tablice obiektów ... 205
„Prawie ręczne" tworzenie metod zdarzeniowych i nasłuchiwaczy ... 208
Metoda zdarzeniowa ... 211
Odmierzanie czasu w osobnym wątku. Tworzenie szablonów. Układanie puzzli na czas. ... 219

Rozdział 5. Aplikacje konsolowe w JBuilderze ... 227

Rozdział 6. Komponenty JavaBean od środka ... 231
Tworzenie komponentów JavaBean ... 231
Klasa MigajacyPrzycisk i środowisko do jej testowania ... 232
Wykorzystanie interfejsu Runnable ... 237
Właściwości komponentów ... 243
Definiowanie dodatkowych zdarzeń ... 247
Instalowanie komponentu na palecie komponentów JBuildera ... 254
Klasa typu BeanInfo towarzysząca komponentowi ... 258
BeansExpress ... 261
Właściwości ... 263
Zdarzenia ... 266
Edytor właściwości i BeanInfo ... 267
Diagram UML komponentu ... 270
Silnik komponentu (metoda run) ... 271
Inicjacja komponentu ... 273
Testowanie komponentu ... 274

Rozdział 7. Dystrybucja apletów, aplikacji i ziarenek JavaBean ... 277
Które pliki przenieść na serwer WWW, aby aplet działał? Jakie pliki należy dołączyć do aplikacji, aby ją uruchomić na innym komputerze? ... 278
Rozpoznawanie plików skompilowanych klas ... 278
Tworzenie skryptów, plików wsadowych i skrótów ... 279
„Izolowanie" klas aplikacji ... 282
Pliki JAR. Archive Builder ... 282
Umieszczanie aplikacji w archiwach JAR ... 283
Umieszczanie apletów w archiwach JAR ... 289
Komponenty JavaBean w archiwach JAR ... 290

Tworzenie pliku uruchamialnego ... 294
Tworzenie dokumentacji ... 296
Tworzenie dokumentacji za pomocą javadoc z JDK ... 297
Kreator Javadoc ... 300

Rozdział 8. Bazy danych w JBuilderze ... 305
Kilka słów wprowadzenia ... 305
Podstawowe pojęcia związane z bazami danych ... 306
Czy znajomość SQL jest niezbędna? ... 306
Lokalne bazy danych. JDBC i ODBC ... 307
Konfigurowanie DSN dla bazy danych ODBC (Windows) ... 308
Tworzenie modułu danych za pomocą Data Modeler. Składnia polecenia SELECT języka SQL. ... 309
Tworzenie aplikacji bazodanowych ... 315
Komponenty dbSwing ... 318
Samodzielne konfigurowanie połączenia ... 323
Master/Detail ... 325
Database Pilot ... 329
JDataStore. W kierunku baz danych typu klient-serwer ... 330
JDataStore Explorer ... 330
Tworzenie modułu danych ... 335
Aplikacje korzystające z bazy danych JDataStore ... 337
Zdalne bazy danych i JDataStore Server ... 346

Dodatek A Kompilacja i archiwizowanie narzędziami JDK bez JBuildera ... 351
Kompilacja ... 351
Tworzenie archiwów JAR ... 353

Dodatek B Co zabawnego można zrobić z obrazem? ... 355
Konwersja na tablicę pikseli i jej modyfikowanie na przykładzie negatywu ... 355
Konwersja obrazu na tablicę pikseli opisywanych typem int ... 356
Składowe A, R, G i B każdego piksela. Operacje na bitach. Negatyw ... 357
Konwersja tablicy pikseli do obiektu typu Image ... 360
Implementacja w JPictureView ... 360
Trochę zabawy: konwersja obrazu do ASCII Art. ... 362

Skorowidz ... 365

Rozdział 6.

Komponenty JavaBean od środka

Wykorzystamy teraz wiedzę i praktyczne umiejętności na temat Javy i JBuildera zdobyte w poprzednich rozdziałach do stworzenia komponentu JavaBean[1]. Ziarenko, które przygotujemy, będzie cyklicznie zmieniało kolory, czyli mówiąc prościej — będzie migającym przyciskiem. Tak zresztą nazwiemy nasz komponent — `MigajacyPrzycisk`.

Niestety, będziemy musieli w dużym stopniu przygotować ten komponent „ręcznie". O ile w wersji komercyjnej JBuilder zawiera narzędzia `BeansExpress` i `BeanInsight`, które wydatnie wspomagają tworzenie ziarenek JavaBean, to w wersji darmowej JBuildera działają one tylko przez 30 dni od instalacji. Mimo to, także w JBuilder Foundation, istnieje możliwość względnie szybkiego przygotowania komponentu JavaBean, o czym postaram się Czytelnika przekonać w ćwiczeniach z tego rozdziału.

W pierwszej części rozdziału przedstawię podstawowe informacje o projektowaniu komponentów w JBuilder Foundation, w tym o definiowaniu ich nowych właściwości i zdarzeń. Następnie omówię procedurę instalacji komponentów w JBuilderze, a na końcu sposób przygotowania klasy typu `BeanInfo`.

Najwłaściwszą formą dystrybucji komponentu jest archiwum JAR. Jak je tworzyć, dowiemy się w rozdziale 7.

Tworzenie komponentów JavaBean

Zaprojektujemy klasę `MigajacyPrzycisk`, która będzie prostym przykładem komponentu JavaBean. `MigajacyPrzycisk` zbudujemy na bazie komponentu `java.awt.Button` z biblioteki AWT. Jego działanie nie będzie skomplikowane — będzie to po prostu

[1] *Bean* w języku angielskim oznacza ziarno, ziarenko. W kofeinowej poetyce Javy chodzi najpewniej o ziarenko kawy.

przycisk, w którym cyklicznie zamieniane są kolory tła i napisu. Klasę `Button` rozszerzymy o nowe właściwości `okres`, `opoznienie` oraz `miganieWlaczone` umożliwiające kontrolę migania przycisku.

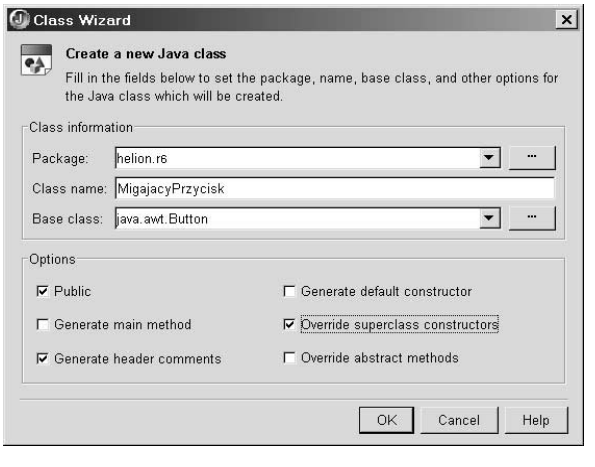
Klasa MigajacyPrzycisk i środowisko do jej testowania

Podczas projektowania komponentu wygodnie jest mieć aplet, w którym możemy go od razu testować. Dlatego zaczniemy od zdefiniowania dwóch klas, klasy komponentu `MigajacyPrzycisk` oraz klasy apletu `MigajacyPrzyciskDemo`.

Ćwiczenie 6.1.

Aby stworzyć klasę o nazwie `MigajacyPrzycisk` rozszerzającą klasę `java.awt.Button`:

1. Z menu *File* wybieramy pozycję *New Class*....
2. W kreatorze klasy (rysunek 6.1) podajemy:
   a) nazwę pakietu (pole *Package*) `helion.r6`,
   b) `nazwę klasy (pole Class name) MigajacyPrzycisk` oraz
   c) klasę bazową (*Base class*) `java.awt.Button`.



Rysunek 6.1. *Rozszerzanie klasy java.awt.Button*

3. W kreatorze klasy zaznaczamy:
   a) opcję *Override superclass constructor*, oznaczającą, że w wygenerowanym przez JBuildera kodzie nadpisane zostaną wszystkie konstruktory klasy bazowej, oraz
   b) opcję *Public*, ponieważ klasa komponentu JavaBean musi być oczywiście publiczna.
4. Naciskamy *OK*.

Po zamknięciu kreatora uzyskujemy następujący kod zaprezentowany na listingu 6.1. Ze względu na opcję zaznaczoną w punkcie 3a ćwiczenia w klasie zadeklarowane zostały dwa konstruktory obecne również w klasie bazowej: konstruktor domyślny oraz konstruktor przyjmujący jako argument treść etykiety na przycisku.

Listing 6.1. *Stworzona przez kreator klasa MigajacyPrzycisk (pomijam komentarz nagłówkowy)*

```
package helion.r6;

import java.awt.Button;
import java.awt.HeadlessException;

public class MigajacyPrzycisk extends Button
{

  public MigajacyPrzycisk() throws HeadlessException
  {
  }

  public MigajacyPrzycisk(String label) throws HeadlessException
  {
    super(label);
  }
}
```

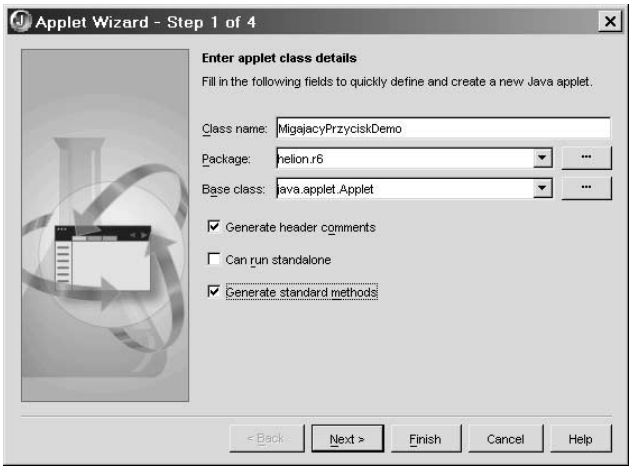
Ćwiczenie 6.2.

Aby stworzyć aplet `MigajacyPrzyciskDemo`, który będzie służył do testowania klasy `MigajacyPrzycisk`:

1. Uruchamiamy kreator apletu, a więc z menu *File* wybieramy pozycję *New*, w otwartym oknie *Object Gallery* przechodzimy na zakładkę *Web* i klikamy ikonę *Applet*. Naciskamy *OK*.
2. W kreatorze apletu (rysunek 6.2) wpisujemy:
   a) w polu *Class name* nazwę klasy apletu `MigajacyPrzyciskDemo`,
   b) w polu *Package* nazwę pakietu `helion.r6`,
   c) w polu *Base class* klasę bazową `java.applet.Applet`.
3. Zaznaczamy opcję *Generate standard methods*, aby JBuilder w kodzie apletu umieścił metody standardowe `start`, `stop` i `destroy`.
4. W ostatnim kroku kreatora zaznaczamy opcję tworzenia konfiguracji uruchamiania.
5. W komercyjnych wersjach JBuildera zmieniamy domyślną konfigurację uruchamiania na *MigajacyPrzyciskDemo* (zobacz ćwiczenie 1.18).

Efektem wykonania tego ćwiczenia jest dodanie do pakietu `helion.r6` klasy apletu `MigajacyPrzyciskDemo`, która posłuży nam jako środowisko testowania komponentu. Ponieważ klasa `MigajacyPrzycisk` należy do tego samego pakietu, będzie widoczna z apletu bez potrzeby importowania.

Rysunek 6.2.
Kreator apletu



Ćwiczenie 6.3

Aby w aplecie `MigajacyPrzyciskDemo` zdefiniować dwa obiekty typu `MigajacyPrzycisk`:

1. Dodajemy w obrębie klasy apletu dwie linie wyróżnione na poniższym listingu.

Listing 6.2. *W klasie apletu deklarujemy dwa obiekty typu MigajacyPrzycisk*

```
public class MigajacyPrzyciskDemo extends Applet
{
  private boolean isStandalone = false;

  MigajacyPrzycisk mp1=new MigajacyPrzycisk();
  MigajacyPrzycisk mp2=new MigajacyPrzycisk("Helion");

  //Dalsza część klasy apletu
```

Do stworzenia pierwszego obiektu wykorzystaliśmy domyślny konstruktor, a drugiego — konstruktor pozwalający ustalić etykietę przycisku.

Ćwiczenie 6.4.

Aby zdefiniowane w poprzednim ćwiczeniu przyciski umieścić na powierzchni apletu:

1. Umieszczamy w metodzie `jbInit` apletu wywołanie metody `this.add` dla każdego z przycisków:

```
private void jbInit() throws Exception
{
  this.add(mp1);
  this.add(mp2);
}
```

2. Naciskamy klawisz *Ctrl+F9*, aby skompilować klasę `MigajacyPrzycisk` i aplet `MigajacyPrzyciskDemo`.

3. Naciskamy klawisz *F12*, aby przenieść się na zakładkę *Design* i obejrzeć podgląd apletu z przyciskami.

Z punktu widzenia Javy nie ma znaczenia, czy polecenia z punktu pierwszego umieścimy w metodzie `init`, czy w wywoływanej z niej `jbInit` — po uruchomieniu apletu wynik będzie identyczny. Należy jednak pamiętać, że do przygotowania podglądu na zakładce *Desing* JBuilder analizuje tylko tę drugą metodę.

Warunkiem uzyskania prawidłowego podglądu apletu z umieszczonymi na nim przyciskami jest kompilacja komponentu do postaci pliku *MigajacyPrzycisk.class* (drugi punkt). Jeżeli JBuilder nie znajdzie takiego pliku, zamiast przycisków zobaczymy parę tzw. „czerwonych komponentów" (ang. *red components*). Czerwone komponenty pojawiają się w widoku projektowania wówczas, gdy JBuilder ma problemy z utworzeniem obiektów właściwych komponentów. Najczęściej powodem jest po prostu braku plików skompilowanych klas.

Ćwiczenie 6.5.

Aby — korzystając z możliwości projektowania RAD — zmienić rozmiar i położenie przycisków na powierzchni apletu:

1. Pozostajemy w widoku projektowania (zakładka *Design*).
2. W oknie struktury zaznaczamy obiekt apletu (`this`).
3. Za pomocą inspektora zmieniamy właściwość `layout` apletu na `null`. W ten sposób wyłączamy menedżera położenia komponentów, co umożliwia ustalenie dowolnego rozmiaru i położenia komponentów.
4. Korzystając z metod projektowania RAD, powiększamy przyciski na tyle, żeby widoczny był napis na jednym z nich.
5. Zaznaczamy przycisk `mp1` na podglądzie apletu (nazwę zaznaczonego przycisku można sprawdzić w oknie struktury — jest tam również zaznaczony).
6. Następnie za pomocą inspektora edytujemy jego właściwość `label` odpowiadającą za treść etykiety. Niech napis na przycisku brzmi *JBuilder*.

Obiekty przycisków `mp1` i `mp2` zachowują się identycznie jak komponenty `Button`. Tak jak w przypadku innych komponentów wizualnych możemy za pomocą myszy dowolnie zmienić ich położenie i rozmiar, a także modyfikować inne ich właściwości korzystając z inspektora.

Ćwiczenie 6.6.

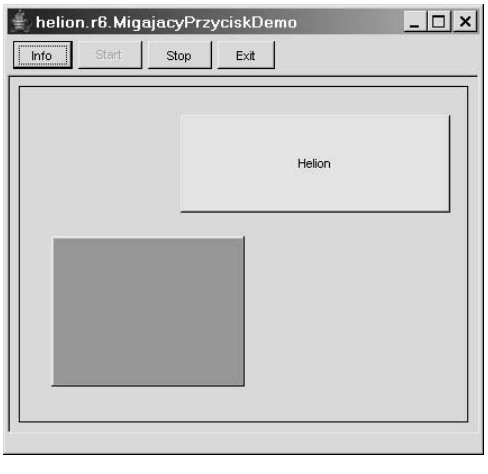
Aby upewnić się, czy komponenty umieszczone na aplecie nie są zwykłymi przyciskami typu `Button`, dodajmy na chwilę do konstruktorów klasy `MigajacyPrzycisk` polecenia zmiany koloru tła:

1. W edytorze przełączamy zakładkę plików na `MigajacyPrzycisk`.
2. Importujemy klasę `Color` dodając polecenie na początku pliku apletu przed klasą `import java.awt.Color;`.
3. Uzupełniamy klasę o umieszczone w konstruktorach polecenia zmiany tła przycisków (wyróżnione fragmenty):

```
public MigajacyPrzycisk() throws HeadlessException
{
  setBackground(Color.green);
}

public MigajacyPrzycisk(String label) throws HeadlessException
{
  super(label);
  setBackground(Color.yellow);
}
```

No i proszę. Tła przycisków na podglądzie apletu `MigajacyPrzyciskDemo` lub po uruchomieniu go w *AppletTestbed* (*F9*) zmieniły się. Możemy być więc pewni, że na powierzchni apletu rzeczywiście znajdują się dwa komponenty klasy `MigajacyPrzycisk` (rysunek 6.3).



Rysunek 6.3. *Aplet z dwoma komponentami MigajacyPrzycisk, na razie jeszcze niemigającymi, uruchomiony w przeglądarce apletów JBuildera AppletTestbed*

Ćwiczenie 6.7.

Aby sprawdzić, czy aplet działa w przeglądarce z wyłączonym pluginem Java 2:

1. Uruchamiamy zewnętrzną przeglądarkę. Jeżeli to możliwe wyłączamy plugin Java 2.
2. Wczytujemy stworzony przez JBuildera plik *MigajacyPrzycisk.html*, który znajduje się w podkatalogu *classes* projektu.
3. Po upewnieniu się, że aplet działa, wracamy do JBuildera i usuwamy polecenia zmiany koloru tła z konstruktorów.

Jak już wiemy, aby aplet uruchomił się w przeglądarce korzystającej z wirtualnej maszyny zgodnej z pierwszą specyfikacją Javy, konieczne jest, żeby podczas jego działania nie była wywoływana żadna metoda, która obecna jest dopiero w Java 2. W szczególności dotyczy to całego pakietu *Swing*[2].

Aplet, który stworzyliśmy, jest w pełni zgodny z wirtualną maszyną w wersji 1.1 bez względu na to, czy jest to VJM z JDK 1.1, czy wirtualna maszyna Javy dołączana do Internet Explorera przez Microsoft. Inaczej wygląda sprawa klasy komponentu `MigajacyPrzycisk`. Proszę zwrócić uwagę, że zdefiniowane przez kreator konstruktory zawierają deklaracje możliwości zgłoszenia wyjątku `HeadlessException` (zobacz listing 6.1), który, możemy się o tym przekonać w dokumentacji JDK, jest dostępny dopiero od wersji 1.4. Należy się zatem spodziewać, że ewentualne wystąpienie tego wyjątku spowoduje całkowite zatrzymanie działania apletu w przeglądarce niekorzystającej z wirtualnej maszyny Java 2. O to możemy się jednak nie obawiać, ponieważ zgłoszenie tego wyjątku wymaga sytuacji dość egzotycznej. Wyjątek ten jest zgłaszany, gdy kod konstruktora odwołuje się do klawiatury, monitora lub myszy w środowisku, w którym takie urządzenia nie istnieją.



Podczas instalacji JDK dołączonego do JBuildera nie jest instalowany plugin Java 2 do przeglądarek. Jeżeli zainstalowaliśmy niezależnie JDK, możliwe jest, że przeglądarki mają taki plugin jednak zainstalowany. Do celów testowania apletów i komponentów niekorzystających z Java 2, jak `MigajacyPrzycisk`, dobrze jest go dezaktywować. W Windows można to zrobić w *Panelu sterowania*, ikona *Java Plug-in*. Aby upewnić się, która wersja wirtualnej maszyny Javy jest uruchamiana przez przeglądarkę, można wykorzystać stworzony specjalne do takich celów aplet `Info` (zobacz rozdział 3.).

Wykorzystanie interfejsu Runnable

Do pilnowania, aby przycisk cyklicznie w równych odstępach czasu zmieniał kolory, najlepiej wykorzystać osobny wątek. Wątek będzie odpowiedzialny nie tylko za odmierzanie czasu, ale również za samą zamianę kolorów. W tym celu możemy ponownie zastosować rozwiązanie użyte w aplecie `PuzzleNaCzas` lub dla odmiany wykorzystać interfejs `Runnable`, nadający korzystającej z niego klasie niektóre właściwości klasy wątku `Thread`.

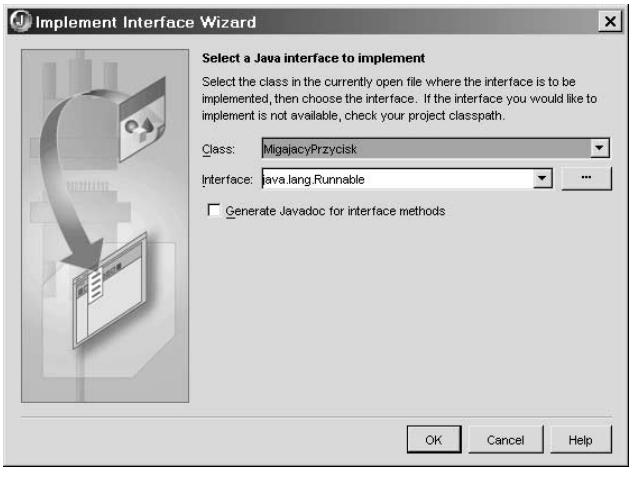
Szczerze mówiąc — niezbyt lubię interfejs `Runnable`. Wolę sam tworzyć klasę wątku i definiować w niej metodę `run` tak, jak robiliśmy to w przypadku `Puzzle`. Jednak ponieważ wszystkiego warto się nauczyć, wbrew moim upodobaniom tym razem skorzystajmy z pomocy tego interfejsu.

Ćwiczenie 6.8.

Aby dodać do klasy `MigajacyPrzycisk` interfejs `Runnable`:

[2] Zobacz uwagi na ten temat w rozdziale 3.

1. W edytorze przełączamy zakładkę plików (nad kodem) na `MigajacyPrzycisk`.
2. Z menu *Wizards* wybieramy *Implement Interface…*. Zobaczymy kreator o nazwie *Implement Interface Wizard* (rysunek 6.4).



Rysunek 6.4. *Kreator dodawania interfejsu do klasy*

3. W rozwijanej liście *Class* znajduje się lista klas z bieżącego pliku, z której należy wybrać tę klasę, do której chcemy dodać interfejs. W naszym pliku zdefiniowana jest wyłącznie klasa `MigajacyPrzycisk`.
4. Kolejna rozwijana lista o nazwie *Interface* pozwala na wybranie interfejsu, który chcemy zaimplementować w klasie.
5. Jeżeli nie znajdziemy go pośród elementów listy, możemy go odszukać naciskając przycisk z trzema kropkami. Pojawi się okno *Search for Interface Class* z drzewem interfejsu. Wybieramy gałąź `java.lang`, a na niej zaznaczamy interfejs `Runnable`. Zamykamy okno dialogowe naciskając *OK*.
6. Dodajemy interfejs do klasy naciskając *OK* w oknie kreatora.

Kreator wprowadził do definicji klasy dwie modyfikacje (listing 6.3). W sygnaturze klasy dodane zostało słowo kluczowe `implements` z nazwą interfejsu, a do definicji klasy dodana została metoda `run` (na razie jedynie zgłaszająca wyjątek domagający się jej implementacji). Metoda `run` zadeklarowana jest w interfejsie `Runnable` i musi być wobec tego zdefiniowana w korzystającej z tego interfejsu klasie. Wykorzystanie interfejsu `Runnable` w klasie `MigajacyPrzycisk` oznacza, że jej metoda `run` może być wykonywana w osobnym wątku. Ale nie stanie się to samo — należy ten wątek własnoręcznie uruchomić[3]. I to uruchomić w taki sposób, żeby wiedział, że jego zadaniem jest uruchomienie metody `run` z klasy migającego przycisku.

---

[3] Częstą pomyłką jest przekonanie, że zdefiniowanie metody `run` w klasie korzystającej z interfejsu `Runnable` oznacza, że zostanie ona automatycznie uruchomiona w osobnym wątku. Niestety nie. Drugą równie częstą pomyłką jest pomysł, że jawne wywołanie metody `run` spowoduje jej wykonywanie w osobnym wątku. Tak również nie jest.

Listing 6.3. *Klasa MigajacyPrzycisk po dodaniu przez kreator interfejsu Runnable*

```
public class MigajacyPrzycisk extends Button implements Runnable
{
  public MigajacyPrzycisk() throws HeadlessException
  {
  }

  public MigajacyPrzycisk(String label) throws HeadlessException
  {
    super(label);
  }

  public void run()
  {
    /**@todo Implement this java.lang.Runnable method*/
    throw new java.lang.UnsupportedOperationException("Method run() not yet
implemented.");
  }
}
```

Ćwiczenie 6.9.

Aby zdefiniować metodę `uruchomWatek`, której zadaniem będzie uruchamianie wątku związanego z klasą `MigajacyPrzycisk`:

1. Definiujemy jako pole klasy prywatną referencję do klasy `Thread` o nazwie `watek` (listing 6.4).
2. Definiujemy prywatną metodę `uruchomWatek` bez argumentów i bez zwracanej wartości. Wewnątrz tej metody tworzymy obiekt klasy `Thread` korzystając z konstruktora, który jako argument pobiera klasę z interfejsem `Runnable`. W naszym przypadku tym argumentem powinna być bieżąca klasa komponentu — używamy zatem referencji `this`. Zwracany przez operator `new` adres zapisujemy do zmiennej `watek`.
3. Dodajemy wywołanie metody `uruchomWatek` do obu konstruktorów komponentu.
4. Usuwamy tymczasową zawartość metody `run` (dwie linie między nawiasami klamrowymi) i umieszczamy tam polecenie zmiany koloru tła przycisku np. na niebieski. Pozwoli nam to upewnić się, czy wątek rozpoczął niezależną pracę wykonując metodę `run` naszego przycisku.
5. Po kompilacji i uruchomieniu apletu (klawisz *F9*) powinniśmy zobaczyć na powierzchni apletu `MigajacyPrzyciskDemo` dwa niebieskie przyciski.

Listing 6.4. *Do automatycznego uruchomienia wątku po stworzeniu obiektu klasy MigajacyPrzycisk korzystamy z pomocniczej prywatnej metody uruchomWatek*

```
public class MigajacyPrzycisk extends Button implements Runnable
{
  private Thread watek=null;

  public MigajacyPrzycisk() throws HeadlessException
  {
```

```
    uruchomWatek();
  }

  public MigajacyPrzycisk(String label) throws HeadlessException
  {
    super(label);
    uruchomWatek();
  }

  private void uruchomWatek()
  {
    watek=new Thread(this);
    watek.start();
  }

  public void run()
  {
    this.setBackground(Color.blue);
  }
}
```

Przypomnijmy podstawowe informacje na temat klasy `Thread`. Obiekt tej klasy reprezentuje osobny niezależny wątek apletu lub aplikacji. Czynności wykonywane przez ten wątek określamy definiując metodę `run` klasy wątku. W typowym przypadku, który poznaliśmy w aplecie `PuzzleNaCzas`, metodę `run` zdefiniowaliśmy rozszerzając klasę `Thread` nadpisując nic nierobiącą metodę zdefiniowaną w klasie bazowej. Teraz metodę `run` mamy zdefiniowaną w klasie `MigajacyPrzycisk`. Prosimy zatem obiekt `watek` o wykonanie właśnie tej metody wywołując konstruktor, w którym jako argument podajemy referencję do obiektu przycisku. Warunkiem jest, że obiekt ten musi korzystać z interfejsu `Runnable`.

Zdefiniowana przez nas metoda `run` na razie zmienia tylko kolor przycisku na niebieski. Zatem jeżeli po uruchomieniu apletu zobaczymy dwa niebieskie przyciski, możemy być pewni, że oba te obiekty uruchomiły osobne wątki i wykonana została metoda `run` na rzecz każdego z nich. Po zakończeniu wykonywania metody `run` praca wątku automatycznie kończy się[4].

Ćwiczenie 6.10.

Aby zdefiniować metodę `run` w taki sposób, żeby implementowała miganie przycisku z ustalonym okresem:

1. Deklarujemy w klasie `MigajacyPrzycisk` prywatne pole klasy o nazwie `okres` (przypominam, że należy ją umieścić wewnątrz klasy, ale poza jej metodami). Inicjujemy ją wartością 1000. Za jednostkę wybieramy milisekundę, czyli tysięczną część sekundy.

```
private long okres=1000;
```

[4] Wyjątkiem jest sytuacja, w której wątek jest tzw. demonem (zob. informację na ten temat w dokumentacji metody `Thread.setDaemon`).

2. Definiujemy metodę `run` w następujący sposób:

```
public void run()
{
  while(watek!=null)
  {
    Color kolorNapisu=this.getForeground();
    Color kolorTla=this.getBackground();

    this.setForeground(kolorTla);
    this.setBackground(kolorNapisu);
    try {watek.sleep(okres);}
    catch(InterruptedException exc){System.err.println(exc.getMessage());}
  }
}
```

Działanie metody `run` opiera się na pętli `while`, która wykonywana jest dopóki, dopóty referencja `watek` ma wartość różną od `null`. Z tego wynika, że aby zakończyć działanie metody `run` i w ten sposób zakończyć działanie wątku, wystarczy zmienić wartość pola `watek` na `null`.

Wewnątrz pętli tworzymy dwie zmienne pomocnicze przechowujące obecne kolory tła i etykiety przycisku. Następnie zamieniamy kolory przycisku miejscami i usypiamy wątek na czas określony przez `okres`. Służąca do tego metoda `Thread.sleep`, którą poznaliśmy już wcześniej, wymaga obsługi wyjątku `InterruptedException` — stąd odpowiednia konstrukcja `try..catch`.

Po uruchomieniu apletu lub w podglądzie na zakładce *Design* powinniśmy zobaczyć migające przyciski, które teraz rzeczywiście migają.

Ćwiczenie 6.11.

Aby po naciśnięciu przycisków umieszczonych na aplecie `MigajacyPrzyciskDemo` zmieniły się kolory, w jakich one migają:

1. Przechodzimy do apletu `MigajacyPrzyciskDemo`. Zapewne jest dostępny na górnej zakładce w oknie edytora. Jeżeli nie — trzeba w oknie projektu dwa razy kliknąć plik *MigajacyPrzyciskDemo.java* znajdujący się w pakiecie `helion.r6`.
2. Przełączamy okno edytora na widok projektowania (zakładka *Design*).
3. Dwukrotnie klikamy migający przycisk `mp1`. Powstanie metoda zdarzeniowa związana z domyślnym zdarzeniem `actionPerformed` tego obiektu.
4. Wewnątrz tej metody umieszczamy polecenia zmieniające kolor tła na zielony, a etykiety na żółty:

```
void mp1_actionPerformed(ActionEvent e)
  {
    mp1.setForeground(Color.yellow);
    mp1.setBackground(Color.green);
  }
```

5. Z drugim przyciskiem (`mp2`) postępujemy podobnie. W jego przypadku zmieniamy kolory na pomarańczowy i różowy:

```
void mp2_actionPerformed(ActionEvent e)
  {
    mp2.setForeground(Color.orange);
    mp2.setBackground(Color.pink);
  }
```

6. Kompilujemy i uruchamiamy aplet (klawisz *F9*).

7. Aby zmienić kolory migających przycisków, należy tylko kliknąć każdy z nich.

W tym ćwiczeniu nie zmieniliśmy właściwości samych komponentów `MigajacyPrzycisk`. Związaliśmy jedynie ze zdarzeniami dwóch konkretnych obiektów tego typu metody zmieniające kolory tych dwóch obiektów.

To ćwiczenie pokazuje również, że dobrze nam znane metody ustalania koloru tła i etykiety przycisku nadal działają pozwalając teraz na wybór kolorów, w jakich miga przycisk. Ponadto przekonaliśmy się, że nie zmienił się także sposób obsługi zdarzeń w nowym komponencie `MigajacyPrzycisk`.

Ćwiczenie 6.12.

Aby do migającego przycisku dodać możliwość opóźnienia rozpoczęcia migania po utworzeniu obiektu:

1. Wracamy do klasy `MigajacyPrzycisk` (zakładka na górze edytora).

2. Deklarujemy kolejne globalne pole o nazwie `opoznienie` zainicjowane wartością 3000:

```
private long opoznienie=3000;
```

3. Uzupełniamy metodę `run` w następujący sposób (należy dopisać tylko wyróżnione dwie linie kodu):

```
public void run()
{
  try {watek.sleep(opoznienie);}
  catch(InterruptedException exc){System.err.println(exc.getMessage());}

  while(watek!=null)
    {
      Color kolorNapisu=this.getForeground();
      Color kolorTla=this.getBackground();

      this.setForeground(kolorTla);
      this.setBackground(kolorNapisu);
      try {watek.sleep(okres);}
      catch(InterruptedException exc){System.err.println(exc.getMessage());}
    }
}
```

1. Kompilujemy kod klasy naciskając *Ctrl+F9*.

Idea jest prosta. Należy uśpić wątek na czas określony przez `opoznienie`, zanim rozpocznie się pętla zmieniająca kolory. Aby zobaczyć efekt tej modyfikacji, można przełączyć edytor w widok projektowania lub uruchomić aplet naciskając *F9*.

Właściwości komponentów

W rozdziale 2. kontrastowaliśmy zmienne i funkcje z polami i metodami klas. Tylko te drugie są dostępne w Javie. W językach związanych z programowaniem RAD istnieje zazwyczaj jeszcze jeden element klas — właściwość. Konstrukcja ta tworzy pewnego rodzaju pomost między metodami a polami klasy. Jej głównym zadaniem jest przygotowanie klasy komponentu do współpracy z narzędziami typu inspektor.

Jak przekonaliśmy się w rozdziale 2. na przykładzie klasy `Ulamek`, korzystanie z metod dostępu typu `set..` jest właściwsze od bezpośredniego modyfikowania pól obiektu. Możliwa jest w ten sposób kontrola ustalanej wartości pola, a ponadto korzystanie z metody pozwala na synchronizację pozostałych właściwości obiektu, jeżeli te są w jakiś sposób zależne od modyfikowanego pola. Konstrukcja właściwości klasy wykorzystuje te korzyści wynikające z korzystania z metod dostępu podczas projektowania RAD.

Jak wspomniałem przed chwilą, do języków związanych z narzędziami typu RAD (np. Delphi i C++ Builder) dodawana jest konstrukcja, która udostępnia tzw. właściwość obiektu wiążąc parę metod `get..` i `set..` związanych z daną właściwością. W Object Pascalu należy właściwości zadeklarować osobno za pomocą słowa `property` (do C++ w C++ Builderze Borland dodał w tym celu niestandardowe słowo kluczowe `__property`). Dla programisty używającego klasy korzystanie z właściwości[5] niczym nie różni się od modyfikowania pola. Jednak różnica istnieje i to poważna; zmiana wartości właściwości powoduje wywołanie metody `set..`, a jej odczytanie — metody `get...`. W Delphi i C++ Builderze właściwości obiektów mogą być dostępne w *Object Inspector*, jeżeli zadeklarowane są jako `published` — czyli opublikowane (nowy zakres dostępu).

W Javie sytuacja jest nieco prostsza. Po pierwsze, nie istnieje specjalne słowo kluczowe pozwalające na konstruowanie właściwości. Przez to tworząc „ręcznie" kod źródłowy nadal mamy dostęp tylko do metod `get..` i `set..`. Po drugie, wszystkie środowiska programistyczne RAD Javy, w tym JBuilder, rozpoznają właściwości tylko po odpowiednio zdefiniowanej parze metod dostępu `get..` i `set..` i dodają tak określoną właściwość do odpowiedników inspektora. W tej sytuacji z właściwości możemy korzystać jedynie podczas projektowania RAD.

W kolejnych ćwiczeniach zajmiemy się tworzeniem metod dostępu do pól `okres` i `opoznienie`, które będą rozpoznawane jako określające właściwości komponentu i zostaną dodane do właściwości widocznych w inspektorze.

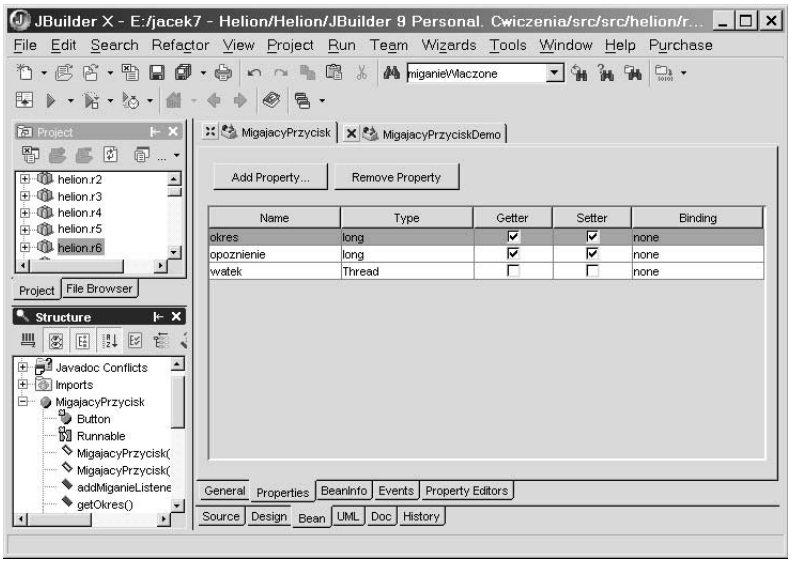
[5] Należy zwrócić uwagę, że w czystym C++ właściwością nazywa się zazwyczaj to, co w Javie i Object Pascalu jest polem. Standard C++ nie zna konstrukcji właściwości. Jest on dodany np. w C++ Builderze jako element niestandardowy (dlatego słowo kluczowe `property` ma tu dwa znaki podkreślenia).

Można stworzyć te metody ręcznie, ale o wiele szybciej można je zdefiniować korzystając z minimalnego wsparcia, jakie w wersji JBuilder Foundation jest dostępne przy budowaniu komponentów JavaBean (szczątkowe fragmenty *BeansExpress*).

Ćwiczenie 6.13.

Aby przygotować metody pozwalające na czytanie i modyfikację właściwości `okres` i `opoznienie`:

1. Zmieniamy zakładkę na górze okna edytora na *MigajacyPrzycisk*. Jeżeli nie jest dostępna, to korzystając z okna projektu, otwieramy plik *MigajacyPrzycisk.java* z pakietu `helion.r6`.
2. Dolną zakładkę edytora zmieniamy na *Bean*[6], a w niej wybieramy podzakładkę *Properties*.
3. Zobaczymy tabelę z zadeklarowanymi polami klasy: `watek`, `okres` i `opoznienie` (rysunek 6.5) Kolumny tej tabeli to: nazwa właściwości (*Name*), zadeklarowany typ (*Type*), możliwość pobrania wartości przez metodę `get...` (*Getter*), możliwość zmodyfikowania wartości przechowywanej przez właściwość metodą `set...` (*Setter*) oraz sposób związania (*Binding*).



Rysunek 6.5. *Jedyny element narzędzia BeansExpress dostępny w wersji Foundation*

4. Przy właściwościach `okres` i `opoznienie` zaznaczamy pozycje w kolumnach *Getter* i *Setter*. W kodzie klasy powstaną odpowiednie metody `getOkres`, `setOkres`, `getOpoznienie` i `setOpoznienie`. W ten sposób zwykłe pole zmieni się we właściwość.

[6] Zakładka *Bean* ukrywa wspomniane wcześniej narzędzie *BeansExpress*. Niestety, w wersji Foundation jest ono nieaktywne. Jedyny element, który jest dostępny, wykorzystujemy w tym ćwiczeniu.

Jeżeli zajrzymy do kodu komponentu na zakładce *Source*, zauważymy nowe metody `getOkres`, `setOkres`, `getOpoznienie` i `setOpoznienie`. Ich budowa jest typowa dla metod dostępu do właściwości. Definiowaliśmy bardzo podobne w klasie `Ulamek` w rozdziale 2.

Możliwość automatycznego tworzenia metod `get..` i `set..` jest jedynie drobnym fragmentem możliwości narzędzia o nazwie *BeanExpress* w pełni dostępnego na zakładce *Bean* tylko w wersjach komercyjnych JBuildera oraz przez miesiąc po zainstalowaniu jego darmowej wersji. Zakładka *Properties*, którą teraz wykorzystaliśmy, jest w wersji Foundation jedyną, jaka działa. Pozostałe zakładki po upływie miesiąca przełączane są w tryb tylko do odczytu.

Ćwiczenie 6.14.

Aby zobaczyć nowe właściwości:

1. Z menu *Project* wybieramy polecenie *Rebuild Project "helion.jpx"*.

Naciśnięcie kombinacji klawiszy *Ctrl+F9* odpowiadającej poleceniu *Make Project "helion.jpx"* z tego samego menu nie wystarczy, aby inspektor zobaczył nowe zdarzenia.

Obecność metod `get`*Nazwa* i `set`*Nazwa* o odpowiednich sygnaturach jest wystarczającym sygnałem dla inspektora, żeby dodał do listy właściwość *nazwa*. W naszym przypadku będą to właściwości `okres` i `opoznienie`.

Ćwiczenie 6.15.

Aby w aplecie `MigajacyPrzyciskDemo` za pomocą inspektora zmienić okres migania przycisku `mp1` na 500 milisekund:

1. W edytorze przechodzimy do klasy apletu `MigajacyPrzyciskDemo`.
2. Zmieniamy zakładkę na *Design*.
3. Zaznaczamy komponent `mp1`.
4. W oknie inspektora zobaczymy wszystkie właściwości komponentu, a między nimi trzy nowe właściwości `okres` i `opoznienie`. Zmieńmy właściwość `okres` zaznaczonego komponentu na 500.

Po wykonaniu ćwiczenia przycisk zacznie migać znacznie szybciej (dwa razy na sekundę).

Ćwiczenie 6.16.

Aby do komponentu dodać metody kontrolujące uruchamianie i zatrzymywanie wątku, a w konsekwencji włączać i wyłączać miganie przycisku:

1. W edytorze przechodzimy do klasy `MigajacyPrzycisk`, zakładka *Source*.
2. Do klasy komponentu dodajemy metodę `setMiganieWlaczone`:

```
public void setMiganieWlaczone(boolean MiganieWlaczone)
{
  if (MiganieWlaczone)
  {
    if (watek==null) uruchomWatek();
  }
  else watek=null;
}
```

3. Dodajemy również metodę `getMiganieWlaczone`:

```
public boolean getMiganieWlaczone()
{
  return (watek!=null);
}
```

4. Nie zmieniając zakładki klasy ani zakładki *Source*, kompilujemy projekt (*Ctrl+F9*).

Po kompilacji w inspektorze pojawi się nowa właściwość o nazwie `miganieWlaczone`, a przy niej rozwijana lista zawierająca dwie możliwe wartości *True* i *False*[7]. Warto wiedzieć, że w przypadku właściwości logicznych nazwa metody pobierającej wartość właściwości może rozpoczynać się od `is...` zamiast `get...`.

Ćwiczenie 6.17.

Aby zatrzymać miganie przycisku w przypadku uruchomienia metody standardowej `stop` apletu `MigajacyPrzyciskDemo` oraz ponowić po wywołaniu metody `start`:

1. Przechodzimy do edycji klasy `MigajacyPrzyciskDemo`, zakładka *Source*.
2. Podczas tworzenia apletu `MigajacyPrzyciskDemo` zaznaczyliśmy opcję *Generate standard methods*, dzięki czemu kreator dodał do klasy apletu metody standardowe `start`, `stop` i `destroy`[8].
3. Do metody `stop` należy dodać wywołanie metod `setMiganieWlaczone` każdego z przycisków z argumentem `false`:

```
public void stop()
{
  mp1.setMiganieWlaczone(false);
  mp2.setMiganieWlaczone(false);
}
```

4. Analogicznie w metodzie `start` wywołujemy te same metody z argumentem `true`:

---

[7] Proszę zauważyć, że w klasie `MigajacyPrzycisk` nie mamy zadeklarowanego pola typu logicznego o nazwie `miganieWlaczone`. Nie jest ono konieczne. Dla *Inspektora* liczą się tylko metody `setMiganieWlaczone` i `getMiganieWlaczone`, na ich podstawie tworzy właściwość o nazwie `miganieWlaczone`.

[8] Można je stworzyć również samodzielnie definiując samodzielnie bezargumentowe i niezwracające wartości metody publiczne, np. `public void stop(){}`. Więcej informacji na ten temat znajduje się w rozdziałach 1. oraz 3.

```
public void start()
{
  mp1.setMiganieWlaczone(true);
  mp2.setMiganieWlaczone(true);
}
```

Definiowanie dodatkowych zdarzeń

Czym są zdarzenia?

Najłatwiej odpowiedzieć na to pytanie podając przykłady zdarzeń. Zdarzeniem może być kliknięcie przycisku przez użytkownika, może być nim również upłynięcie określonego czasu, rozpoczęcie lub zakończenie migania w naszym komponencie. Zdarzeniem może być każda sytuacja, której wystąpienie można sprawdzić za pomocą parametrów dostępnych w programie.

W przypadku naszego komponentu `MigajacyPrzycisk` możemy zdefiniować na przykład zdarzenie związane z rzeczywistym rozpoczęciem migania po odczekaniu okresu określonego przez `opoznienie`. Możemy również wywołać zdarzenie, gdy przycisk kończy miganie oraz przy każdym mignięciu. Zdefiniujemy wobec tego trzy typy zdarzeń.

Jak działa mechanizm zdarzeń w Javie?

Jest to dość skomplikowane do wytłumaczenia, więc najpierw opiszę podobny przykład z życia. Załóżmy, że w pewnej uczelni przeprowadzany jest nabór na studia. Pisemne egzaminy już się odbyły i kandydaci na studentów czekają w domach na informację o tym, czy dostali się na studia i czy przyznane zostało im miejsce w akademiku. Pierwsza informacja zostanie wysłana w chwili ogłoszenia wyników, a druga po zakończeniu prac komisji rozdzielającej miejsca w akademikach. Zatem naszymi zdarzeniami są moment ogłoszenia wyników i moment zakończenia prac komisji ds. akademików. Zdarzenia te zostaną opisane w wysłanych do studenta oddzielnych listach, w których przekazana zostanie informacja o jego wynikach i o przyznaniu lub nie miejsca w akademiku. Sekretariat uczelni musi oczywiście posiadać listę kandydatów i tylko do nich wysyłana jest taka informacja. Żeby obraz jeszcze nieco skomplikować, przypomnę, że student nie dostaje listu do ręki. List jest wkładany do jego skrzynki pocztowej. Gdy student odbierze zawiadomienie z uczelni, w zależności od wyniku „uruchamiana jest jedna z jego metod": radości lub smutku.

Podobnie realizowane są zdarzenia w Javie. Posłużmy się przykładem naszego komponentu `MigajacyPrzycisk`. Załóżmy, że `MigajacyPrzycisk` ma mieć możliwość informowania innych obiektów o rozpoczęciu i zakończeniu migania. W odpowiednich momentach migający przycisk powinien rozesłać do wszystkich zainteresowanych tym faktem obiektów informację o wystąpieniu zdarzenia. Informacja ta będzie oczywiście również obiektem — jak wszystko w Javie — musi więc być opisana przez klasę (powiedzmy, że będzie to zdefiniowana przez nas klasa `MiganieEvent`; ta sama klasa będzie wykorzystywana w obu zdarzeniach). Ściślej rzecz biorąc, obiekt zostanie przekazany nie bezpośrednio do komponentu, ale do jego „skrzynki na listy" — obiektu towarzyszącego

nazywanego nasłuchiwaczem i tylko pod warunkiem, że nasłuchiwacz wcześniej zgłosił migającemu przyciskowi, że chce być o zdarzeniu informowany (nasłuchiwacz musi znajdować się na odpowiedniej liście w migającym przycisku). Dopiero z nasłuchiwacza jest wywoływana metoda obiektu informowanego, która jest nazywana metodą zdarzeniową[9].

Jeżeli w migającym przycisku zdefiniowane są dwa zdarzenia (rozpoczęcie i zakończenie migania), to nasłuchiwacz powinien posiadać dwie metody odpowiadające obu zdarzeniom. Student posiada tylko jedną skrzynkę na listy, ale wie, jaki typ listu otrzymał — na tej samej zasadzie w naszym przykładzie obiekt informowany o zdarzeniu ma tylko jeden nasłuchiwacz, a migający przycisk wywołuje odpowiednią metodę nasłuchiwacza w zależności od tego, które zdarzenie wystąpiło.

Musimy wobec tego posiadać dwie klasy towarzyszące komponentowi `MigajacyPrzycisk`. Pierwsza to interfejs `MiganieListener` — obiekty tego typu to nasłuchiwacze, które zdefiniowane w otoczeniu migającego przycisku odbierają od migającego przycisku informacje o wystąpienia zdarzenia. Drugą jest klasa-nośnik informacji o zdarzeniu `MiganieEvent` — obiekty tego typu są przekazywane z migającego przycisku do każdego zarejestrowanego w migającym przycisku nasłuchiwacza.

Podsumujmy. Aby np. aplet `MigajacyPrzyciskDemo` był informowany o zakończeniu migania przez `MigajacyPrzycisk`, musi mieć towarzyszący mu obiekt-nasłuchiwacz implementujący interfejs `MiganieListener`. Nasłuchiwacz musi zgłosić migającemu przyciskowi, że chce być dopisany do listy obiektów informowanych o zakończeniu migania (metoda `addActionListener` migającego przycisku). Te czynności na szczęście wykonuje już za nas JBuilder. Wystarczy w inspektorze w zakładce *Event* dwa razy kliknąć odpowiednie zdarzenie. To jest dobrze nam znany element programowania RAD, który zwalnia nas z myślenia o całym mechanizmie zdarzeń tworząc połączenie między określonym zdarzeniem komponentu a metodą zdarzeniową apletu. Musimy tylko postarać się o to, żeby odpowiednie zdarzenia pojawiły się w inspektorze. I do tego właśnie doprowadzą kolejne ćwiczenia.

Interfejs nasłuchiwacza i klasa opisująca zdarzenie

A więc do dzieła! Dodajmy do ziarenka zdarzenie wywoływane, gdy przycisk zacznie migotać z uwzględnieniem opóźnienia, zdarzenie wywoływane, gdy miganie zostanie przerwane, oraz zdarzenie wywołane przy każdym mignięciu. Zdefiniujemy własną klasę zdarzenia `MiganieEvent` przenoszącą informacje o zdarzeniu oraz własny interfejs nasłuchiwacza `MiganieListener`, którego zadaniem jest czekanie na wystąpienie zdarzenia. Tak naprawdę własną klasę zdarzenia zdefiniujemy bardziej dla pokazania, jak to się robi, niż z rzeczywistej potrzeby. W poniższych przykładach bez problemu można by posłużyć się też standardową klasą zdarzeń `java.util.EventObject`.

[9] Proponuję zajrzeć do rozdziału 4., gdzie samodzielnie definiowaliśmy nasłuchiwacze w aplecie `Puzzle`. To powinno nieco pomóc w zrozumieniu mechanizmu zdarzeń.

Ćwiczenie 6.18.

Aby zdefiniować klasę `MiganieEvent`, czyli nośnik informacji o zdarzeniu:

1. W menu *File* JBuildera wybieramy pozycję *New Class*...
2. W kreatorze klasy ustalamy:
   a) nazwę pakietu: `helion.r6`,
   b) nazwę klasy: `MiganieEvent`,
   c) klasę bazową: `java.util.EventObject`.
3. Zaznaczamy tylko dwie opcje: *Public* oraz *Override superclass constructor*.
4. Klikamy *OK*.

Powstanie skromna klasa, którą w przyszłości będzie można rozbudować tak, aby przenosiła np. szczegółowe informacje o zdarzeniu i stanie komponentu w trakcie jego wystąpienia. Na razie jedyną ważną przekazywaną przez tę klasę informacją jest właściwość `source` zdefiniowana w `EventObject`, która identyfikuje obiekt, w którym wystąpiło zdarzenie. Właściwość ta jest ustalana w momencie tworzenia obiektu zdarzenia; podajemy ją w argumencie jedynego konstruktora.

Ćwiczenie 6.19.

Aby zdefiniować interfejs nasłuchiwacza `MiganieListener`:

1. Wybieramy pozycję *New* z menu *File*.
2. W galerii obiektów na stronie *General* zaznaczamy ikonę *Interface* i naciskamy *OK*.
3. W kreatorze interfejsu wpisujemy:
   a) nazwę pakietu: `helion.r6`,
   b) nazwę interfejsu: `MiganieListener`,
   c) nazwę interfejsu bazowego: `java.util.EventListener`.
4. Naciskamy *OK*.
5. Za pomocą edytora dodajemy do interfejsu nasłuchiwacza deklaracje publicznych metod, które będą wywoływane przez migający przycisk w momencie wystąpienia zdarzenia[10]. Metody nie zwracają żadnej wartości, a ich argumentem jest obiekt opisujący zdarzenie `MiganieEvent`. Deklarujemy trzy metody: `miganieRozpoczete`, `miganieZakonczone`, `miganieMigniecie` (listing 6.5).

[10] Więcej informacji o deklaracji metod w interfejsach znajduje się w rozdziale 2.

Listing 6.5. *Interfejs MiganieListener*

```
public interface MiganieListener extends EventListener
{
  public void miganieRozpoczete(MiganieEvent e);
  public void miganieZakonczone(MiganieEvent e);
  public void miganieMigniecie(MiganieEvent e);
}
```

Realizacja wywoływania zdarzeń w komponencie

Komponent JavaBean musi przechowywać listę obiektów będących nasłuchiwaczami, które mają być powiadamiane o wystąpieniu poszczególnych zdarzeń. Najprościej zrealizować taką listę za pomocą klasy `Vector`. Poza tym konieczne są metody dopisujące nasłuchiwacze do listy i usuwające z niej.

Ćwiczenie 6.20.

Aby do komponentu `MigajacyPrzycisk` dodać listę zarejestrowanych nasłuchiwaczy:

1. Przechodzimy w edytorze do klasy komponentu (zakładka *MigajacyPrzycisk*). Jeżeli jest to konieczne, otwieramy plik *MigajacyPrzycisk.java* w oknie projektu.
2. Importujemy klasę `Vector` poleceniem umieszczonym na początku pliku *MigajacyPrzycisk.java* za deklaracją nazwy pakietu, ale przed deklaracją klasy `MigajacyPrzycisk`:

   ```
   import java.util.Vector;
   ```

3. Następnie wewnątrz klasy `MigajacyPrzycisk` deklarujemy listę nasłuchiwaczy dodając do niej pole:

   ```
   private Vector listaMiganieListeners;
   ```

Prawdę mówiąc, to dodaliśmy do klasy komponentu jedynie referencję, a nie obiekt listy. Rzeczywisty obiekt zostanie stworzony w metodzie, którą zdefiniujemy w następnym ćwiczeniu.

Ćwiczenie 6.21.

Aby do klasy komponentu dodać metody pozwalające na zarejestrowanie i wyrejestrowanie nasłuchiwacza:

1. Pozostajemy w kodzie klasy `MigajacyPrzycisk`.
2. Dodajemy do klasy definicję metody `addMiganieListener` dodającej nasłuchiwacz podany w argumencie do listy zdefiniowanej w poprzednim ćwiczeniu:

   ```
   public synchronized void addMiganieListener(MiganieListener listener)
   {
     if (listaMiganieListener==null) listaMiganieListener=new Vector(2);
   ```

```
    if (!listaMiganieListener.contains(listener))
          listaMiganieListener.addElement(listener);
    }
```

3. Dodajemy także definicję metody `removeMiganieListener`, która usuwa podany w argumencie nasłuchiwacz z listy:

```
public synchronized void removeMiganieListener(MiganieListener listener)
{
  if (listaMiganieListener!=null && listaMiganieListener.contains(listener))
        listaMiganieListener.removeElement(listener);
}
```

4. Z menu *Project* wybierz *Rebuild Project "helion.jpx"*. Naciśnięcie kombinacji klawiszy *Ctrl+F9* nie wystarczy, aby inspektor zobaczył nowe zdarzenia.

Definiując metody, posłużyliśmy się metodami klasy `Vector`: `addElement` i `removeElement` służącymi odpowiednio do dodania do listy i usunięcia z niej elementu podanego jako argument. Wykorzystaliśmy także metodę `Vector.contains` sprawdzającą, czy obiekt podany w argumencie znajduje się na liście.

Metoda `addMiganieListener` sprawdza najpierw, czy za referencją `listaMiganieListener` kryje się obiekt, czy jest on może jeszcze niezdefiniowany. Jeżeli obiektu nie ma, tzn. referencja `listaMiganieListener` ma wartość `null`, tworzymy go nadając liście wstępny rozmiar równy 2. W następnej linii sprawdzamy metodą `Vector.contains`, czy na liście jest już nasłuchiwacz, który chcemy dodać. Jeżeli nie, dodajemy go. Metoda pozwalająca usuwać elementy z listy nasłuchiwaczy `removeMiganieListener` sprawdza, czy lista istnieje oraz czy element usuwany znajduje się rzeczywiście na liście. Jeżeli oba warunki są spełnione, element jest usuwany metodą `Vector.removeElement`.

Modyfikator `synchronized` został użyty, żeby zapobiec sytuacji, w której dwa wątki równocześnie próbują modyfikować listę nasłuchiwaczy. Użycie tego modyfikatora informuje wirtualną maszynę Javy, że metoda ma być wykonywana tylko przez jeden wątek w danym czasie. Drugi musi poczekać w „kolejce".

Dodanie tych metod do klasy ziarenka jest sygnałem dla inspektora, że powinien pokazać nowe zdarzenie na zakładce *Events*. Konieczne jest, co podkreśliłem w punkcie 4. ćwiczenia, wywołanie polecenia *Rebuild Project "helion.jpx"* z menu *Project*, a nie tylko polecenia *Make Project "helion.jpx"* (lub prowadzącego do tego samego naciśnięcia kombinacji klawiszy *Ctrl+F9*). Nazwy zdarzeń odpowiadają nazwom metod zdefiniowanych w interfejsie nasłuchiwacza. W inspektorze pojawią się więc zdarzenia: `miganieRozpoczete`, `miganieZakonczone` i `miganieMigniecie`.

Ćwiczenie 6.22.

Aby w aplecie testowym stworzyć metodę zdarzeniową odpowiadającą zdarzeniu `miganieMigniecie`:

1. Przejdźmy w edytorze do apletu `MigajacyPrzyciskDemo`.
2. W widoku projektowania (zakładka *Design* edytora) zaznaczmy migający przycisk `mp1`.

3. Stwórzmy metodę zdarzeniową związaną ze zdarzeniem `miganieMigniecie` tak, jak robiliśmy to wiele razy:

   a) zmieniamy w inspektorze zakładkę na *Events*,

   b) odnajdujemy zdarzenie `miganieMigniecie`,

   c) klikamy dwukrotnie w jego polu edycyjnym.

Do metody `jbInit` zostało dodane wywołanie zdefiniowanej przez nas przed chwilą metody rejestrującej nasłuchiwacz w komponencie `mp1`:

```
mp1.addMiganieListener(new MigajacyPrzyciskDemo_mp1_miganieAdapter(this));
```

Pliku apletu został również wzbogacony o deklarację klasy zewnętrznej określającej nasłuchiwacz `MigajacyPrzyciskDemo_mp1_miganieAdapter` wykorzystującej zdefiniowany przez nas interfejs `MigajacyListener`. W tej klasie zostały zdefiniowane trzy metody, które zadeklarowaliśmy jako abstrakcyjne w interfejsie. W tej chwili nietrywialną wartość ma tylko metoda `miganieMigniecie`, która wywołuje metodę zdarzeniową `mp1_miganieMigniecie` apletu.

Ćwiczenie 6.23.

Aby dodać do metody zdarzeniowej z poprzedniego ćwiczenia polecenia zmieniające kolor tła apletu:

1. Do stworzonej w poprzednim ćwiczeniu przez JBuildera metody zdarzeniowej dopisujemy polecenia zaznaczone na listingu 6.6.

Listing 6.6. *Metoda zdarzeniowa zmieniająca kolory tła apletu*

```
void mp1_miganieMigniecie(MiganieEvent e)
{
  MigajacyPrzycisk mp=(MigajacyPrzycisk)e.getSource();
  this.setBackground(mp.getForeground());
  this.setForeground(mp.getBackground());
}
```

W metodzie wykorzystałem informacje o kolorach obiektu `mp1` wykorzystując referencję do tego obiektu, która została przekazana we właściwości `source` w obiekcie zdarzenia `e`. Obiekt ten przekazany został do nasłuchiwacza, a z niego do metody zdarzeniowej. Metoda `getSource` (z ang. *pobierz źródło*) tego obiektu zwraca referencję do obiektu wywołującego zdarzenie `mp1`. My wiemy, że źródło jest klasy `MigajacyPrzycisk`, więc wykonujemy odpowiednie rzutowanie. Dzięki znajomości referencji do `mp1`, a przez to informacji o jej kolorach, mogliśmy metodę zaprogramować w taki sposób, żeby kolory apletu były zamienione względem kolorów przycisku.

Do działania całego mechanizmu brakuje już tylko jednego elementu. Musimy w odpowiednich miejscach metody `run` komponentu wywołać zdarzenia, co znaczy, że musimy wywołać odpowiednie metody wszystkich nasłuchiwaczy (`miganieRozpoczete`, `miganie➥Zakonczone` i `miganieMigniecie`) znajdujących się na liście `listaMiganieListener`.

Ćwiczenie 6.24.

Aby zdefiniować metody informujące nasłuchiwacze typu `miganieListener` o wystąpieniu zdarzeń w migającym przycisku:

1. Wracamy do klasy komponentu (zakładka *MigajacyPrzycisk* edytora).
2. Dodajemy do klasy metody z listingu 6.7.

Listing 6.7. *Metody informujące nasłuchiwacze o wystąpieniu zdarzenia*

```
protected void wywolajMiganieRozpoczete(MiganieEvent e)
{
  if (listaMiganieListener == null) return;
  for (int i = 0; i < listaMiganieListener.size(); i++)
    ((MiganieListener) listaMiganieListener.elementAt(i)).miganieRozpoczete(e);
}

protected void wywolajMiganieZakonczone(MiganieEvent e)
{
  if (listaMiganieListener == null) return;
  for (int i = 0; i < listaMiganieListener.size(); i++)
    ((MiganieListener) listaMiganieListener.elementAt(i)).miganieZakonczone(e);
}

protected void wywolajMiganieMigniecie(MiganieEvent e)
{
  if (listaMiganieListener == null) return;
  for (int i = 0; i < listaMiganieListener.size(); i++)
    ((MiganieListener) listaMiganieListener.elementAt(i)).miganieMigniecie(e);
}
```

Pierwsza metoda będzie wykorzystana do powiadomienia nasłuchiwaczy o rozpoczęciu migania, druga o zakończeniu, a trzecia o każdym mignięciu.

W każdej metodzie sprawdzamy najpierw, czy lista `listaMiganieListener` jest w ogóle zdefiniowana, czyli czy jest kogo powiadamiać. Jeżeli lista istnieje, to na rzecz każdego jej elementu wywołujemy odpowiednio metody `miganieRozpoczete`, `miganieZakonczone` lub `miganieMigniecie` podając jako argument obiekt typu `MiganieEvent`.

Aby uruchomić cały zdefiniowany w tym paragrafie mechanizm zdarzeń, należy teraz umieścić wywołanie metod `wywolajMiganieRozpoczete`, `wywolajMiganieZakonczone` i `wywolajMiganieMigniecie` w odpowiednich miejscach komponentu `MigajacyPrzycisk`. Jakie są to miejsca? Wszystkie metody wywołamy z metody `run` migającego przycisku. Metodę `wywolajMiganieRozpoczete` wywołamy po odczekaniu okresu określonego przez właściwość `opoznienie`. Metodę `wywolajMiganieMigniecie` wywołamy przy każdej zmianie koloru tła i napisu w pętli `while`. Natomiast ostatnią metodę wywołujemy, jeżeli pętla `while` zostanie przerwana.

Ćwiczenie 6.25.

Aby umieścić w klasie komponentu metody `wywolaj...`:

1. Do metody `run` dodajemy polecenia zaznaczone na listingu 6.8.

Listing 6.8. *Dodawanie do metody run wywoływania zdarzeń*

```
public void run()
{
  try {watek.sleep(opoznienie);}
  catch(InterruptedException exc){System.err.println(exc.getMessage());}
  wywolajMiganieRozpoczete(new MiganieEvent(this));

  while(watek!=null)
  {
    Color kolorNapisu=this.getForeground();
    Color kolorTla=this.getBackground();

    this.setForeground(kolorTla);
    this.setBackground(kolorNapisu);
    wywolajMiganieMigniecie(new MiganieEvent(this));
    try {watek.sleep(okres);}
    catch(InterruptedException exc){System.err.println(exc.getMessage());}
  }

wywolajMiganieZakonczone(new MiganieEvent(this));
}
```

Teraz możemy skompilować `MigajacyPrzyciskDemo` i zobaczyć, że aplet miga razem z przyciskiem, a więc mechanizm nowych zdarzeń komponentu działa prawidłowo.

* * *

Na tym zakończymy budowanie komponentu `MigajacyPrzycisk`. Uzyskaliśmy prosty, ale bardzo elegancki komponent z trzema nowymi właściwościami: `okres`, `opoznienie` i `miganieWlaczone` oraz zdarzeniami pozwalającymi śledzić jego działanie. W dalszej części tego rozdziału wiedzę na temat ziarenek uzupełnimy jeszcze informacjami na temat ich instalowania w JBuilderze oraz przygotowania do dystrybucji.